

A TYPED PROGRAMMING LANGUAGE  
The Semantics of Rank Polymorphism

JUSTIN SLEPAK

*A dissertation submitted in partial  
fulfillment of the requirements for the  
degree of Doctor of Philosophy to the  
faculty of the*

KHOURY COLLEGE OF COMPUTING SCIENCES  
Northeastern University  
Boston, Massachusetts

July 3, 2020




A Typed Programming Language:

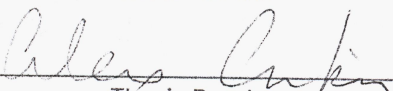
Thesis Title: The Semantics of Rank Polymorphism

Author: Justin Slepak

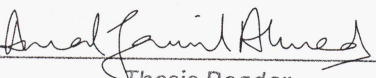
PhD Thesis Approval to complete all degree requirements for the PhD in Computer Science.

  
Thesis Advisor

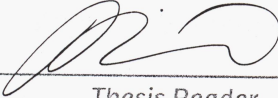
2020/8/4  
Date

  
Thesis Reader

5/28/20  
Date

  
Thesis Reader

5/28/20  
Date

  
Thesis Reader

5/29/20  
Date

\_\_\_\_\_  
Thesis Reader

\_\_\_\_\_  
Date

**KHOURY COLLEGE APPROVAL:**

\_\_\_\_\_  
Associate Dean for Graduate Programs

\_\_\_\_\_  
Date

**COPY RECEIVED BY GRADUATE STUDENT SERVICES:**

\_\_\_\_\_  
Recipient's Signature

\_\_\_\_\_  
Date

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI Website.

**Employment upon Graduation** - *to be turned in to the Graduate Office  
(does not need to be included with your electronic dissertation submission)*

Name: Justin Slepak

Graduation Date: August 27, 2020

Please choose **one**, and indicate location/company and the position

Research - Academic or Industrial: Research Scientist, Facebook  
\_\_\_\_\_  
\_\_\_\_\_

Development: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

No firm commitment, but possible options include: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

## ABSTRACT

---

In the rank-polymorphic programming model, all functions operate on aggregate data of arbitrarily high *rank*, or number of dimensions. During function application, an argument array is split into *cells*, the individual components the function expects to consume. For example, an RGB-to-greyscale pixel transform operates on each vector in an arbitrarily large array. The aggregate structure surrounding the cells, called the *frame*, serves as the iteration space for cell-wise function application. The programming model was first developed by Iverson with the language APL [43], but it struggled with a barrier to efficient compilation: Loop nesting structure is derived from data computed at run time.

This dissertation presents the design and formal semantics of Remora, a higher-order, rank-polymorphic programming language with a static type system which identifies the shape of run-time data. This overview is followed by formal semantics for a core language. Remora’s static semantics ascribes to each expression a type which describes the shape of the resulting array. Quantification over the shape of cells and the type of atoms within an array is explicit, but the polymorphism over frames is entirely implicit. That is, a function’s type only describes its cell-level behavior, while implicit iteration—which is common to all functions—is identified by typing rules. A type-driven dynamic semantics determines the iteration space for functions applied to computed array data, and a type soundness theorem ensures that the types—and shapes—asccribed to expressions match those of their eventual results.

While frame polymorphism is instantiated implicitly in Remora’s formal semantics, explicitly instantiating cell polymorphism is a severe annotation burden. For example, a vector-mean function can be used on a  $3 \times 5 \times 4$  array with no explanation that the array is a  $3 \times 5$  frame, but the function must be explicitly instantiated to operate on vectors of length 4. That burden is alleviated by a bidirectional typing system which uses a novel constraint solver for the theory of array shapes to identify implicit dimension and shape arguments. The vector-mean function can then be applied directly to the  $3 \times 5 \times 4$  array, with bidirectional rules elaborating to code which explicitly instantiates it for 4-vector cells.

Two translation steps link Remora’s formal semantics to conventional rank-monomorphic languages with explicit iteration. While Remora’s dynamic semantics relies heavily on run-time type information, a type erasure pass can change from carrying full type information in dynamically created closures and arrays to describing argument and iteration-space shapes statically at sites. With that shape information at each call site, the program can be translated from using rank-polymorphic function calls to rank-monomorphic explicit iteration.



*Who should I thank?*

— Olin Shivers

## ACKNOWLEDGMENTS

---

Northeastern University’s Programming Research Lab has collectively been an incredible asset. Over the years, I’ve seen the lab culture go through some good and bad times, but the essential parts stand strong. Careful criticism and refinement of each other’s research, writing, and speaking helps get our message out clearly. Rather than a traditional “journal club,” new students are brought up to speed in a student-run seminar with sessions focused on major topics in programming languages. Establishing broad shared background knowledge across the lab helps us understand each other’s work and the immense variety of research coming from other labs. Daily contact with others who work on seemingly unrelated projects is the only way I could have pulled together all the pieces on which to build my own. I’ve learned a lot from the Formal Methods Group as well. Though the group is smaller than the PRL, the latter half of my PhD drew heavily on techniques I learned from them.

There were also many earlier influences in life that led me to this field, this lab, and this dissertation. Over a three-semester span at Michigan Technological University, I got my first real exposure to compilation, static analysis, parallel programming, and the design behind programming languages. The faculty teaching these courses were eager not only to take us through a long tour of the grimy innards of such systems but also point at unanswered questions. Having a grand old time digging around here is how I found out for certain that these were the kinds of projects I like to work on.

Even earlier than that, my father told me when first teaching me to use BASIC on the family’s Apple IIe that *programming* is how to get access to the full power this amazing tool has to offer. Never-ending encouragement from my parents to keep following my curiosity (and occasionally to send back long-winded reports on what I’d found) has kept me going as long as I can remember. So I thank the whole family, especially my parents Jeff and Toni and my brother Alex, for their unwavering support. For a more down-to-earth note, I must also thank my parents for their generosity as frequent hosts during semester breaks and for logistical support through the two and a half cross-country moves involved in my PhD.



- Olin Shivers brings to the Remora project an intense enthusiasm for and long experience using rank-polymorphic programming,

which makes for a well-tuned intuition about what programming in a new rank-polymorphic language ought to be like. I don't think this work could have been what it is without being guided by Olin's taste in programming. Lots of people have asked me what Olin's technical role in Remora is because it looks so far removed from his past research, so there it is. That said, an advisor's role is more than just *technical* guidance. Academia is a strange place to an outsider, and it's not really a culture I was socialized for before I came to Northeastern. Olin has always been happy to explain the odd customs and unwritten rules, often before I knew to ask.

- Pete Manolios was originally the "formal methods consultant" for Remora. A type system is designed to encode invariants about arrays' shapes needs to be limited to invariants a type checker (or inferrer) can reason about effectively. Pete's expertise on decision procedures served both as a counterbalance for the temptations of expressiveness and as guidance once the goal of type inference pushed Remora's needs beyond the bounds of already established work. The argument supporting the type-erasure transformation also follows a form of reasoning I learned from Pete.
- Amal Ahmed is known as the lab's type systems expert. My familiarity with the design of type systems starts from the seminar she ran my first year at Northeastern. That extends also to reasoning about type systems, which turned out to be the bulk of my dissertation work. Future development of Remora still has a lot to learn from her work on integrating programming languages with differing type disciplines.
- Alex Aiken brings to the committee a working familiarity with the practical uses of parallelism and compilation issues related to it. Having him on the committee helped keep the design of Remora from straying too far into head-in-the-clouds esoterica. While he shares a lot of common understanding with Olin, he has insight we just can't get locally.



My grasp on formal semantics is due primarily to Matthias Felleisen's teaching, and he has done a lot to shape the lab culture that helps us students grow. I also thank him for insistently reminding everyone (inside and outside the PRL) that programming languages are for *programming* and for keeping the craft of software as the underlying motivation for research in programming languages. He has also kept me from falling through administrative cracks now and then.

Having Jan Vitek join the lab has helped me keep aware of contemporary language design work looking at the same kinds of problems as



Remora, and that's only one province of his intercontinental research empire. He is also heavily into dynamic compilation techniques. Although that sort of work has not had much influence on this project's direction so far, it looks like a good possibility for the future.

Will Clinger was my official mentor when I came to Northeastern. Language design is strongly shaped by history, and you can learn a lot of it from him. As a first-semester student—especially a first-month student—I didn't know what I wanted to spend the upcoming years doing. Will was happy to talk about a variety of possible directions, whether they were close to his background or not.



You can learn a lot in a surprisingly short time by standing around at a whiteboard with someone in the same field who has a radically different way of thinking about it. One of the most valuable features of the PRL has been a steady stream of labmates with what felt to me like completely foreign mindsets: Paul Stansifer and Michael Ballantyne, immersed in syntax and how to transform it; Leif Andersen, who is like them but more so and also much more ambitious about what programming could be; Tony Garnock-Jones, whose thoughts are shaped by prior focus on distributed systems; Andrew Cobb, who dove deep into automatic differentiation;<sup>1</sup> Max New, who has facility with a broad range of mathematical structures and fits tricky semantic issues neatly onto them.

My co-advisees Jonathan Schuster, Alex Marquez, and Dionna Glaze, were always around to commiserate about whatever went wrong.

I am grateful for the support, shop talk, and advice from the rest of the Programming Research Lab too: Mitch Wand, Heather Miller, Frank Tip, John Reppy,<sup>2</sup> Jason Hemann, Ben Lerner, Stephen Chang, Ben Greenman Sam Caldwell, Ben Chung, Oli Flückiger, Daniel Patterson, Ming-Ho Yee, Aviral Goel, Aaron Weiss, Artem Pelenitsyn, Julia Belyakova, Alexi Turcotte, Ellen Arteca, Olek Gierczak, Cameron Moy, Nate Yazdani, Ryan Culpepper, Sam Tobin-Hochstadt Carl Eastlund, Christos Dimoulas, Dimitris Vardoulakis, Jesse Tov, Tess Strickland, Aaron Turon, Jamie Perconti, Erik Silkensen, Philip Mates, Vincent St-Amour, Asumu Takikawa, Kevin Clancy, William Bowman, Celeste Hollenbeck, Di Zhong, Hyeyoung Shin, Sam Lazarus,<sup>3</sup> Eli Barzilay, David Van Horn, Filip Křikava, Gabriel Scherer, Konrad Siek, Paley Li, and Saba Alimadadi.

Thanks as well to the other faculty and students of the Formal Methods Group: Thomas Wahl, Stavros Tripakis, Ankit Kumar, Ben Boskin, Andrew Walter, Konstantinos Athanasiou, Harsh Chamarthi, Mitesh Jain, Vasilis Papavasileiou, Peizun Liu, and Jaideep Ramachandran.

My research has been helped a lot by consults with Dougal Maclaurin, Alexey Radul, Joshua Dunfield, Neel Krishnaswami, Mooly Sagiv,

<sup>1</sup> We described him as the only one of us who does any real work.

<sup>2</sup> Visiting on sabbatical

<sup>3</sup> Helped test out Remora's suitability as a tool for signal processing code

Jeremy Gibbons, Burke Fetscher, Gabriel Radanne, Ed Kmett, Pierre-Evariste Dagand, Didier Rémy, François Pottier, Vinod Grover, Sean Lee, and Mahesh Ravishankar.

Thanks to the faculty from Michigan Tech who helped set me along this path: Steve Carr, Zhenlin Wang, Steve Seidel, Soner Onder, and Don Kreher.

My motivation for this line of research also comes partly from shop talk with a few non-CS folks I knew at Tech: Greg Karlovits, Caleb Carlin, Katie Sebeck, and Chuanzhi Zang.

Finally, thanks to Carl West for always pushing me to think about the design and function of the tools I use.

## CONTENTS

---

1	INTRODUCTION	1
1.1	My Thesis	2
<b>I FORMAL SEMANTICS</b>		
2	BACKGROUND	7
2.1	Rank Polymorphism	7
2.2	Formalism for APL	11
2.3	Related Array-Oriented Languages	12
2.4	Dependent Types	19
3	PROGRAMMING WITH RANK POLYMORPHISM	23
3.1	Rank Polymorphism with Dynamic Typing	23
3.2	A Type Discipline for Rank Polymorphism	31
4	SEMANTICS OF TYPED REMORA	37
4.1	Syntax	37
4.2	Static Semantics	43
4.3	Dynamic Semantics	54
4.4	Type Soundness	58
<b>II TYPE INFERENCE</b>		
5	BACKGROUND	63
5.1	Local Type Inference and Bidirectional Typing	63
5.2	Dependent Type Inference	65
5.3	Theory of Sequences	67
6	LOCALLY INFERRING DEPENDENT TYPES	73
6.1	Syntax	75
6.2	Solver Invocation	79
6.3	Bidirectional Judgment Forms	80
6.4	Subtyping Judgment Forms	90
7	FIRST-ORDER THEORY OF ARRAY SHAPES	101
7.1	Structure of Solver Queries	102
7.2	String Equations Modulo Theories	103
7.3	Generalizing to a Mixed-Prefix Fragment	113
8	EVALUATION	115
8.1	Elaboration Soundness	115
8.2	Practical Use	123
<b>III TRANSLATION</b>		
9	BACKGROUND	145
9.1	Compilation Targets	145
9.2	Dataflow Graphs	147
10	TYPE ERASURE	149

10.1	Erased Remora	149
10.2	Correctness of Translation	152
11	EXPLICIT ITERATION	159
11.1	Mapping and Replication	159
11.2	Further Steps to a Low-Level Language	165
12	CONCLUSION	169
12.1	Future Directions	170

BIBLIOGRAPHY	173
--------------	-----

#### IV APPENDIX

A	PROOFS (4.2: STATIC SEMANTICS)	187
B	PROOFS (4.4: TYPE SOUNDNESS)	229
C	PROOFS (8.1: ELABORATION SOUNDNESS)	241
D	PROOFS (10.2: CORRECTNESS OF TRANSLATION)	259

## LIST OF FIGURES

---

Figure 4.1	Core Remora grammar	38
Figure 4.2	Value forms, syntactic contexts, and evaluation contexts	39
Figure 4.3	Common array-manipulation primitive operations and their Remora types. Each function type is wrapped in a scalar, with the function name bound at that scalar type in the base environment. For readability, we elide the enclosing $\Pi$ and $\forall$ forms.	42
Figure 4.4	Types for <code>iota</code> and its variants. More detailed argument-shape information allows a more precise result shape: <code>iota/v</code> always produces a vector, while <code>iota/s</code> and <code>iota/w</code> have their result shape specified by their input.	44
Figure 4.5	Sorting rules	45
Figure 4.6	Kinding rules	46
Figure 4.7	Typing rules (introduction forms)	48
Figure 4.8	Typing rules (elimination forms)	49
Figure 4.9	Type equivalence	51
Figure 4.10	List-processing metafunctions	55
Figure 4.11	Dynamic semantics for Remora	57
Figure 5.1	Overlap axiom, visualized: $w$ is the overlapping portion of $a$ and $d$ .	68
Figure 6.1	Grammar for implicitly typed Remora	77
Figure 6.2	Environment structure for implicitly typed Remora	78
Figure 6.3	Type synthesis judgment	83
Figure 6.4	Type checking judgment	86
Figure 6.5	Application synthesis judgment	89
Figure 6.6	Instantiating existential type variables as subtypes	93
Figure 6.7	Instantiating existential type variables as super-types	94
Figure 6.8	Subtype rules, part 1: simple type forms	97
Figure 6.9	Subtype rules, part 2: polymorphic type forms	98
Figure 6.10	Supplemental “reach-through” rules for instantiating existential variables	100
Figure 8.1	Base environment entries used for type synthesis and elaboration of sample code	125
Figure 10.1	Abstract syntax for type-erased Remora	150
Figure 10.2	Dynamic semantics for erased Remora	151
Figure 10.3	Type erasure for Remora	153

Figure 10.4	Type-erasing Remora evaluation contexts	154
Figure 11.1	Abstract syntax for an internal representation with explicit iteration	160
Figure 11.2	Dynamic semantics for explicit iteration forms	161
Figure 11.3	Dynamic semantics for optional forms	162
Figure 11.4	Converting explicit Remora's implicit iteration to explicit iteration	164
Figure 11.5	Converting erased Remora's implicit iteration to explicit iteration	166

## INTRODUCTION

---

The essence of the rank-polymorphic programming model is implicitly treating all operations as *aggregate* operations, usable on arrays with arbitrarily many dimensions. The model was first introduced by Iverson with the language APL [43] (short for “A Programming Language”). Over time, Iverson continued to develop this programming model, making it gradually more flexible, eventually leading to the creation of J [50] as a successor to APL. The boon APL offered programmers was a notation without loops or recursion: Programs would automatically follow a control-flow structure appropriate for the data being consumed. The nature of the implicit iteration structure could be modified using second-order operators, such as folding, scanning, or operating over a moving window. These second-order operators would directly reveal all loop-carried data dependences.

In this sense, other languages demanded that unnecessary work be put into both compilers and user programs. The programmer would be expected to write the program’s iteration structure explicitly; in many languages this entails describing a particular serial encoding of what is fundamentally parallelizable computation. The compiler must then perform intricate static analysis to see past the programmer’s overspecified iteration schedule.

The design of APL earned a Turing award for Iverson [44] as well as a mention in an earlier Turing lecture [4], praising it for showing the basis of a solution to the “von Neumann bottleneck.” However APL’s subsequent development proceeded largely in isolation from mainstream programming-language research. The APL family of languages painted itself into a corner with design decisions such as requiring functions to take only one or two arguments and making parsing dependent on values assigned at run time. As a result, APL compilers were forced to support only a subset of the language (such as Budd’s compiler [10]) or to operate on small sections of code, alternating between executing each line of the program and compiling the next one [49]. What we gain from the rank-polymorphic programming model’s natural friendliness to parallelism, we can easily lose by continually interrupting the program to return control to a line-at-a-time compiler. Limiting the compiler to operating over a narrow window of code can also eliminate opportunities for code transformations like fusion, forcing unnecessary materialization of large arrays.

The tragedy of rank-polymorphic programming does not end at foregone opportunities for performance. Despite the convenience of rank polymorphism for writing array-processing code—a common task in

many application domains—APL and its close descendants do not see widespread use. There is enough desire for implicitly aggregate computation to support user communities for systems such as NumPy [73] and MATLAB [63], which do not follow as principled or as flexible a rule for matching functions with aggregate arguments<sup>4</sup>. However, programmers are driven away from APL itself by features such as obtuse syntax, restrictions on function arity, poor support for naming things, and a limited universe of atomic data to populate the arrays [2]. Investigating rank polymorphism itself, separated from the idiosyncrasies of APL and J themselves, calls for developing a new language. A new language can serve as a base from which to launch new design experiments not directly compatible with past languages.

<sup>4</sup> For example, operations which already expect aggregate data—perhaps the programmer writes a function to compute the norm of a vector or the determinant of a matrix—do not always lift easily to consume even higher-dimensional arguments

## 1.1 MY THESIS

The implicit, data-driven control structure of higher-order rank-polymorphic programs can be identified statically by a type system suitable for the programming style common in rank-polymorphic code.

Rank polymorphism is not a new programming model, but since it evolved mostly in isolation from the programming-languages research community, formal semantics has not kept up with the development of the programming model. In promising a *static*, type-based analysis, this thesis implicitly incurs the obligation to formalize the *dynamic* behavior of rank polymorphism. This type system is not only for safety: it also describes the program’s implicit control structure, whose discovery used to have to be deferred until run time. We need to know that the type system’s description of the program’s control structure is accurate, and static semantics can only be proven sound with respect to some dynamic semantics. On the other hand, any type system which rules out programs which “go wrong” can be expected to also rule out some programs which do not. So just having a sound type system for a programming model is not enough. We need a type system which is not so restrictive as to prohibit the code programmers typically write when using that programming model. Some prior work *has* imposed too much restriction, making many common array-programming primitive operations not just unwieldy but impossible to use. This thesis promises to avoid that dead end.

What follows to support my thesis is a design document for Remora, a rank-polymorphic programming language. While APL has seemed “too dynamic” for good static compilation, due to deriving its control structure from computed data, Remora uses a type system which tracks array shapes in order to identify the implicit iteration space of each function call. In order for types to provide enough detail about array shapes, Remora uses a restricted form of dependent typing, in the style



of Dependent ML [102]. In Dependent ML, types are not parameterized over arbitrary program terms but over a much more restricted language. For Remora, our language of type indices consists of natural numbers, describing individual dimensions, and sequences of natural numbers, describing array shapes.

Past work on applying dependent types to computing with arrays has focused on ensuring the safety of accessing individual array elements [93, 103]. Bounds checking array indices is essential in a programming model where extracting a single element is the only elimination form for arrays, but the rank-polymorphic programming model generally eschews this operation. Instead, arrays are consumed whole, and function application itself serves as the elimination form for arrays. Remora’s use of dependent types instead aims to check that arrays have compatible shapes without having to consider whether a particular element index falls within legal bounds. Remora’s type system is flexible enough to express polymorphism over the cell shape, such as a determinant function that can operate on square matrix cells of any size. It can also handle functions whose output shape is not determined by input shape alone, such as reading a vector of unknown size from user input or generating an array of caller-specified shape.

Part I gives a more extended overview of the rank-polymorphic programming model, including a demonstration using an untyped variant of Remora in Chapter 3. Remora’s operational semantics and the type system which describes program control structure are presented in Chapter 4, including a proof of type soundness (*i.e.*, the control structure implied by the type system is the actual program behavior).

Remora’s types themselves are very detailed and therefore verbose. A function’s type includes descriptions of its input and output—both the elements those arrays contain and their shapes—as well as explicit quantification over those element types and portions of the associated shapes. For example, the type of the `filter` function, which takes a bitmask specifying which parts of an array to keep or drop, has the type

```
(Arr (∀ ((t Atom))
      (A (Π ((d Dim) (s Shape))
          (A (-> ((Arr Bool (Shp d))
                  (Arr t (++) (Shp d) s))))
          (A (Σ ((k Dim))
              (A t (++) (Shp k) s))))
          (Shp)))
      (Shp)))
(Shp))
```

This type quantifies over `t`, the type of data appearing inside the array; `d`, the length of the array’s leading axis; and `s`, the sequence of remaining

axes. The first argument is the bitmask, a boolean vector with length  $d$ . The second argument is the  $d \times s$  array of  $t$ . The result, described with the  $\Sigma$  type, is a  $k \times s$  array, where  $k$  is the number of items the bitmask says to retain. Since the actual value of  $k$  is dependent on run-time data, it is existentially quantified—the result array’s leading axis has unknown length.

Explicit type annotations in Remora code can easily make up more of the program text than term-level code. Making the language usable for human programmers calls for type inference, to help identify restrictions on the shape of a function’s arguments and fill in the details of how argument arrays fit that shape. The algebra of array shapes makes Remora’s type system incompatible with global type inference strategies that rely on automatic generalization and principal typing. Instead, Remora uses bidirectional typing, as described in Chapter 6. The task of inferring shapes requires a new constraint solver for string equations modulo theories. The theory of array shapes and the structure of the corresponding solver are laid out in Chapter 7. The efficacy of the combined system is shown in Chapter 8, by proving that the elaborated code has the appropriate type—and so, the appropriate control structure—as well as by demonstrating type synthesis on a collection of sample code.

Part III considers issues in translating Remora code to a lower-level target. While the formal semantics makes heavy use of run-time type information, this is more of a convenience for the formalism. Chapter 10 shows how the run-time type information can be pared down to only the portion directly relevant to program control flow, moving the description of the expected input shape from a closure, a dynamically created object, to its call site, a static part of the program. The complementary portion of translation, replacing the implicit iteration structure with explicit looping, is described in Chapter 11.

Part I

FORMAL SEMANTICS



## BACKGROUND

## 2.1 RANK POLYMORPHISM

In rank-polymorphic array programming languages, such as APL [43], J [50], and FISH [46], all functions automatically lift over large array arguments. The universe of data in these languages consists of regular (*i.e.*, hyper-rectangular) arrays. Such an array is fully described by its sequence of “atoms,” which are the base values contained in the array, and its “shape,” a sequence of natural numbers describing how the atoms are arranged. For example, the matrix  $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$  has shape  $[2, 3]$  and atoms  $[1, 2, 3, 4, 5, 6]$ . The “rank” of an array is the number of dimensions it has, or the length of its shape, such as 2 for matrices. “Rank polymorphism” is the property of accepting arguments of arbitrarily high rank. As a brief demonstration<sup>5</sup> of rank polymorphism, in a dynamically typed, Lisp-like dialect, we can add a vector and a matrix using the same operation as we use to add two scalars.

```
> (+ 1 2)
3
```

```
> (+ [10 20]
     [[1 2 3]
      [4 5 6]])
[[11 12 13]
 [24 25 26]]
```

Making a function compatible with high-rank arguments does not require special handling by that function. In rank-polymorphic languages, it is instead a part of the semantics of function application itself. User-defined functions are treated no differently than functions built into the language. This places APL and its descendants in contrast with systems such as MATLAB [63], which includes iteration in the definitions of most built-in functions, or NumPy [73], where an *ad hoc* mechanism like operator overloading for an array data structure is used to make certain primitive operators lift while user code is denied such privileges.

Early design work on APL only permitted lifting operations to either two arrays of identical shape or an array of any shape and a scalar. In the case of two array arguments, the operation maps over corresponding pairs of array elements, producing a result whose shape matches that of the arguments. The semantics for the mixed scalar/aggregate case can

<sup>5</sup> These examples are written as an imaginary session in a read-eval-print loop. For now, all examples will be in a dynamically typed variant of Remora. Static typing will be introduced later.

be seen as replicating the scalar argument before mapping the operation over corresponding pairs of array elements. Functional programmers might prefer to think of it as partially applying the function to the scalar argument, producing a liftable unary function to map over the aggregate argument.

```
> (+ [[90 80 70]
      [60 50 40]]
     [[1 2 3]
      [4 5 6]])
[[91 82 73]
 [64 55 46]]
```

```
> (+ 1 [[1 2 3]
        [4 5 6]])
[[2 3 4]
 [5 6 7]]
```

```
> (add1 [[1 2 3]
         [4 5 6]])
[[2 3 4]
 [5 6 7]]
```

In order to generalize the implicit lifting beyond functions on scalars, it was necessary to associate with each function the ranks it expects for its arguments. This allows, for example, a vector-norm function to lift over rank- $n$  data by viewing it as a rank- $(n - 1)$  collection of vectors. Each of these vectors within the larger array is called a “cell,” and the vector-norm function will compute an independent result for each cell in its argument. More generally, when a function expects a rank- $r$  argument, the actual argument’s cells are its rank- $r$  sub-arrays. The cell shape consists of the  $r$  rightmost entries in the array’s shape. The aggregate structure around the cells is called the “frame.” When rank- $n$  data is passed to a function which expects rank- $r$  input (with  $r \leq n$ ), the rank- $(n - r)$  frame serves as the iteration space for lifting the function over that data. The final result is the individual cells’ results assembled in that rank- $(n - r)$  frame.

This way, the lifting mechanism generalizes from requiring matching shapes to requiring matching *frames*. For example, if a polynomial-evaluation function expects a vector of coefficients and a scalar at which to evaluate the polynomial, it is also applicable to a matrix and a vector, provided their respective leading axes have the same length (*i.e.*, there are as many coefficient vectors as there are values).

```
> (poly-eval [-10 5 1] 3) ; -10 + 5x + x^2, at x=3
```

```
> (poly-eval [[-10 5 1] ; this polynomial at x=3
             [ 5 3 4]] ; this one at x=2
             [3 2])
[14 27]
```

Iverson generalized the frame-compatibility rules one step further by considering two frames compatible as long as one is a *prefix* of the other. Imagining the lifted function application as an implicit nest of `for` loops, the prefix-agreement rule says that both arguments must agree on the outermost loops, but one argument may demand additional inner loop layers. For example, adding a matrix and a vector corresponds to two nested loops: the outer loop traverses the matrix's major axis and the vector, and the inner loop traverses only the matrix's minor axis, keeping a constant position within the vector. Generalizing to prefix agreement enables the vector-matrix addition example we began with.

```
> (+ [10 20]
     [[1 2 3]
      [4 5 6]])
[[11 12 13]
 [24 25 26]]
```

Remora makes two additions to Iverson's prefix-agreement implicit lifting. First, functions of any arity are permitted, whereas APL and J only allow lifting for functions of one or two arguments. The rule that one argument frame must prefix the other is generalized to requiring that all frames be prefix-orderable. Then the frame which has all others as prefixes is the *principal* frame which determines the iteration space for that function application. We can write a ternary function (in this case, with  $\theta$  indicating 0-dimensional expected inputs) as follows:

```
> (define (lerp (lo  $\theta$ ) (hi  $\theta$ ) (alpha  $\theta$ ))
    (+ (* lo (- 1 alpha))
       (* hi alpha)))

> (lerp [1 1] [0 3] 0.75)
[0.25 2.5]
```

Second, Remora supports first-class functions. This means a function may produce a function as a result, and lifting the function-producing function will build an array whose atoms are functions. The function position in an application form can therefore contain an array of functions, which is considered to have scalar cells—its frame is its entire shape. The functions in the array must agree as to the cell rank of each argument so that a single frame-of-cells decomposition can apply to each one.

```

> ([+ -] 10 3)
[13 7]
> ((curry +) [3 4]) [[10 20 30]
                    [40 50 60]]
[[13 23 33]
 [44 54 64]]

```

Allowing arrays in function position—applying the old functional programming idea that behavior is itself data—means the fundamentally SIMD programming model is also able to express MIMD computation. However, the indirect jumping and diverging control paths which tend to arise from the use of closures (or virtual function calls) are often poorly supported on commodity parallel hardware. This “higher-order SIMD” case is likely to fit better with coarser-grained task parallelism than actual SIMD hardware.

Some applications demand data which is not strictly regular. The programmer might want to operate on a list of strings, where each string is represented as a vector of characters. Then a list of strings is a matrix of characters, with one row for each string. Requiring regularity would mean that all strings in the same list must have the same length. One solution used in prior languages is to designate a padding character—such as a null byte or a space—and extend every string to match the length of the longest. This character is known in Iverson’s terminology as the “fill” element, and APL and J have a designated fill for each type of atom. Since the fill element is still an ordinary character (or integer, float, *etc.*), the particular use case must not treat fill elements at the end as semantically significant (*e.g.*, a convention of ignoring trailing whitespace). Filling can also cause a quadratic blowup in storage costs if there is one extremely long string in the list.

Non-regular data can even arise from processing regular data, by lifting a function whose result shape depends on the particular atoms in its input array, not just the input shape. For example, `iota` takes as input a vector and produces an array whose shape is that argument vector, containing natural numbers counting up from 0 as its atoms.

```

> (iota [6])
[0 1 2 3 4 5]

> (iota [2 4 3])
[[[0 1 2]
 [3 4 5]
 [6 7 8]
 [9 10 11]]
 [[12 13 14]
 [15 16 17]
 [18 19 20]
 [21 22 23]]]

```



If `iota` is applied to a matrix, there will be several result cells with possibly different shapes. A regular array cannot have sibling subarrays of different shape. In certain cases, APL and J will insert fill elements, but a more general method of handling non-regular data is available.

A “box” is an atom containing a single array of any shape. Several boxes can be collected together in an array, although their contents may differ in shape or even rank. While APL and J each offer a function for extracting the data from a box, they recreate the danger of having a function which is unsafe to use in lifted application.<sup>6</sup> Instead, Remora uses a `let`-like form for temporarily binding the box’s contents to a variable, giving an opportunity to clean up any raggedness.

```
> (define b (box [4 5 6]))

> (unbox contents b
   (sum contents))
15
```

General tree data falls at the extreme end of irregularity. Encoding and manipulating such data is beyond the scope of this dissertation, but Hsu’s description of a compiler implemented in APL [42] includes a discussion of several ways to do so.

## 2.2 FORMALISM FOR APL

APL has received less attention in the way of formal semantics than  $\lambda$ -calculus, and much of the prior work has focused on the behavior of the primitive (first- and second-order) functions rather than on rank-polymorphic function application itself.

Two complementary lines of work cover a large portion of the problem. First, Gerhart set out to formally verify APL programs, which of course requires a formal specification of their behavior [32]. The proofs themselves deal with assertions about program state at particular points during execution, much like Hoare logic [40]. Gerhart’s inference method accounts for the constraints which functions place on their operands’ shapes: either one must be scalar, or the two must be equal.<sup>7</sup> Mullin complements Gerhart’s work by laying out the semantics of APL’s toolbox of array operations [85]. Rather than focusing on APL’s unique function application mechanics, Mullin’s formalism is designed to develop an algebra of array computation such as might be desired by an APL implementor seeking to optimize user programs.

The prospect of separating out a core language to isolate the essence of APL from surrounding incidental complexity was explored by Tu’s Functional Array Calculator (FAC) [94]. FAC explicitly defines high-dimensional arrays as deeply nested one-dimensional arrays, with a “partition” construct allowing an array to be broken up into pieces which

<sup>6</sup> From the viewpoint of type-based analysis, this way of consuming boxed data corresponds to strong dependent sums, whereas Remora—like *Dependent ML*—uses weak sums.

<sup>7</sup> This does not handle functions with non-scalar cells or the prefix agreement rule—Gerhart handled APL as it existed at the time.

can be operated on in parallel. A variety of APL primitives are defined in terms of FAC constructs. FAC comes with a sketch of the mathematical structures needed for a denotational semantics of an array language. However, it does not fully define the mapping from syntax to domain elements. Most critically, it lacks a definition of rank-polymorphic function application. Instead, the semantics for FAC’s application is written as simply invoking meta-level function application.<sup>8</sup>

<sup>8</sup> This is referred to using metafunctions named `MonadicMap` and `DyadicMap`, suggesting that map-like behavior is intended, but their definitions do not include it.

Orgass constructs a denotational semantics using only natural numbers as the domain of values [75]. Using a conventional encoding of tuples, an array is represented as a 5-tuple whose elements are rank, shape (itself a tuple of dimensions), number of atoms, type tag, and a tuple containing the atoms themselves. Several common array combinators are defined as primitive recursive functions on  $\mathbb{N}$ . Lifting over large aggregate input is included in the definitions of individual functions. An example defines how this lifting is performed for a function on scalars, but some machinery needed for modern-day rank polymorphism—such as higher-rank cells and prefix agreement—is missing, as this work predates those developments.

### 2.3 RELATED ARRAY-ORIENTED LANGUAGES

The rank-polymorphic programming model arose from Iverson’s work on APL [43] over the course of several decades. He eventually designed J [50] to be a successor to APL, fixing what he considered design mistakes. Iverson worked in isolation from mainstream programming language research, so language-design developments often now taken for granted, such as lexical scope and higher-arity functions, are not used in APL and J. Iverson also used his own vocabulary derived from linguistics. In J, arrays, first-order functions, and variables are referred to as “nouns,” “verbs,” and “pronouns” respectively. Second-order functions are “adverbs” if unary and “conjunctions” if binary. Thus, like in natural languages, an adverb modifies a verb, changing what it does when applied to one or two nouns. Adopting this terminology constrains how the user thinks about programming and even restrains further development of the design of the language. Second-order functions cannot be composed because composition itself is second-order (*i.e.*, it is only usable on first-order functions). These languages have developed a reputation as “write-only” languages due to Iverson’s unconventional selection of symbols for primitive operators and a popular attitude among the user community which disdains naming intermediate results of computation.

Several aspects of APL’s design—echoed as well in J—interfere with efforts to produce an effective compiler. Juxtaposition is overloaded to mean several things: array construction; two kinds of function composition; and function application, which may itself be prefix, infix, or postfix. Which of the above syntactic forms juxtaposition means depends

on the run-time values associated with juxtaposed symbols. The goal of statically parsing APL led in one case to the use of interprocedural data-flow analysis [101].

Although APL was initially intended as a mathematical notation, suitable for both algebraic manipulation and mechanized evaluation, many primitive functions include special-case behavior which makes it difficult to come up with robust rewrite rules. Operations which might appear amenable to reordering or fusion turn out not to be if one of them invokes the cell-padding mechanism.

There is also a sort of pole in the semantics around the results of many computations involving empty arrays. The result shape of a lifted operation consists of the principal frame shape followed by the shape of the result cells. If we lift an operation over a frame containing a 0-length dimension, there are no result cells. So what shape does the result array have? Resolving this in a way that reliably preserves straightforward algebraic reasoning requires knowing the full range of behavior of the function being lifted; APL and J instead estimate by observing the function's behavior on one example.

A more fundamental problem, which motivates this work, is heavy use of implicit control. When a hypothetical compiler encounters a function application, the iteration space is derived from the shapes of the argument arrays and the argument ranks the function expects. The conventional solution is to defer decisions about control flow until run time. An APL interpreter can simply inspect the actual functions<sup>9</sup> and arrays in order to recover the necessary information.

However, doing so at every function call can incur a high cost. Several second-order primitives are iteration-pattern manipulators, like reduction or stencil computation, and such functions repeatedly apply the first-order function they modify. While an ordinary first-order function application can amortize the rank-dispatch cost over a large argument array, invoking that function many times on smaller pieces of the array cannot. It is common for interpreters to identify some common applications of second-order functions as “special code” which is handled by an alternative implementation. Thus  $+/$  (folding with addition) can be implemented by a simple accumulating loop instead of going back and forth between the interpreter and the implementation of  $+$  itself. Unfortunately, recognition of special code is brittle and limited to built-in functions. Iteration involving user-defined functions cannot have a special subroutine in the interpreter.

Later work inspired by APL avoided some of the design decisions that interfere with compilation. Static typing provides some information about the arrays' shapes. *Repa* [52] is a Haskell library providing a regular array datatype with parallel aggregate operations. An array's type includes its shape, expressed as a type-level list of natural numbers. The inhabitants of a type implementing the *Shape* type class are indices

<sup>9</sup> Since a function's expected argument rank is an important part of its behavior, that information is generally included in its run-time representation.

into an array of that shape, such as integer sequences of the appropriate length. Whole-array operations such as `foldl` and `backpermute` have types which describe how their results’ shapes are determined from their arguments’ shapes. An extra `Slice` type class is needed to describe both extracting a sub-array and adding new axes via replication. Implementing a `Slice` instance describes how to convert between indexing into the entire array and indexing into a sub-array. A slice specifier thus identifies its target sub-array by describing the index transformation. In order to support a form of loop fusion within library code, `Repa` makes heavy use of laziness, representing a delayed array as a function from index to element. Later work exposed `Repa`’s data representation decisions so that the programmer could choose more easily when to force or delay aggregate operations [66].

There is no *implicit* lifting over an argument frame (only a `map` function for applying a Haskell function to every atom) and no notion of expected rank. Choosing the axis to use for an aggregate operation like reduction is accomplished by permuting the axes beforehand. The focus on lifting scalar operations also limits `filter`, which uses a predicate on array elements, to consuming vectors because having additional axes would mean leaving irregularly-placed holes in the resulting array, with no dimension sequence able to describe the output. By contrast, the analogous function in APL selects rank- $(r - 1)$  subarrays from a rank  $r$  array.

A more APL-faithful semantics is offered by Gibbons’ work with Naperian functors [33]. This builds on applicative functors [65], which can be thought of as structures which carry along some extra information about how function application ought to work. One common example is using the `List` type to encode nondeterminism—the potential for each computation step to lead to several different possible results. Using an operator called `<*>`, the `Applicative` type class’s extended version of function application, accumulates a growing collection of intermediate results (which may themselves be functions to apply with `<*>` later).

A similar `Applicative` instance can be built for rank-polymorphic application of arrays of functions, but Gibbons points out that the pattern can be generalized beyond any specific regular-array datatype. A Naperian functor is a structure in which element positions can be identified with inhabitants of some particular type. In other words, a functor is Naperian if there is a type which can be used to index into it.<sup>10</sup> Using some Haskell language extensions, it is possible to define a type-level function which produces a type corresponding to natural numbers up to a chosen bound  $n$  (itself given as a type-level natural number). This is the appropriate index type for an  $n$ -vector type, and the type of  $r$ -tuples containing bounded naturals indexes into rank- $r$  arrays.

Generalizing from nesting of vectors or lists to nesting of Naperian functors allows rank-polymorphic programming over data structures

<sup>10</sup> If we imagine an array as a function from positions to elements (i.e., the exponential type  $\text{ElT}^{\text{Pos}^n}$ ) then the logarithm of an array type is its index space. A functor with a logarithm is called “Naperian” in honor of John Napier, discoverer of logarithms.

with the same essence as arrays but substantially different run-time representation, such as homogeneous pairs (indexed by booleans) or even full binary trees (indexed by bit-vectors).

Imposing a sequential order on the index space, corresponding to the `Traversable` type class, enables whole-array operations such as reducing, scanning, and permuting along the dimension it represents.

FISh [46] ascribes a static shape to each array, also consisting of a sequence of natural numbers. It is checked by a shape-analysis pass, which is effectively a separate judgment from the type judgment. In FISh's shape judgment, the shape of a function is a function on shapes. This shape language includes nontrivial computation capability, so functions are not required to give specific argument shapes. Allowing shape-level computation means that a function's type does not universally quantify over pieces of its arguments' shapes (as in `Repa`) with instantiation to be worked out by unification or some other form of constraint solving. Instead, the result shape for a function application is discovered by running the corresponding shape function on the shapes already discovered for its arguments. Rank-polymorphic functions are thus shapeable, using shape functions which only manipulate the rightmost elements of their arguments. However, function application itself is not rank-polymorphic in FISh—instead, an explicit `map` with a scalar function is required.

The separation of shape into a separate computation language hit a dead end when dealing with functions like `filter` and `iota` because the result shape is not a well-defined function of the argument shape. Where code in a sufficiently extended Haskell might use universal quantification over a dimension or index type, FISh had no clear path forward. Requiring complete, static shape predictability proved to be too restrictive for practical use.

Another common array-programming model is centered on ragged nested vectors, giving up rank-polymorphic lifting in exchange for a more permissive data model. In NESL [7], a vector's type does not specify its length, but it does specify nesting depth. For example, `[int]` represents vectors of integers, while `[[int]]` represents vectors of vectors of integers. Vectors are typically consumed and produced by parallel comprehension, but a comprehension only iterates along the outermost nesting level. When consuming a nested vector, it is expected that the operation performed on each element—a vector with one less nesting level—may itself be a parallel computation. Since NESL is meant for working with ragged data, the inner sequences are expected to have differing lengths. Simply forking off a parallel sub-task for each one and collecting their results at the end would mean a lot of idle time for short sub-tasks while waiting for the longest to finish.

In order to simplify the cost model presented to the programmer, NESL's key contribution is a flattening technique which transforms a nested parallel computation on a nested vector into a flat parallel

computation on the flattened version of the vector [8]. Once a vector is in flattened form, it can be freely broken into equal-sized pieces to distribute across the available hardware resources. The flattened representation of a vector of vectors includes a “segment descriptor,” a list of the sub-vectors’ lengths. When inner pieces of a nested parallel computation have some nontrivial dependence structure, like in a scan or reduce, the segment descriptor identifies the boundaries where data dependence should not be carried forward.

Data Parallel Haskell imports the nested-parallelism programming model into Haskell. This required substantial work to integrate with the native Haskell data model, and using an efficient memory representation required a new type-system extension. For most datatypes, GHC, the predominant Haskell compiler, represents a value as a pointer to dynamically allocated memory<sup>11</sup> containing a thunk which will produce the actual value when executed. Traversing an array to operate on each element requires dereferencing many pointers and possibly forcing many thunks, losing the cache-friendliness of a contiguous array representation. For performance reasons, GHC therefore offers arrays containing the elements’ actual data rather than pointers to that data, but this is only available for certain built-in types. Data Parallel Haskell uses a non-parametric representation of array types in order to achieve the memory-access performance typically associated with arrays. The Haskell community’s traditional interest in loop fusion also provides performance benefits, eliminating not only intermediate data structures but also the synchronization that would be necessary to construct them.

<sup>11</sup> Outside of APL tradition, this is known as “boxed” data.

Whitney’s K programming language [56] was inspired by his work with Iverson on APL, but it uses the nested-vector programming model rather than rank polymorphism. While implicit and explicit mapping over nested data are available, depending on the operation being mapped, K is typically used for interacting with the kdb+ database. A language extension called Q offers conventional relational operators and queries over kdb+.

The close match between array-oriented programming models and GPU hardware, which uses widely vectorized functional units with high memory bandwidth for graphics computation, has sparked several projects developing high-level languages for general-purpose GPU computing. Unlike using SIMD instructions on a CPU, GPUs conventionally have several hardware threads grouped and scheduled together. Conditional branching is discouraged because it forces the group to temporarily split—only one subgroup can make progress at a time until they reach a control join point. Conveniently, array-oriented languages favor independent operations on many array elements at once with only rare control divergence.

Harlan [41] targets heterogeneous parallel hardware by allowing the programmer to specify where particular computation will take place and

where data will be stored. Data-flow analysis allows the compiler to elide unnecessary data movement. The distinction between procedures running on the CPU versus on the GPU reflects a separation required in lower-level GPU programming languages such as CUDA [70]. Despite this distinction, Harlan sets a design goal to avoid restricting the computation model available when writing GPU kernels. Preserving the simplicity of the programming model motivates much of the research related to Harlan, including extracting coarse task parallelism from finer-grained data-parallel code [90] and taking advantage of recent hardware advances which permit nested parallelism [104].

In Nova, *all* user-defined functions are meant to be used as parallel compute kernels, invoked from a second host language [13]. There is no syntax for array data in Nova because they are meant to be passed in as arguments from host code or generated from certain built-in functions. The programming model is otherwise fairly conventional functional style, with  $\lambda$  and a fixed-point form  $\mu$ , as well as a combinator library for common array operations such as `map` and `scan`. Arrays inhabit a nestable vector type. The type system thus tracks the rank of each array, but not the full shape. Instead, concrete array dimensions are kept alongside run-time data. Nova is meant to be usable as a compilation target for domain-specific languages or for embedding in standalone applications.

Futhark shares some of Nova’s design goals—a typed, high-level GPU language suitable as a compilation target—but it uses a more detailed type system which describes the full shapes of regular arrays [38]. Futhark is less flexible about common functional-language facilities, like first-class functions and recursion. It is effectively a first-order language. Certain second-order functions are provided, but they are built-in syntax with special treatment by the compiler. Programmers are not free to implement their own second-order functions. For the sake of practicality, Futhark’s type system must allow polymorphism over array dimensions, but Futhark’s compiler aggressively monomorphizes not only polymorphic function calls but also module instantiations [27].

Accelerate is a Haskell-embedded domain-specific language for GPU computation [11]. It uses a type encoding similar to Repa, with an array’s type encoding its rank. Since Accelerate is built as an embedded DSL, with Haskell serving as a metalanguage for building GPU kernels, Accelerate has the opportunity to perform substantial code transformation such as including fusing array operations without requiring special support from GHC [57]. The inevitable separation of GPU-resident data and computation from ordinary Haskell data living in CPU memory makes Accelerate somewhat less fluid to use than Repa.

While array-oriented programming is already a somewhat domain-specific programming model to begin with, some languages have targeted more specific application domains. In machine learning, TensorFlow [1] and PyTorch [77] provide multidimensional array structures similar to

those of NumPy, but they provide a collection of automatically differentiable operations, which can be executed on either the CPU or GPU. In a style similar to Haskell embedded DSLs, Python serves as the meta-language for constructing array programs. A data-flow graph model of array programs facilitates automatic differentiation.

Halide was developed to make performance tuning easier by separating the iteration schedule from the algorithmic core. Rather than general multidimensional array computation, it focuses specifically on manipulating image data. Common tasks such as stencil computation can be stated easily enough in a mainstream C-like language (or in a rank-polymorphic language), but maximizing performance depends on the order in which intermediate scalar values are calculated. For example, two rounds traversing the same image can make poor use of the data cache. Where a high-level language user might look to the compiler to perform fusion, Halide allows the programmer to specify when each iteration of each pass should happen, thus covering transformations such as loop interchange, fusion, and tiling, as well as rematerialization and the use of temporary storage. The schedule language is itself a DSL whose notation is based on commonplace iteration decisions about when to feed one pass's results to the next, rather than explicit `for` loops.

Diderot is a DSL for processing medical and scientific images using the vocabulary of tensor calculus, which operates on a continuous domain. Diderot code is explicit about its use of parallelism, with a thread-like construct separating out responsibility for regions of input and output in the tensor-field computation. Domain specificity enables Diderot to use only a small collection of types without polymorphism, thus avoiding the monomorphization effort of Futhark, and to provide an extensive toolkit of built-in tensor functions.

Recognizing the fundamentally array-like nature of relational databases, Chen *et al* developed HorseIR as an intermediate representation for relational queries, allowing optimizations commonly used in array-oriented languages to be used in a database engine. Although it is a 3-address notation, as is typical of low-level IRs, all operations are vector operations. HorseIR is monomorphic, but it does include a “wild card,” or dynamically checked, type for rare cases in which a table column has a statically uncertain type.

Some systems take a more conservative approach by injecting parallelizable array computation into what is typically thought of as a mainstream sequential programming model. These include language extensions such as OpenMP and Coarray Fortran, as well as some independently standing languages. In SaC, static single assignment form is enforced at source level—despite a superficially C-like programming model, mutation is prohibited [89]. Array computation is written using parallelizable comprehension forms corresponding to conventional operations (such as `map`, `fold`, *etc.*). Since array data can only be con-



sumed by extracting an individual element, the body of a comprehension necessarily computes a location in an array from which to select an element. The reliance on array indexing motivated Qube [93], an extension which uses Xi-style singleton and range types to check that an index fits within an array’s bounds [103]. The single-assignment approach was also used in SISAL [67]. While much of SISAL’s design focuses on stream processing—intrinsically serial computation since data must be accessed sequentially—the compiler finds opportunities for parallelism in pipelining stream operations. SISAL also includes parallelizable loops, though some of the analysis of dependence between iterations may be left to a run-time scheduler [29].

Of the prior languages discussed here, only APL and J offer general rank polymorphism, with argument frame shapes chosen based on a function’s expected argument ranks and the application automatically lifted over the principal frame. K allows implicit iteration for scalar functions applied to arguments of identical structure, but functions which consume aggregate input or are applied to arguments of differing structure require explicit use of a map-like operator. NumPy’s operator overloading allows certain primitives to be used on NumPy arrays with compatible shapes.<sup>12</sup> MATLAB’s inclusion of iteration in built-in functions’ definitions allows programmers to mostly avoid writing loops themselves. Otherwise, array-oriented programming languages have required the programmer to be explicit about using iteration, whether through a function like `map` or comprehension-like syntax.

<sup>12</sup> NumPy allows shapes to be brought into agreement by extending any unit dimensions and prepending new dimensions.

## 2.4 DEPENDENT TYPES

Remora’s ability to describe array shapes in detail comes from integrating rank polymorphism into a dependent type system. In the same way that parametric polymorphism in System F generalizes simple typing by allowing terms to be parameterized over types, and type constructors in the style of System  $F_\omega$  allow types to be parameterized over other types, dependent typing allows types to be parameterized over terms [5]. A classic example is a `List` type which is parameterized over the type of list elements and the length of the list. Functions which have restrictions as to the length of list they can accept can encode such restrictions in their input type. For example, `head` and `tail` only work on nonempty lists, so they would demand input of type `List T (n+1)`. The type of a list-producing function’s output must also include the length of the list, so we have a `tail` producing a `List T n`.

In order to support such input and output types, a dependent function type must quantify over the type  $T$  and the natural number  $n$ . Fully written out with explicit quantification, the type of `tail` is

$$\prod T:\text{Type}, n:\text{Nat} . \text{List } T (n+1) \rightarrow \text{List } T n$$

We use  $\Pi$  for the generalization of functions and universal quantification, but it is commonly known as a “dependent product.” A type of the form  $\Pi n:\text{Nat} . F[n]$  can be seen as a product type, much like a tuple, except that instead of a finite number of elements, this product has one for every natural number. It is effectively

$$F[0] \times F[1] \times F[2] \times \dots$$

Element projection from this dependent product works more like function application than selecting a statically named or numbered field.

The analogous existential quantifier is  $\Sigma$ . The type  $\Sigma n:\text{Nat} . F[n]$  carries both a hidden natural number  $n$  and a datum whose type depends on  $n$ . Since the existential variable  $n$  is a term variable, this type acts somewhat like a tuple, in contrast to a System F-style  $\exists$  type. Similar to  $\Pi$  types,  $\Sigma$  types are called “dependent sums” because they generalize sum types to indexing over an arbitrary type. The example  $\Sigma n:\text{Nat} . F[n]$  encodes the infinite sum

$$F[0] + F[1] + F[2] + \dots$$

There are two flavors of case analysis on dependent sums. One possibility, referred to as “strong” dependent sums, is to use projection operators, similar to those used for tuple types. Applying the projection  $\pi_1$  extracts the value of the existential variable, and  $\pi_2$  extracts the value whose type depends on it. Alternatively, “weak” dependent sums limit access to the existentially quantified value by only offering a *let*-like form for destructing. This is more similar to the conventional way of destructing  $\exists$  types. The two values are temporarily bound to variables, so the first variable can be used to type operations involving the second, but the *let* body’s type must not mention those variables.

Several dependent type theories have been developed, most notably Martin-Löf type theory [62] and the Calculus of Constructions [14]. They have formed the basis of several programming languages, including Agda [72], Coq [91], Epigram [64], and Idris [9]. Development of such languages and related infrastructure often focuses on tools for programming-language metatheory, using dependent types to state theorems about a language, with inhabitants of those types serving as proofs.

Even outside of metatheory, dependent type systems allow very detailed invariants about a function to be encoded in its type, for example requiring that two numbers be relatively prime or that a search tree be balanced. In Remora’s case, the purpose of the type system is not proving arbitrary invariants, but establishing enough information about array shapes to identify the implicit control structure of rank polymorphism. This is a smaller job, which calls for a smaller tool. The task of producing proofs of required properties as some of a function’s arguments is recognized as a barrier to adoption of dependent typing, motivating the design of systems which use specific decision procedures to check whether these type-like invariants are maintained.

One strategy is to restrict dependent types so that types are only parameterized over a restricted language with a decidable theory instead of depending on arbitrary program terms. This restricted dependent typing is the basis of Dependent ML [102]. The language of type indices is completely separate from the term-level and type-level languages:  $\Pi$  and  $\Sigma$  only bind index variables, and a separate  $\forall$  quantifier is used for type variables. Some computation can be done in the type-index language—which is necessary for expressing interesting properties of the data being typed—but computation is limited for the sake of decidability. Dependent ML is often presented with an index language based on Presburger arithmetic, allowing types like that of `append`:

$$\forall T. \Pi m, n: \text{Nat}. \text{List } T \ m \rightarrow \text{List } T \ n \rightarrow \text{List } T \ (m+n)$$

Presburger arithmetic is unable to express multiplication, so it is impossible to characterize the behavior of `flatten`, which would turn a `List (List T m) n` into a list whose length is  $m * n$ . Instead, `flatten` would have to return a list of indeterminate length, represented by the type

$$\Sigma l: \text{Nat}. \text{List } T \ l$$

An alternative strategy is to decorate ordinary non-dependent types with refinements expressed in a carefully chosen logic. In the case of function types, the refinements on the input and output types correspond to pre- and post-conditions. This refinement-based approach is known as “logically qualified datatypes,” or “liquid types” [86].

Restricted dependent types and liquid types are essentially equivalent in terms of the invariants they can express, with index arguments in the former corresponding to computationally irrelevant proof witnesses for logical qualifiers in the latter. Remora uses restricted dependent types because the goal of Remora’s type system is to identify the implicit shape-driven control structure. Merely ensuring shape compatibility is not enough—finding the iteration space requires access to the shape witnesses themselves.



Adopting an unfamiliar programming model requires developing a new mindset and intuition, for both how programs behave and how to make use of the model’s particular mechanics. Practical use of rank polymorphism often relies on manipulating a function’s argument ranks or generating a data structure which serves to represent an iteration space.

This chapter presents a tutorial on programming with rank polymorphism, starting in an untyped variant of Remora. This variant is implemented as an embedded language within Racket [30], invoked by identifying the language as `#lang remora/dynamic`. The language itself is available as a Racket package, with source and installation instructions at <https://github.com/jrslepak/Remora/tree/master/remora>.

Some syntactic sugar is included for making arrays easier to write out. Enclosing a sequence of expressions in brackets stands for a vector frame built from those expressions, which may themselves be bracketed sequences. As another syntactic convenience, an atom appearing in a syntactic position which requires an expression is automatically converted to a scalar. So the numeral 3 appearing in an expression position is replaced by `(array () 3)`. A vector can be written out as `[2 4 6 8]`, a matrix as `[[2 4] [6 8]]`, and so on.

### 3.1 RANK POLYMORPHISM WITH DYNAMIC TYPING

The syntax of Remora distinguishes expressions, whose eventual values must be arrays, from syntactic atoms, which stand for the basic values which populate an array. The numeric literal `20` is an atom, with no associated shape information. The array literal `(array () 20)` is an expression, denoting a scalar array whose sole atom is `20`. Its shape is the empty sequence because a scalar has rank 0. The expression

```
(array (2 3) 1 2 3 4 5 6)
```

is also an array—a  $2 \times 3$  matrix containing as atoms the numbers 1 through 6.

Although functions only operate on expressions, an individual function such as `(λ ((x 0)) x)` is an atom. However, the function position in an application form must be occupied by an expression. So applying that function requires it to at least be wrapped as a scalar (or larger array). A boxed array—the escape hatch from regularity—is also an atom, so it too must be included in an array in order to compute with it.

A frame expression form allows arrays to be assembled from other arrays, rather than directly from atoms. This means we can use an

expression which will compute the array we want rather than the array value itself. The shape described in a frame is only the leading dimensions which describe how the sub-array cells are to be laid out. The  $2 \times 3$  matrix above could be written as follows:

```
(frame (2)
  (array (3) 1 2 3)
  (array (3) 4 5 6))
```

This is a 2-vector frame built around 3-vector cells, which is the same as a  $2 \times 3$  matrix. Writing out constant values with frame instead of array does not buy us anything. The purpose of the frame form is to build arrays from *expressions* which can compute new arrays. The same  $2 \times 3$  matrix could be the result of evaluating this expression:

```
(frame (2)
  (- 10 [ 9 8 7])
  (sqrt [16 25 36]))
```

Regularity demands that every cell used to build a frame have the same shape. There is no matrix whose rows are  $[1\ 2\ 3]$  and  $[4\ 5]$  because these two vectors have different length. So they cannot coexist as cells in the same frame form. In a dynamically typed setting, combining them like this raises a run-time error:

```
(frame (2)
  (array (3) 1 2 3)
  (array (2) 4 5))
```

When static typing is introduced later, such cases can be ruled out ahead of time.

The handful of examples from the opening of Chapter 2 demonstrate several cases of implicitly lifting a function to consume arguments of various ranks. We now examine the behavior of one of them.

```
> (+ [10 20]
     [[1 2 3]
      [4 5 6]])
[[11 12 13]
 [24 25 26]]
```

There are two intuitive ways to understand the lifting behavior. One option is write out the iteration space as an explicit part of the program's control structure by translating to C-like for loops.

```
for (i = 0; i < 2; i++)
  for (j = 0; j < 3; j++)
    result[i][j] = vec[i] + mat[i][j];
```

Another option is a more evaluation-oriented intuition, keeping the iteration space represented in the data. In our vector-matrix addition example, the true iteration space is the  $[2, 3]$  frame arising from the matrix argument's shape, whereas the vector's shape is just the singleton sequence  $[2]$ . To make the iteration space more apparent, we transform the vector by replicating each of its cells—in this case, 10 and 20.

```
(+ [10 20]
   [[1 2 3]
    [4 5 6]])
  ↓
(+ [[10 10 10]
   [20 20 20]]
   [[1 2 3]
    [4 5 6]])
  ↓
[[(+ 10 1) (+ 10 2) (+ 10 3)]
 [(+ 20 4) (+ 20 5) (+ 20 6)]]
  ↓
[[11 12 13]
 [24 25 26]]
```

The evaluation intuition generalizes more easily to situations where the function being applied includes its own internal looping behavior, which is a very common situation in array-oriented programming.

A function is written out using a Scheme-like  $\lambda$ , but with some extra information attached to each formal parameter. Application lifts functions over the arguments' cells, and that behavior depends on how big those cells are. So the behavior of a function is not fully specified unless we also state the cell rank for each argument. For example, the following mean function has the rank of its formal parameter  $v$  specified as 1, meaning that it operates on vector cells.

```
(λ ((v 1))
  (/ (reduce + 0 v)
     (length v)))
```

In `#lang remora/dynamic`, we can turn this into a declaration for later use:

```
> (define (vec-mean (v 1))
  (/ (reduce + 0 v)
     (length v)))
> (vec-mean [4 8 0])
3
> (vec-mean [[6 3 6]
             [4 8 0]])
[5 4]
```

Although vector mean serves well as an example of how rank-polymorphic programs behave, the above version is less generally applicable than it could be. Remembering the row-vector versus column-vector question about addition, we might want the means of a matrix's columns. Several built-in functions, including `reduce` and `length`, treat the entire argument as a single cell. No matter how high the argument's rank, `length` returns the size of its major axis—the number of atoms in a vector, rows in a matrix, planes in a 3D array, *etc.*

```
> (length [1 2 3 4])
4
> (length [[1 2 3 4]
           [5 6 7 8]])
2
```

User-written code also has access to this capability by specifying a parameter's rank as `all`. The cell rank we use when applying this `sum` function is whatever rank the actual argument happens to have:

```
> (define (sum (v all))
    (reduce + 0 v))
> (sum [[1 2 3 4]
        [5 6 7 8]])
[6 8 10 12]
```

The major-axis mean function is written much like the vector-mean function, but it lifts differently.

```
> (define (mean (v all))
    (/ (reduce + 0 v)
       (length v)))
> (mean [4 8 0])
3
> (mean [[6 3 6]
         [4 8 0]])
[5 11/2 3]
```

Working with rank-polymorphic code favors a geometric intuition about the data, choosing which axis to use for some operation or how we want some arguments' axes aligned. Carrying out that choice means choosing a function with the right input-cell ranks, so Remora makes it easy to specify them. In the earlier example of vector-matrix addition, the vector is effectively treated as a column, but some cases, such as row operations used in Gaussian elimination, call for a vector to be treated as a row. All this requires is a version of the `+` function which expects rank-1 arguments. Such a function can be written by “reranking” `+`, that is,  $\eta$ -expanding it to take arguments of a different rank:



```
(λ ((a 1) (b 1)) (+ a b))
```

This is a common enough pattern to be worth some syntactic sugar. We will write a reranked function by preceding the original function with a tilde and a list of new argument ranks:  $\sim(1\ 1)+$  is the vector-addition function. When we apply this function to a vector and a matrix, their respective frames are scalar and vector.

```
(~(1 1)+
 [10 20 30] ; scalar frame, one vector cell
 [[1 2 3]   ; vector frame of vector cells
 [4 5 6]])
```

The explicit-control version of this code is a single `for` loop around a call to the function which implements the reranked `+`. That function itself is also a single loop traversing its two vector arguments. Inlining that call produces something quite similar to the previous vector-matrix addition, but with the *inner* loop traversing the vector argument.<sup>13</sup>

```
for (int i = 0; i < 2; i++)
  for (int j = 0; j < 3; j++)
    result[i][j] = vec[j] + mat[i][j];
```

Following the evaluation-based intuition, the 3-vector argument expands to a vector of vector cells, each of which is identical to the original vector. In the third step,  $\beta$  reduction reveals the lifted use of `+`, corresponding to the inlined inner loop above.

```
(~(1 1)+ [10 20 30]
 [[1 2 3]
 [4 5 6]])
↓
(~(1 1)+ [[10 20 30]
 [10 20 30]]
 [[1 2 3]
 [4 5 6]])
↓
[(~(1 1)+ [10 20 30] [1 2 3])
 (~(1 1)+ [10 20 30] [4 5 6])]
↓
[(+ [10 20 30] [1 2 3])
 (+ [10 20 30] [4 5 6])]
↓
[[(+ 10 1) (+ 20 2) (+ 30 3)]
 [(+ 10 4) (+ 20 5) (+ 30 6)]]
↓
[[11 22 33]
 [14 25 36]]
```

<sup>13</sup> This is a single-character difference from the previous version, and inlining a mutation-heavy function can be a substantial leap of logic, so evaluation is favored over translation for describing the intuition behind array-oriented programming.

Reranking is important for effectively using common whole-array operations, such as `append`, `reduce`, and `rotate`. These functions operate along the major axis, treating the entire array as a single cell. No matter the argument's shape, it will be considered to have a scalar frame.

```
> (append [[1 2]
           [3 4]]
      [[5 6]
       [7 8]])
[[1 2]
 [3 4]
 [5 6]
 [7 8]]

> (reduce + 0 [[1 2]
               [3 4]])
[4 6]

> (rotate [[1 2 3]
           [4 5 6]
           [7 8 9]]
         1)
[[4 5 6]
 [7 8 9]
 [1 2 3]]
```

Suppose we instead want to stitch matrices together horizontally or find the sum of a matrix along its minor axis or rotate the rows instead of the columns. The minor-axis summation might be rephrased as the “sum of each vector,” indicating that we want a vector-consuming version of `reduce`. We write that function as `~(0 0 1)reduce`. Similarly, appending along the minor axis is the same as appending corresponding rows and then gathering them back together as a matrix, and rotating along the minor axis is the same as rotating each row individually:

```
> (~(1 1)append [[1 2]
                 [3 4]]
      [[5 6]
       [7 8]])
[[1 2 5 6]
 [3 4 7 8]]

> (~(0 0 1)reduce + 0 [[1 2]
                       [3 4]])
[3 7]
```

```
> (~(1 0)rotate [[1 2 3]
                 [4 5 6]
                 [7 8 9]]
      1)
[[2 3 1]
 [5 6 4]
 [8 9 7]]
```

Splitting an array into cells means those cells' major axes are some later axis of the larger array. So functions initially written to operate along an argument's major axis are easy to adapt to use other axes by reranking. This also applies to the mean function we wrote earlier.

```
> (~(1)mean [[6 3 6]
             [4 8 0]])
[5 4]
```

Boxed data relaxes the regularity requirements by allowing an array of any arbitrary shape to be wrapped up as a single atom. Several boxes can be packed together into an array, although their contents may differ in shape or even rank. The shape of boxes' contents has no bearing on the shape of the array containing the boxes.

```
> (shape-of [(box [4 5 6])
            (box [[1 2]
                [3 4]])])
[2]
```

While common array-computation primitives like `iota` are available in #lang remora/dynamic, they are not recommended for lifting over an argument frame:

```
> (iota [3])
[0 1 2]
> (iota [4])
[0 1 2 3]
> (iota [[3] [4]])
Error: Result cells have mismatched shapes
```

A lifting-safe variant `iota*` instead produces boxed output, ensuring that the result cell is always scalar.

```
> (iota* [3])
(box [0 1 2])

> (iota* [4])
(box [0 1 2 3])
```

```
> (iota* [[3] [4]])
[(box [0 1 2])
 (box [0 1 2 3])]
```

Boxed data is consumed using a special `unbox` form, which acts much like `let`, temporarily binding the box's contents. With `unbox`, the body gives the programmer an opportunity to account for the raggedness remaining in the result data from some lifted operation. Boxing enforces a separation between the regular outer axes, where implicit lifting is available, and potentially non-regular inner axes, which require explicit handling. When taking the sum of several boxed vectors, we know we get scalar result cells, so there is no need to `box` them.

```
> (define (boxvec-sum (b 0))
  (unbox vec b
    (sum vec)))

> (boxvec-sum [(box [ 5 6 7 8])
              (box [12 13 14])])
[26 39]
```

On the other hand, applying `add1` to each box's contents would preserve their various shapes, so each box's result needs to be boxed itself.

```
> (define (box-add1 (b 0))
  (unbox contents b
    (add1 contents)))

> (box-add1 (box [1 2 3]))
[2 3 4]

> (box-add1 [(box [1 2 3])
            (box [7 8])])
Error: Result cells have mismatched shapes

> (define (box-add1* (b 0))
  (unbox contents b
    (box (add1 contents))))

> (box-add1* (box [1 2 3]))
(box [2 3 4])

> (box-add1* [(box [1 2 3])
            (box [7 8])])
[(box [2 3 4])
 (box [8 9])]
```

Higher-order programming makes it easy to modify existing functions to operate within a box or extract data from a box for regularization.

```
> (define (from-box (f 0))
  (λ ((b 0)) (unbox contents b (f contents))))

> ((from-box sum) (iota* [[3] [4]]))
[3 6]

> (define (in-box (f 0))
  (λ ((b 0))
    (box (unbox contents b (f contents)))))

> ((in-box double) (iota* [[3] [4]]))
[(box [0 2 4])
 (box [0 2 4 6])]
```

So the list of varying-length strings can be represented with a ragged array, with each individual string wrapped up as a boxed vector. An operation which consumes an entire string is still liftable, and a function like `in-box` can adapt it to handle boxed strings.

One awkward hole remains in `#lang remora/dynamic`. When function application lifts over an empty frame—*i.e.*, the frame has a zero dimension—what is the shape of the resulting array? The frame portion of the shape is clear, but the cell portion is indeterminate. Ideally, the result cell shape should be whatever the function would have produced as the cell shape had there been any cells. For example, `sqrt` would give a scalar cell shape, an `RGBα` pixel compositing function would give a 4-vector result shape, and `reverse` would give a result cell shape matching the input cell shape. While `#lang remora/dynamic` offers an alternative form of application with a user-specified result shape to use if the frame is empty, a more robust solution is to use static information about the function’s behavior.

### 3.2 A TYPE DISCIPLINE FOR RANK POLYMORPHISM

Although the original motivation for Remora’s type system was static understanding of rank polymorphism’s data-driven control flow, having control structure tied so closely to data means that ensuring sensible control flow requires ensuring valid data. Function application without a valid frame, whether due to ill-structured data or mutually incompatible arguments, cannot produce a valid result.

Checking whether the function and argument arrays have compatible shapes for frame lifting requires tracking their shapes within their types.<sup>14</sup> Rank-polymorphic lifting itself is common to all functions, so

<sup>14</sup> Tracking only rank can solve the problem of trying to use a matrix-inverse function on a vector but not the problem of trying to use it on a non-square matrix. The problem of trying to use it on a non-singular matrix is beyond the scope of this work.

it does not need to be described in each function's type. Instead, a function's type need only describe its behavior on an individual cell, and the type of its result when lifted over some frame can be derived using the frame and result-cell shapes. For example, a simple addition function can itself be typed as consuming two numeric scalars and producing one numeric scalar. The type resulting from applying it to a 3-vector and a  $3 \times 4$  matrix is a  $3 \times 4$  matrix, whereas applying it to a 3-vector and a 4-vector is ill-typed.

While we are now stepping outside of what can be done in #lang remora/dynamic, there should ideally be little difference in the code we write. Arrays of base values are quite easy to type: `[2 4 6 8]` is clearly a 4-vector containing integers, which we'll write as `[Int 4]` and `[#t #t #f]` is a 3-vector of booleans, or `[Bool 3]`. For scalar data, we could write `[Float]`, with no dimensions listed, but when the context clearly requires an array type, for brevity we can just write the atom type itself: `Float`. For higher-rank data, we can list additional axes, such as typing this  $30^\circ$  rotation matrix

```
[[0.866 -0.500]
 [0.500  0.866]]
```

as `[Float 2 2]`. This is shorthand for the more explicit `(A Float (shape 2 2))`—an array of floating-point numbers with the shape  $2 \times 2$ . The type constructor `Arr` builds an array from an atom type, but it is indexed by a more term-like shape expression. In a context where a type describes an array as opposed to a bare atom, an unadorned atom type like `Int` is taken to mean an array type with the empty shape, `(A Int (shape))`.

With an array type indexed by shape, we now need a language for type indices. That language is split into two sorts: dimensions and shapes. A dimension describes the length of one axis of an array, and a shape is a sequence of dimensions. Any natural number is a valid dimension. We also allow addition on dimensions,<sup>15</sup> written as `(+ d1 d2 ...)`. This way, we can support operations like appending new elements onto a vector. Shapes can also be appended, capturing the nesting of cells in a frame. The shape of a  $3 \times 7$  frame of  $2 \times 4$  cells is constructed by appending the frame shape and cell shape: `(++ (shape 3 7) (shape 2 4))` is equal to `(shape 3 7 2 4)`.

The shorthand for array types can be thought of as using an atom type followed by a sequence of shapes to append, with each dimension `d` being implicitly promoted to the vector shape `(shape d)`. Using shape variable `@s` and dimension variable `$d`,<sup>16</sup>

```
[Int (+ $d 1) @s 3]
```

is shorthand for

```
(A Int (++ (shape (+ $d 1)) @s (shape 3)))
```

<sup>15</sup> For decidability reasons discussed later, we do not allow multiplication.

<sup>16</sup> This desugaring relies on knowing the difference between shape variables and dimension variables, so we use different sigils to distinguish them.

When writing out functions, we will include a bit more detail. Instead of stating the cell rank for each formal parameter, we state the entire cell type. For linear interpolation on floats, we'll write:

```
(λ ((lo Float) (hi Float) (a Float))
  (+ (* a hi) (* (- 1 a) lo)))
```

Adapting the `sum` function from the previous section to work on vectors in typed code introduces a new problem. Suppose we write something like

```
(λ ((v [Int 3]))
  (reduce + 0 v))
```

We can only use this function on vectors of length 3. What we want is a function that is polymorphic in the length of the cells, replacing the concrete dimension 3 with a variable dimension. For now, we'll call that dimension `$v` and write out an explicit abstraction for it:

```
(Iλ ((($v Dim)))
  (λ ((v [Int $v]))
    (reduce + 0 v)))
```

The `Iλ` form parameterizes over a dimension, which we then use to specify the type we want for `v`'s cells. This way, the function can be used on vectors of any length, as long as the length is passed in, analogous to passing a pointer to a buffer along with an integer indicating the buffer's size. The type of this function is roughly

```
(Π (($v Dim))
  (-> ([Int $v]) Int))
```

`Π` is a universal quantifier as typically used in dependently typed languages, though `Remora` imposes more limits on what computation can be used to decide how to instantiate a `Π` type.

A single-dimension `sum` is still a bit unsatisfying compared to our untyped version which works on all-ranked cells. In order to consume any arbitrarily high-ranked array as a single cell in typed code, we must quantify over shapes—sequences of dimensions—not just individual dimensions. We might naïvely try asking for any shape at all, remembering that the untyped `sum` would lift over anything at all:

```
(Iλ (((@v Shape)))
  (λ ((v [Int @v]))
    (reduce + 0 v)))
```

A `Shape` is an arbitrary-length sequence of `Dims`, which turns out to be *too* general. In order to reduce along an array's major axis, it must have a major axis. A scalar argument cannot be allowed. In order to make this work, we can parameterize over both the major axis and the remainder of the shape:

```
(Iλ (($v Dim) (@v Shape)))
  (λ ((v [Float $v @v]))
    (reduce + 0 v)))
```

Note that in the array-type shorthand, the atom type is followed by a sequence of indices, some of which are `Dims` while others are `Shapes`. These are two different sorts of things—individual dimensions and sequences of dimensions—but we will allow them to be interspersed as a syntactic convenience, such as in `[Int 8 @s 10]` indicating an array with major axis 8, minor axis 10, and `@s` standing for intermediate<sup>17</sup> axes. Analogous to the shorthand offered for writing arrays in terms of their atoms, the notation for a shape as a sequence of components implicitly converts `Dims` to singleton `Shapes` and concatenates those sequences. Concatenation on `Shapes` is possible in user code, via the `++` operation: `(++ (shape 3 4) (shape 2 5))` is equivalent to `(shape 3 4 2 5)`.

<sup>17</sup> Colleagues in the lab have suggested that these ought to be called the Dorian, Phrygian, etc. axes.

The other operation on indices is adding individual dimensions. This can be used to describe the behavior of functions like `append`, whose output’s leading dimension is the sum of its inputs’ leading dimensions. It can also be used to encode restrictions on argument cell shapes, such as limiting `mean` to avoid division by zero:

```
(Iλ (($v Dim) (@v Shape))
  (λ ((v [Int (+ 1 $v) @v]))
    (/ (sum v) (length v))))
```

Typed `Remora`’s use of boxes effectively treats ragged data and uncertain result shape as the same problem. Existential quantification can hide the uncertain portion of an array’s shape, such as giving `iota*` the output type  $(\Sigma ((@s \text{Shape})) [\text{Int } @s])$ , or the portion that makes it incompatible with its sibling cells, such as the typing the ragged array we got from lifting `iota*`:

```
[(box [0 1 2] ((@s Shape)) [Int @s])
 (box [0 1 2 3] ((@s Shape)) [Int @s])]
```

Note that each individual boxed array must be annotated with its  $\Sigma$  type. Without this annotation, it is uncertain what shape information is meant to be hidden and what is meant to be revealed. A box typed as  $(\Sigma ((@s \text{Shape})) [\text{Int } @s])$  makes no promises at all about the shape of the underlying array. The more specific type  $(\Sigma ((\$l \text{Dim})) [\text{Int } \$l])$  would guarantee that the box contains a vector, but that vector could be of any length.

Suppose we had `iota0` as a variant of `iota` which took a scalar, specifying the result vector length. Then `iota0` could have that more specific output type reflecting the guarantee that the output will be a boxed vector. Any computation using the unboxed result of `iota*` must be completely independent of the rank of the underlying array, which could be a scalar,



vector, matrix, *etc.*, but with `iota0`, downstream computation only needs to accommodate vectors. A function might even give an output type like  $(\Sigma ((\$l \text{ Dim})) [\text{Float } (1 + \$l)])$ , promising a nonempty vector of floats or  $(\Sigma ((\$l \text{ Dim})) [\text{Float } \$l \ \$l])$  promising a square matrix.

This points to a broader design principle for typed rank-polymorphic programming: Keep type-level information at type level instead of moving it to term level and hoping to reconstruct it later.

The code samples above gloss over another important issue, for the sake of conciseness. Built-in functions with polymorphic types still have to be instantiated before use. So a *fully* explicitly typed language is unsuitable for human use due to the immense and numerous type annotations required. However it is suitable for developing the formal semantics of Remora in Chapter 4. Mitigating the verbosity of cell-type annotations and explicit polymorphic instantiation will be one of the core goals of type inference in Part II.



This chapter presents a formal semantics for Remora. The formalism is explicit in quantification and instantiation for cell types, but rank-polymorphic frame lifting remains implicit. The mechanics of rank-polymorphic lifting are driven by types, rather than by inspecting array values' shapes, resolving the empty-frame problem encountered in some untyped code. In describing high-arity syntactic forms, ellipses identify sequences of syntactic elements.<sup>18</sup> So the pattern  $(+ n \dots)$  stands for an s-expression with the symbol  $+$  followed by zero or more subexpressions, which are collectively called  $n \dots$ .

<sup>18</sup> Following the style of *Macro-by-Example* [54]

An ellipsis may be applied to a larger fragment of syntax, such as in  $(++ (\text{shape } x \ y) \dots)$ . Here, we have a sequence of s-expressions, each of which is  $\text{shape}$  followed by two items. Successfully matching an s-expression to this pattern also identifies two sequences,  $x \dots$  and  $y \dots$ . They are, respectively, the sequence of all first dimensions from the shapes and the sequence of all second dimensions from the shapes.

Ellipses can be nested, as in  $(+ (+ n \dots) \dots)$ . This pattern describes a sequence of sequences of summands. We also lift this sequence notation to describe sequences of premises in judgment forms, such as checking the type of every argument in a function application form.

The formalism presented in this chapter is based on a model developed using PLT Redex [28]. The model is available at <https://github.com/jrslepak/Revised-Remora>. Within the model source, `language.rkt` defines the core language of this chapter (explicitly typed Remora), `typing-rules.rkt` implements typing and related judgments, and `reduction.rkt` gives the reduction relation on an extended version of the language which includes full type annotations.

## 4.1 SYNTAX

The grammar for Core Remora is given in Figure 4.1, with value forms specified in Figure 4.2. Term-level syntax is divided into atoms, written as  $a$ , and expressions, written as  $e$ . Expressions produce arrays, which contain atoms. For the most part, atom terms perform only trivial computation. This rule applies to base values, written as  $b$ ; primitive operators, written as  $\sigma$ ; and  $\lambda$ -abstractions, which may abstract over terms, types, and type indices. As an exception, a `box` gives an atomic view of an array of any shape and may therefore perform any computation to compute its contents. A `box` hides part of its contents' shape, using a dependent sum. It existentially quantifies type indices, but an explicit type annotation

$e \in Expr ::=$	<i>Expressions</i>
$x$	<i>Variable reference</i>
$  (array (n \dots) \mathfrak{a} \mathfrak{a} \dots)$	<i>Array, containing atoms</i>
$  (array (n \dots) \tau)$	<i>Empty array and its atom type</i>
$  (frame (n \dots) e e \dots)$	<i>Frame, containing array cells</i>
$  (frame (n \dots) \tau)$	<i>Empty frame and its cell type</i>
$  (e_f e_a \dots)$	<i>Term application</i>
$  (t\text{-app } e \tau \dots)$	<i>Type application</i>
$  (i\text{-app } e \iota \dots)$	<i>Index application</i>
$  (unbox (x_i \dots x_e e_s) e_b)$	<i>Let-binding box contents</i>
$\mathfrak{a} \in Atom ::=$	<i>Atoms</i>
$\mathfrak{b}$	<i>Base value</i>
$  \mathfrak{o}$	<i>Primitive operator</i>
$  (\lambda ((x \tau) \dots) e)$	<i>Term abstraction</i>
$  (T\lambda ((x k) \dots) e)$	<i>Type abstraction</i>
$  (I\lambda ((x \gamma) \dots) e)$	<i>Index abstraction</i>
$  (box \iota \dots e \tau)$	<i>Boxed array</i>
$\tau \in Type ::=$	<i>Types</i>
$x$	<i>Type variable</i>
$  B$	<i>Base type</i>
$  (A \tau \iota)$	<i>Array</i>
$  (-> (\tau \dots) \tau')$	<i>Function</i>
$  (\forall ((x k) \dots) \tau)$	<i>Universal</i>
$  (\Pi ((x \gamma) \dots) \tau)$	<i>Dependent product</i>
$  (\Sigma ((x \gamma) \dots) \tau)$	<i>Dependent sum</i>
$k \in Kind ::= Array \mid Atom$	<i>Kinds</i>
$\iota \in Idx ::=$	<i>Type indices</i>
$x$	<i>Type variable</i>
$  n$	<i>Single dimension</i>
$  (shape \iota \dots)$	<i>Sequence of dimensions</i>
$  (+ \iota \dots)$	<i>Adding dimensions</i>
$  (++ \iota \dots)$	<i>Appending shapes</i>
$\gamma \in Sort ::= Shape \mid Dim$	<i>Index sorts</i>
$\mathfrak{o} \in Op ::= \%+ \mid \% - \mid \%* \mid \% /$	<i>Primitive operators</i>
$  \%append \mid \%reduce$	
$  \%iota \mid \dots$	
$f \in Func ::= \mathfrak{o} \mid e$	<i>Functions</i>
$t \in Term ::= \mathfrak{a} \mid (\lambda ((x \tau) \dots) e)$	<i>Terms</i>

Figure 4.1: Core Remora grammar

$v \in Val ::= x \mid (\text{array } (n \dots) \mathfrak{v} \dots)$	<i>Values</i>
$\mathfrak{v} \in Atval ::= \mathfrak{b} \mid \mathfrak{o}$	<i>Atomic values</i>
$\mid (\lambda ((x \tau) \dots) e)$ $\mid (\top \lambda ((x k) \dots) e)$ $\mid (\text{I} \lambda ((x \gamma) \dots) e)$ $\mid (\text{box } \iota \dots v \tau)$	
$\mathbb{C} ::= \mathbb{E} \mid \mathbb{A}$	<i>Syntactic contexts</i>
$\mathbb{E} ::= \square$	<i>Expression contexts</i>
$\mid (\text{array } (n \dots) \mathfrak{a} \dots \mathbb{A} \mathfrak{a} \dots)$ $\mid (\text{frame } (n \dots) e \dots \mathbb{E} e \dots)$ $\mid (e \dots \mathbb{E} e \dots)$ $\mid (\text{t-app } \mathbb{E} \tau \dots)$ $\mid (\text{i-app } \mathbb{E} \iota \dots)$ $\mid (\text{unbox } (x_i \dots x_e \mathbb{E}) e)$ $\mid (\text{unbox } (x_i \dots x_e e) \mathbb{E})$	
$\mathbb{A} ::= \square$	<i>Atom contexts</i>
$\mid (\lambda ((x \tau) \dots) \mathbb{E})$ $\mid (\top \lambda ((x k) \dots) \mathbb{E})$ $\mid (\text{I} \lambda ((x \gamma) \dots) \mathbb{E})$ $\mid (\text{box } \iota \dots \mathbb{E} \tau)$	
$\mathbb{V} ::= \square$	<i>Evaluation contexts</i>
$\mid (\text{array } (n \dots) \mathfrak{a} \dots (\text{box } \iota \dots \mathbb{V} \tau) \mathfrak{a} \dots)$ $\mid (\text{frame } (n \dots) e \dots \mathbb{V} e \dots)$ $\mid (v \dots \mathbb{V} e \dots)$ $\mid (\text{t-app } \mathbb{V} \tau \dots)$ $\mid (\text{i-app } \mathbb{V} \iota \dots)$ $\mid (\text{unbox } (x_i \dots x_e \mathbb{V}) e)$ $\mid (\text{unbox } (x_i \dots x_e v) \mathbb{V})$	

Figure 4.2: Value forms, syntactic contexts, and evaluation contexts

is required. A box built from the index 3 and a  $3 \times 3$  matrix could be meant, for example, as an unspecified-length vector containing 3-vectors, with type  $(\Sigma ((n \text{ Dim})) (A \text{ Int (shape n 3)}))$  or as a square matrix of unspecified size, with type  $(\Sigma ((n \text{ Dim})) (A \text{ Int (shape n n)}))$ .

An array can be written as a literal, with its shape and individual atoms listed directly. It can also be written in nested form as a frame containing cells (its subexpressions) arranged in the specified shape. For example, the matrix  $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$  can be written as the literal

```
(array (2 2) 1 2 3 4)
```

or as a vector frame of vector literal cells:

```
(frame (2)
  (array (2) 1 2)
  (array (2) 3 4))
```

The frame notation allows construction of arrays from unevaluated cells. An empty array (*i.e.*, one whose shape includes a zero dimension) must be written with the type its elements are meant to have. An empty vector of integers is a different value than an empty vector of booleans, and they inhabit different types.

Term, type, and index abstractions can be applied to zero or more expressions, types, or indices. The body of the abstraction must itself be an expression, *i.e.*, all functions produce arrays as their results.

The `unbox` form unwraps each box in an array of boxes, let-binding its index- and term-level contents. Suppose we have  $M$ , a boxed square matrix of unspecified size. Unboxing  $M$  as in  $(\text{unbox } (\lambda a M) e)$  lets us use the index variable  $\lambda$  and term variable  $a$  within  $e$ , the body. The results from evaluating the body for each box's contents are then gathered together to produce the full result.

Types include base types (written as  $B$ ), functions, arrays, universal types, and dependent products and sums. Universals specify the kind of each type argument, and dependent products and sums specify the sort of each index argument. Types are classified as either `Atom` or `Array`. Type indices are naturals and sequences of naturals, with addition and appending as the only operators. They are classified into sorts, `Dim` and `Shape`.

The grammar in Figure 4.1 does not require any specific set of primitive operators, base types, and base values. An example collection of array-manipulation primitives and their types is given in Figure 4.3. Most of these primitives perform some operation along the argument's major axis. For example, `head` extracts the first scalar of a vector, the first row of a matrix, *etc.* This means that the argument shape must have one dimension more than the result shape, and that extra dimension must be nonzero. This is expressed in the type of `head` by giving the argument shape  $(++ (+ 1 d) s)$ , *i.e.*, a single dimension which is 1 plus any arbitrary natural followed by any arbitrary sequence of naturals. In taking

one scalar from a 3-vector, we would instantiate `d` as the dimension 2 and `s` as the empty shape (`shape`). If we want to extract the first plane of a  $5 \times 6 \times 7$  array, we use 4 for `d` and (`shape 6 7`) for `s`.

Since these operations work along the major axis, we can use other axes instead by instantiating them differently. Suppose `mtx` is the matrix (`array (3 2) 0 1 2 3 4 5`), with the type (`A Num (shape 3 2)`). Then `(t-app (i-app head 2 (shape 2)) Num)` is a function which extracts the first row of a  $(1+2) \times 2$  (i.e.,  $3 \times 2$ ) matrix. So `((t-app (i-app head 2 (shape 2)) Num) mtx)` evaluates to `(array (2) 0 1)`. Instead, consider `(t-app (i-app head 1 (shape)) Num)`. This is a function with input type (`A Num (shape 2)`) and output type (`A Num (shape)`). It extracts the first scalar of a 2-vector. When applied to `mtx`, this function *lifts* to extract the first scalar from *each* 2-vector, gathering the results as `(frame (3) (array () 0) (array () 2) (array () 4))`. Then evaluation proceeds, reducing to `(array (3) 0 2 4)`, the first column of `mtx`.

Several primitives must return boxed arrays because the type system cannot keep track of enough information to fully describe the result shape. As an extreme example, `read-nums` reads a vector of numbers from user input, and there is no way of knowing until run time how long a vector the user will enter. In other cases, the necessity of boxing comes from a limit on the type system’s expressive power. The `ravel` function produces a vector whose atoms are all those of the argument array, laid out in row-major order. The length of the `ravel` of some array is fully determined by that array’s shape: it is the product of all of its dimensions. However the undecidability of Peano arithmetic would interfere with type checking (not to mention future efforts on type inference). Since “product of all dimensions” is not expressible in Presburger arithmetic, we instead have `ravel` return a boxed vector.

Boxing is not limited to vectors. For example, `filter` uses a vector of booleans to decide which parts of an array to retain. Since the number of true entries in that vector is unknown, the size of the result’s major axis is also unknown. The resulting  $\Sigma$  type existentially quantifies only that one dimension, and leaves the remaining dimensions externally visible.

The `iota` functions and their variants, described in Figure 4.4, form a useful case study on what invariants can be expressed in Remora’s type system. These functions produce arrays whose atoms are successive natural numbers starting from 0, such as `(array (2 3) 0 1 2 3 4 5)`, representing the matrix  $\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}$ . The argument to `iota` is a vector of numbers specifying the result array’s shape. Since this vector can be dynamically computed, we cannot give any specific shape for `iota`’s return type. Instead, `iota` must return a box with existentially quantified shape. Recall that boxing arrays allows functions with data-dependent result shape to lift safely, since applying `iota` to `(array (2 2) 3 3 4 4)` must produce a  $3 \times 3$  matrix and a  $4 \times 4$  matrix as its two result cells.

<b>Function</b>	<b>Type</b>
head, tail	$(\rightarrow ((A\ t\ (+\ 1\ d))\ s)))$ $(A\ t\ s)$
behead, curtail	$(\rightarrow ((A\ t\ (+\ 1\ d))\ s)))$ $(A\ t\ (+\ (shape\ d)\ s)))$
length	$(\rightarrow ((A\ t\ (+\ (shape\ d)\ s)))$ $(A\ Int\ (shape)))$
shape, ravel	$(\rightarrow ((A\ t\ s))$ $(A\ (\Sigma\ ((d\ Dim))\ (A\ Int\ (shape\ d))))$ $(shape)))$
append	$(\rightarrow ((A\ t\ (+\ (shape\ m)\ s))$ $(A\ t\ (+\ (shape\ n)\ s)))$ $(A\ t\ (+\ (shape\ (+\ m\ n)\ s))))$
reverse	$(\rightarrow ((A\ t\ (+\ (shape\ d)\ s)))$ $(A\ t\ (+\ (shape\ d)\ s)))$
rotate	$(\rightarrow ((A\ t\ (+\ (shape\ d)\ s))$ $(A\ Int\ (shape)))$ $(A\ t\ (+\ (shape\ d)\ s)))$
fold	$(\rightarrow ((A\ (\rightarrow ((A\ t\ s)\ T)\ T)\ (shape))$ $T$ $(A\ t\ (+\ (shape\ d)\ s)))$ $T)$
reduce	$(\rightarrow ((A\ (\rightarrow ((A\ t\ s)\ (A\ t\ s))$ $(A\ t\ s))\ (shape))$ $(A\ t\ (+\ (shape\ (+\ 1\ d)\ s)))$ $(A\ t\ s)))$
scan	$(\rightarrow ((A\ (\rightarrow ((A\ u\ r)\ (A\ t\ s))$ $(A\ u\ r))\ (shape))$ $(A\ u\ r)$ $(A\ t\ (+\ (shape\ d)\ s)))$ $(A\ u\ (+\ (shape\ d)\ r)))$
filter	$(\rightarrow ((A\ Bool\ d)$ $(A\ t\ (+\ (shape\ d)\ s)))$ $(A\ (\Sigma\ ((k\ Dim))\ (A\ t\ (+\ (shape\ k)\ s)))$ $(shape)))$
read-nums	$(\rightarrow ())$ $(A\ (\Sigma\ ((k\ Dim))$ $(A\ Int\ (shape\ k)))\ (shape)))$
iota	$(\rightarrow ((A\ Int\ (shape\ d)))$ $(A\ (\Sigma\ ((s\ Shape))\ (A\ Int\ s))\ (shape)))$
reshape	$(\rightarrow ((A\ Int\ (shape\ d))$ $(A\ t\ r))$ $(A\ (\Sigma\ ((s\ Shape))\ (A\ Int\ s))\ (shape)))$

Figure 4.3: Common array-manipulation primitive operations and their Remora types. Each function type is wrapped in a scalar, with the function name bound at that scalar type in the base environment. For readability, we elide the enclosing  $\Pi$  and  $\forall$  forms.



Variants on `iota` allow the programmer to communicate more detailed knowledge to the type system. When the result is meant to be a vector, `iota/v` takes that vector’s length as the argument. The resulting box is typed as a vector of unknown length rather than an array of completely unknown shape. Knowing that we have a vector of numbers rather than any arbitrary array means, for example, that summing the box’s contents with `reduce` is certain to produce a scalar. We can thus type the following function as consuming and producing *non-boxed* scalars:

```
(λ ((n (A Num (shape))))
  (unbox (len nums (iota/v n))
    ((t-app (i-app reduce len (shape)) Num)
      +
      ((t-app (i-app append 1 len (shape)) Num)
        (array () 0)
        nums))))
```

In a more programmer-friendly surface language, with automatic instantiation of polymorphic functions and conversion of bare atoms to scalar arrays, this might be written as:

```
(λ ((n (A Num (shape))))
  (unbox (len nums (iota/v n))
    (reduce + (append [0] nums))))
```

Alternatively, the programmer might prefer to use `iota/s` to pass the desired result shape as a type index rather than as a term-level vector. In that case, there is no need to box the result array. In the automatic-instantiation shorthand, `iota/s` may be stylistically awkward, calling for the variant `iota/w`, which takes an extra array argument as a “shape witness” rather than instantiating at a shape index. Producing a number array whose shape matches some existing array `xs` could then be written as `(iota/w xs)` instead of `((i-app iota/s shape-of-xs))`.

The `reshape` function behaves similarly to `iota`, except that the atoms in the result array are drawn from the second argument, repeating them cyclically if necessary. So using `reshape` with the shape specification `(array (2) 3 2)` and the vector `(array (5) 1 2 3 4 5)` produces the  $3 \times 2$  matrix `(array (3 2) 1 2 3 4 5 1)`. Like `iota`, `reshape` benefits from alternative ways for the programmer to specify the result shape.

## 4.2 STATIC SEMANTICS

We present typing rules for Remora as well as supporting judgment forms: sorting rules for type indices, kinding rules for types, and a type equivalence judgment handling both  $\alpha$ -conversion of polymorphic types

<b>Function</b>	<b>Type</b>
<code>iota</code>	$(\rightarrow ((A \text{ Int } (\text{shape } d)))$ $(A (\Sigma ((s \text{ Shape})) (A \text{ Int } s)) (\text{shape})))$
<code>iota/v</code>	$(\rightarrow ((A \text{ Int } (\text{shape})))$ $(A (\Sigma ((d \text{ Dim})) (A \text{ t } (\text{shape } d)))$ $(\text{shape})))$
<code>iota/s</code>	$(\Pi ((s \text{ Shape}))$ $(\rightarrow () (A \text{ Int } s)))$
<code>iota/w</code>	$(\rightarrow ((A \text{ t } s))$ $(A \text{ Int } s))$

Figure 4.4: Types for `iota` and its variants. More detailed argument-shape information allows a more precise result shape: `iota/v` always produces a vector, while `iota/s` and `iota/w` have their result shape specified by their input.

and the multiple different ways a particular shape might be written. Typing Core Remora uses a three-part environment structure:  $\Theta$  is a partial function mapping index variables to sorts;  $\Delta$  maps type variables to their kinds; and  $\Gamma$  maps term variables to their types. The stratification of Dependent ML-style types allows indices to be checked using only the sort environment and types using only the sort and kind environments. Following the definition of each judgment form, we give a handful of lemmas which will be needed for a type soundness argument in Section 4.4. The well-formedness judgments each come with a lemma stating that the judgment gives a unique result to each well-formed term and that unique result is preserved by substituting well-formed assignments for free variables. When we show type soundness for Remora, these results will be needed to prove the preservation lemma. Uniqueness of typing is particularly important for Remora, where the implicit iteration in function application (including index and type abstractions) is driven by the types ascribed to the function and argument expressions. Well-defined program behavior relies on having a unique decomposition of each array into a frame of cells.

#### 4.2.1 *Sorting*

Figure 4.5 defines the sorting judgment,  $\Theta \vdash \iota :: \gamma$ , which states that in sort environment  $\Theta$ , the index  $\iota$  has sort  $\gamma$ . Natural number literals have sort `Dim`. A sequence of indices is a `Shape`, provided that every element of the sequence is a `Dim`. Addition is used on `Dim` arguments to produce a `Dim`. `Shape` arguments may be appended, to form another `Shape`. Variables may be bound at either sort, but they can only be introduced into the environment by index abstraction and unboxing terms—the index language itself has no binding forms.

$$\boxed{\Theta \vdash \iota :: \gamma}$$

$$\frac{n \in \mathbb{N}}{\Theta \vdash n :: \text{Dim}} \text{S:NAT} \qquad \frac{(x :: \gamma) \in \Theta}{\Theta \vdash x :: \gamma} \text{S:VAR}$$

$$\frac{\Theta \vdash \iota :: \text{Dim} \quad \dots}{\Theta \vdash (\text{shape } \iota \dots) :: \text{Shape}} \text{S:SHAPE} \qquad \frac{\Theta \vdash \iota :: \text{Dim} \quad \dots}{\Theta \vdash (+ \iota \dots) :: \text{Dim}} \text{S:PLUS}$$

$$\frac{\Theta \vdash \iota :: \text{Shape} \quad \dots}{\Theta \vdash (++ \iota \dots) :: \text{Shape}} \text{S:APPEND}$$

Figure 4.5: Sorting rules

We give two results about the well-behaved nature of the sorting rules: No type index inhabits two different sorts (in the same environment), and replacing an index’s variables with appropriately sorted indices does not change the sort.

**Lemma 4.2.1** (Uniqueness of sorting). *If  $\Theta \vdash \iota :: \gamma$  and  $\Theta \vdash \iota :: \gamma'$ , then  $\gamma = \gamma'$ .*

*Proof.* No non-variable index form is compatible with multiple sorting rules, so they can only have whichever sort their one compatible rule concludes. It remains to show that uniqueness holds for variables. Since  $\Theta$  is a well-defined partial function, mapping variables to sorts,  $\Theta(x)$  can only have one value. If  $\Theta(x) = \gamma$  and  $\Theta(x) = \gamma'$ ,  $\gamma = \gamma'$ .  $\square$

**Lemma 4.2.2** (Preservation of sorts under index substitution). *If  $\Theta, x :: \gamma_x \vdash \iota :: \gamma$  and  $\Theta \vdash \iota_x :: \gamma_x$  then  $\Theta \vdash \iota[x \mapsto \iota_x] :: \gamma$ .*

*Proof sketch.* This is straightforward induction on the original sort derivation.  $\square$

#### 4.2.2 Kinding

Kinding rules are given in Figure 4.6. The `Array` kind is only ascribed to types built by the array type constructor and type variables bound at that kind. The array type constructor requires as its arguments an `Atom` type and a `Shape` index. Base types are fundamental, non-aggregate types, such as `Float` or `Bool`, so they are `Atoms`. Function types have kind `Atom`, but their input and output types must be `Arrays`. This reflects the rule that application is performed on arrays, and the function produces an array result. Similarly, universal types and dependent products, describing type and index abstractions, must have an `Array` as their body, while they themselves are `Atoms`. This rules out types whose inhabitants would have to be syntactically illegal due to containing expressions instead of atoms as their bodies. Since boxes present arrays as atoms, dependent sum

$$\boxed{\Theta; \Delta \vdash \tau :: k}$$

$$\frac{(x :: k) \in \Delta}{\Theta; \Delta \vdash x :: k} \text{K:VAR} \qquad \frac{}{\Theta; \Delta \vdash B :: \text{Atom}} \text{K:BASE}$$

$$\frac{\Theta; \Delta \vdash \tau :: \text{Array} \quad \dots \quad \Theta; \Delta \vdash \tau' :: \text{Array}}{\Theta; \Delta \vdash (-> (\tau \dots) \tau') :: \text{Atom}} \text{K:FN}$$

$$\frac{\Theta; \Delta, x :: k \dots \vdash \tau :: \text{Array}}{\Theta; \Delta \vdash (\forall ((x k) \dots) \tau) :: \text{Atom}} \text{K:UNIV}$$

$$\frac{\Theta, x :: \gamma \dots; \Delta \vdash \tau :: \text{Array}}{\Theta; \Delta \vdash (\Pi ((x \gamma) \dots) \tau) :: \text{Atom}} \text{K:PI}$$

$$\frac{\Theta, x :: \gamma \dots; \Delta \vdash \tau :: \text{Array}}{\Theta; \Delta \vdash (\Sigma ((x \gamma) \dots) \tau) :: \text{Atom}} \text{K:SIGMA}$$

$$\frac{\Theta \vdash \iota :: \text{Shape} \quad \Theta; \Delta \vdash \tau :: \text{Atom}}{\Theta; \Delta \vdash (\text{A } \tau \iota) :: \text{Array}} \text{K:ARRAY}$$

Figure 4.6: Kinding rules

types also have an Array body and are kinded as Atoms. A universal type adds bindings for its quantified type variables to  $\Delta$ . Dependent products and sums do the same for their index variables in  $\Theta$ .

As with sorting of indices, we expect a well-kinded type to inhabit only a single kind (fixing a particular environment). The kinding system should also allow free index or type variables to be replaced with appropriately sorted or kinded indices or types without changing the original type's kind.

**Lemma 4.2.3** (Uniqueness of kinding). *If  $\Theta; \Delta \vdash \tau :: k$  and  $\Theta; \Delta \vdash \tau :: k'$ , then  $k = k'$ .*

*Proof.* As with uniqueness of sorting, no non-variable type is compatible with multiple kinding rules. Since all kinding rules except for K:VAR ascribe a specific kind, the only remaining case is for type variables. The kind environment  $\Delta$  is a well-defined partial function, so  $\Delta(x) = k$  and  $\Delta(x) = k'$  imply  $k = k'$ .  $\square$

**Lemma 4.2.4** (Preservation of kinds under index substitution). *If  $\Theta, x :: \gamma; \Delta \vdash \tau :: k$  and  $\Theta \vdash \iota_x :: \gamma$  then  $\Theta; \Delta \vdash \tau[x \mapsto \iota_x] :: k$ .*

*Proof sketch.* This is straightforward induction on the original kind derivation.  $\square$

**Lemma 4.2.5** (Preservation of kinds under type substitution). *Given  $\Theta; \Delta, x :: k_x \vdash \tau :: k$  and  $\Theta; \Delta \vdash \tau_x :: k_x$  then  $\Theta; \Delta \vdash \tau[x \mapsto \tau_x] :: k$ .*

*Proof sketch.* This is also provable by induction on the kind derivation for  $\tau$ .  $\square$

### 4.2.3 Typing

The typing rules in Figures 4.7 and 4.8 relate a full environment ( $\Theta$  mapping index variables to sorts,  $\Delta$  mapping type variables to kinds, and  $\Gamma$  mapping term variables to types), a term (whether an atom or an expression), and its type under that environment. Since an array type might have its shape described in multiple different ways, *e.g.*, a vector of length 6 or a vector of length (+ 1 5), the T:EQV rule makes reference to a type-equivalence judgment (presented in full detail in §4.2.4) which reconciles such differences according to the algebraic theory of type indices.

The signature  $\mathcal{S}$ , referenced in the T:OP rule, is a function mapping from primitive operators to their types. For example,  $\mathcal{S}[+]$  is  $(\rightarrow ((A \text{ Num (shape)}) (A \text{ Num (shape)})) (A \text{ Num (shape)}))$ , meaning  $+$  is an operator which consumes two scalar numbers and produces one scalar number.

Array literals (T:ARRAY) and nested frames (T:FRAME) both include a length check: the number of atoms or cells must be equal to the product of the given dimensions. In the case of empty arrays, the length-matching condition is fulfilled if and only if the array has a  $\emptyset$  as one of its dimensions. Term, type, and index abstractions (T:LAM, T:TLAM, and T:ILAM respectively) all bind their arguments' names in the appropriate environment.

Typing function application (T:APP) starts by identifying the type of the expression in function position. It must be an array of functions, and the array's entire shape  $\iota_f$  is treated as the function frame. The function input types, also arrays, specify the element type and cell shape for each argument. Each cell shape  $\iota$  must be a suffix of the shape of the corresponding actual argument; the remainder  $\iota_a$  is the argument's frame. In the semilattice defined by the sequence-prefix relation  $\sqsubseteq$ , the least upper bound of a collection of sequences (written as  $\sqcup$ ) is the one which has all the others as prefixes. The maximum of these frames under prefix ordering (where  $[23] \sqsubseteq [232]$  but  $[23] \not\sqsubseteq [632]$ ) is the principal frame  $\iota_p$ . That is, the function and argument arrays will all be lifted so as to have  $\iota_p$  as their frames when the program runs. Then  $\iota_p$  is used as the frame around the function's output type to give the result type for this function application.

Type application (T:TAPP) and index application (T:IAPP) also require arrays in function position, but they can skip prefix comparison as type and index arguments do not come in arrays that must be split into frames of cells. Thus the function's frame shape  $\iota_f$  passes through

$$\boxed{\Theta; \Delta; \Gamma \vdash t : \tau}$$

$$\frac{}{\Theta; \Delta; \Gamma \vdash \mathfrak{a} : \mathcal{S}[\mathfrak{a}]} \text{ T:OP} \qquad \frac{(x : \tau) \in \Gamma}{\Theta; \Delta; \Gamma \vdash x : \tau} \text{ T:VAR}$$

$$\frac{\Theta; \Delta; \Gamma \vdash t : \tau' \quad \tau \cong \tau'}{\Theta; \Delta; \Gamma \vdash t : \tau} \text{ T:EQV}$$

$$\frac{\Theta; \Delta; \Gamma \vdash \mathfrak{a} : \tau \quad \dots \quad \Theta; \Delta \vdash \tau :: \text{Atom} \quad \text{Length}[\llbracket \mathfrak{a} \dots \rrbracket] = \prod n \dots}{\Theta; \Delta; \Gamma \vdash (\text{array } (n \dots) \mathfrak{a} \dots) : (\text{A } \tau \text{ (shape } n \dots)})} \text{ T:ARRAY}$$

$$\frac{\Theta; \Delta \vdash \tau :: \text{Atom} \quad 0 \in n \dots}{\Theta; \Delta; \Gamma \vdash (\text{array } (n \dots) \tau) : (\text{A } \tau \text{ (shape } n \dots)})} \text{ T:OA}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e : (\text{A } \tau \iota) \quad \dots \quad \Theta; \Delta \vdash (\text{A } \tau \iota) :: \text{Array} \quad \text{Length}[\llbracket e \dots \rrbracket] = \prod n \dots}{\Theta; \Delta; \Gamma \vdash (\text{frame } (n \dots) e \dots) : (\text{A } \tau \text{ (} ++ \text{ (shape } n \dots) \iota))} \text{ T:FRAME}$$

$$\frac{\Theta; \Delta \vdash \tau :: \text{Atom} \quad \Theta \vdash \iota :: \text{Shape} \quad 0 \in n \dots}{\Theta; \Delta; \Gamma \vdash (\text{frame } (n \dots) (\text{A } \tau \iota)) : (\text{A } \tau \text{ (} ++ \text{ (shape } n \dots) \iota))} \text{ T:OF}$$

$$\frac{\Theta; \Delta; \Gamma, x : \tau \dots \vdash e : \tau' \quad \Theta; \Delta \vdash \tau :: \text{Array} \quad \dots}{\Theta; \Delta; \Gamma \vdash (\lambda ((x \tau) \dots) e) : (\rightarrow (\tau \dots) \tau')} \text{ T:LAM}$$

$$\frac{\Theta; \Delta, x :: k \dots; \Gamma \vdash e : \tau}{\Theta; \Delta; \Gamma \vdash (\text{T}\lambda ((x k) \dots) e) : (\forall ((x k) \dots) \tau)} \text{ T:TLAM}$$

$$\frac{\Theta, x :: \gamma \dots; \Delta; \Gamma \vdash e : \tau}{\Theta; \Delta; \Gamma \vdash (\text{I}\lambda ((x \gamma) \dots) e) : (\Pi ((x \gamma) \dots) \tau)} \text{ T:ILAM}$$

$$\frac{\Theta \vdash \iota :: \gamma \quad \dots \quad \Theta; \Delta \vdash (\Sigma ((x \gamma) \dots) \tau) :: \text{Atom} \quad \Theta; \Delta; \Gamma \vdash e : \tau[x \mapsto \iota, \dots]}{\Theta; \Delta; \Gamma \vdash (\text{box } \iota \dots e (\Sigma ((x \gamma) \dots) \tau)) : (\Sigma ((x \gamma) \dots) \tau)} \text{ T:BOX}$$

Figure 4.7: Typing rules (introduction forms)

$$\boxed{\Theta; \Delta; \Gamma \vdash t : \tau}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e : (A (\forall ((x k) \dots) (A \tau_u \iota_u))) \iota_f \quad \Theta; \Delta \vdash \tau :: k \quad \dots}{\Theta; \Delta; \Gamma \vdash (\text{t-app } e \tau \dots) : (A \tau_u[x \mapsto \tau, \dots] \text{ } (++) \iota_f \iota_u))} \text{T:TAPP}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e : (A (\Pi ((x \gamma) \dots) (A \tau_p \iota_p))) \iota_f \quad \Theta \vdash \iota :: \gamma \quad \dots}{\Theta; \Delta; \Gamma \vdash (\text{i-app } e \iota \dots) : (A \tau_p[x \mapsto \iota, \dots] \text{ } (++) \iota_f \iota_p[x \mapsto \iota, \dots]))} \text{T:IAPP}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e_s : (A (\Sigma ((x'_i \gamma) \dots) \tau_s) \iota_s) \quad \Theta, x_i :: \gamma \dots; \Delta; \Gamma, x_e : \tau_s[x'_i \mapsto x_i, \dots] \vdash e_b : (A \tau_b \iota_b) \quad \Theta; \Delta \vdash (A \tau_b \iota_b) :: \text{Array}}{\Theta; \Delta; \Gamma \vdash (\text{unbox } (x_i \dots x_e e_s) e_b) : (A \tau_b (++) \iota_s \iota_b))} \text{T:UNBOX}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e_f : (A (-> ((A \tau \iota) \dots) (A \tau' \iota')) \iota_f) \quad \Theta; \Delta; \Gamma \vdash e_a : (A \tau (++) \iota_a \iota) \quad \dots \quad \iota_p = \bigsqcup \{ \iota_f \iota_a \dots \}}{\Theta; \Delta; \Gamma \vdash (e_f e_a \dots) : (A \tau' (++) \iota_p \iota')} \text{T:APP}$$

Figure 4.8: Typing rules (elimination forms)

unaltered, and arguments are substituted into the body type  $\tau_b$  to produce the resulting array's element type.

When constructing a box (T:BOX), a dependent-sum type annotation is provided. The box's index components must match their declared sorts, and substituting them into the body of the dependent sum type must produce a type that matches the box's array component. Unboxing (T:UNBOX) requires that  $e_{\text{box}}$ , the expression being destructured, be a dependent sum. The unbox form names the sum's index and array components and adds them to the sort and type environments when checking  $e_{\text{body}}$ . Although the index components are in scope while checking the body, information hidden by the existential is not permitted to leak out: The end result type  $\tau_{\text{body}}$  must be well-formed without relying on the extended sort environment. Unboxing a frame of boxes (scalars) produces a frame of result cells, similar to lifting function application.

Anticipating a progress lemma, we prove a canonical-forms lemma for Remora's typing rules. Following the atom/array distinction, we have separate lemmas for atoms and arrays. Although an atom can contain an array if that atom is a box, we avoid mutual dependence between the lemmas by not making any claim about the syntactic structure of the box's contents.

**Lemma 4.2.6** (Canonical forms for atomic values). *Let  $v$  be a well-typed atomic value, that is,  $\cdot; \cdot; \cdot \vdash v : \tau$ .*

1. *If  $\tau$  is of the form  $(-> (\tau_i \dots) \tau_o)$ , then  $v$  is of the form  $v$  or  $(\lambda ((x \tau_i) \dots) e)$ .*

2. If  $\tau$  is of the form  $(\forall ((x k) \dots) \tau_u)$ ,  
then  $v$  is of the form  
 $(\top\lambda ((x_u k) \dots) e)$ .
3. If  $\tau$  is of the form  $(\Pi ((x \gamma) \dots) \tau_p)$ ,  
then  $v$  is of the form  
 $(\text{I}\lambda ((x_p \gamma) \dots) e)$ .
4. If  $\tau$  is of the form  $(\Sigma ((x \gamma) \dots) \tau_b)$ ,  
then  $v$  is of the form  
 $(\text{box } \iota \dots v_b (\Sigma ((x_b \gamma) \dots) \tau'_b))$ ,  
with  $\tau \cong (\Sigma ((x_b \gamma) \dots) \tau'_b)$ .
5. If  $\tau$  is of the form  $B$ ,  
then  $v$  is of the form  
 $b$ .

*Proof sketch.* The type derivation may end with T:EQV, so we consider the sub-derivation prior to all final T:EQV instances. We examine which typing rules can ascribe a type of the right form and then identify what form the term must take to match those rule.  $\square$

**Lemma 4.2.7** (Canonical forms for arrays). *Let  $v$  be a well-typed value, that is,  $\cdot; \cdot; \cdot \vdash v : \tau$ ,*

1. If  $\tau$  is of the form  $(A (-> (\tau_i \dots) \tau_o) \iota)$ ,  
then  $v$  is of the form  
 $(\text{array } (n \dots) \bar{f} \dots)$ .
2. If  $\tau$  is of the form  $(A (\forall ((x k) \dots) \tau_u) \iota)$ ,  
then  $v$  is of the form  
 $(\text{array } (n \dots) (\top\lambda ((x_u k) \dots) e) \dots)$ .
3. If  $\tau$  is of the form  $(A (\Pi ((x \gamma) \dots) \tau_p) \iota)$ ,  
then  $v$  is of the form  
 $(\text{array } (n \dots) (\text{I}\lambda ((x_p \gamma) \dots) e) \dots)$ .
4. If  $\tau$  is of the form  $(A (\Sigma ((x \gamma) \dots) \tau_b) \iota)$ ,  
then  $v$  is of the form  
 $(\text{array } (n \dots) (\text{box } \iota \dots v_b (\Sigma ((x_b \gamma) \dots) \tau_b)) \dots)$ ,  
with  $\tau \cong (\Sigma ((x_b \gamma) \dots) \tau'_b)$ .
5. If  $\tau$  is of the form  $(A B \iota)$ ,  
then  $v$  is of the form  
 $(\text{array } (n \dots) b \dots)$ ,  
with  $\cdot; \cdot; \cdot \vdash b : B$  for each of  $b \dots$ .

*Proof sketch.* This proceeds like the proof for Lemma 4.2.6.  $\square$



$$\boxed{\tau \cong \tau'}$$

$$\frac{}{\tau \cong \tau} \text{TEQV:REFL} \qquad \frac{\tau \cong \tau' \quad \models \iota \equiv \iota'}{(A \tau \iota) \cong (A \tau' \iota')} \text{TEQV:ARRAY}$$

$$\frac{\tau_{ij} \cong \tau'_{ij} \quad \dots \quad \tau_o \cong \tau'_o}{(-> (\tau_i \dots) \tau_o) \cong (-> (\tau'_i \dots) \tau'_o)} \text{TEQV:FN}$$

$$\frac{\tau[x \mapsto x_f, \dots] \cong \tau'[x' \mapsto x_f, \dots] \quad \text{with fresh } x_f \dots}{(\forall ((x k) \dots) \tau) \cong (\forall ((x' k) \dots) \tau')} \text{TEQV:UNIV}$$

$$\frac{\tau[x \mapsto x_f, \dots] \cong \tau'[x' \mapsto x_f, \dots] \quad \text{with fresh } x_f \dots}{(\Pi ((x \gamma) \dots) \tau) \cong (\Pi ((x' \gamma) \dots) \tau')} \text{TEQV:PI}$$

$$\frac{\tau[x \mapsto x_f, \dots] \cong \tau'[x' \mapsto x_f, \dots] \quad \text{with fresh } x_f \dots}{(\Sigma ((x \gamma) \dots) \tau) \cong (\Sigma ((x' \gamma) \dots) \tau')} \text{TEQV:SIGMA}$$

Figure 4.9: Type equivalence

#### 4.2.4 Type Equivalence

Remora’s typing rules rely on a type-equivalence relation defined in Figure 4.9. The equivalence relation is essentially  $\alpha$ -equivalence augmented with a check as to whether array shapes are guaranteed to be equal.

The index equality check in `TEQV:ARRAY`, stated as  $\models \iota \equiv \iota'$ , asks whether the equality of  $\iota$  and  $\iota'$  is valid, *i.e.*, whether it is true for every possible choice of values for the variables appearing in the equality. This is based on the algebraic laws of the theory of type indices. For example, since appending is associative, the type indices  $(++ (++ \iota_0 \dots) \iota_1 \dots)$ ,  $(++ \iota_0 \dots (++ \iota_1 \dots))$ , and  $(++ \iota_0 \dots \iota_1 \dots)$  are all equal, no matter the values chosen for the variables appearing in  $\iota_0 \dots$  and  $\iota_1 \dots$ . So any nesting of `++` forms can be rewritten in a canonical form by flattening. Individual dimensions are sums of natural-number literals and variables, *i.e.*, affine combinations of variables. So a dimension can be written in a canonical representation with the number of occurrences of each variable and the total of all natural-number literals.

Then type indices  $\iota$  and  $\iota'$ —whether they are shapes or dimensions—are guaranteed to be equal if and only if their canonical forms are the same. Two shapes built by appending components must have the components match, or else an assignment of variables might give corresponding components different lengths or place a mismatching dimension in components of equal length. For example, the equality  $(++ x y) \equiv (++ x x)$  is falsified by any choice of  $x$  which has a different length than  $y$  (such as  $x = (\text{shape } 3)$ ,  $y = (\text{shape } 3 \ 3)$ ) or which makes any individual dimension differ (such as  $x = (\text{shape } 2 \ 3)$ ,  $y = (\text{shape } 3 \ 3)$ ). When

coefficients on variables within corresponding dimensions do not match perfectly, that is an opportunity for a variable assignment to force those dimensions to differ, *e.g.*,  $(\text{shape } (+ a a 2) 4) \equiv (\text{shape } (+ b a) 4)$  can be falsified by choosing  $a = 1, b = 2$ .

We expect the relation  $\cong$  actually to be an equivalence relation, *i.e.*, reflexive, symmetric, and transitive. Only reflexivity has its own inference rule, so we now show symmetry and transitivity.

**Lemma 4.2.8** (Symmetry of  $\cong$ ). *If  $\tau \cong \tau'$ , then  $\tau' \cong \tau$ .*

*Proof sketch.* This follows via straightforward induction on the equivalence derivation.  $\square$

**Lemma 4.2.9** (Transitivity of  $\cong$ ). *If  $\tau_0 \cong \tau_1$  and  $\tau_1 \cong \tau_2$ , then  $\tau_0 \cong \tau_2$ .*

*Proof sketch.* This follows from induction on the derivations of  $\tau_0 \cong \tau_1$  and  $\tau_1 \cong \tau_2$ . Since both derivations mention  $\tau_1$ , the structure of the equivalence rules prohibits the derivations from ending with different rules (other than TEQV:REFL, which passes that structural requirement on to its premises).  $\square$

**Theorem 4.2.1.**  *$\cong$  is an equivalence relation.*

A type-equivalence relation should not cross kind boundaries. Violation of this principle would allow use of T:EQV to ascribe an ill-kinded type to a well-typed term. It follows directly from inspection of the equivalence rules that they will not relate an `Atom` with an `Array`, but correct use of type and index variables remains to be proven. To that end, we show that two equivalent types will be ascribed the same kind by the same environment.

**Lemma 4.2.10.** *If  $\Theta; \Delta \vdash \tau :: k$  and  $\tau \cong \tau'$ , then  $\Theta; \Delta \vdash \tau' :: k$ .*

*Proof sketch.* This result is proven by induction on the derivation of  $\tau \cong \tau'$ . In each case, the induction hypothesis converts a kind derivation for some fragment of  $\tau$  into a kind derivation for a corresponding fragment of  $\tau'$  (and similar for index fragments).  $\square$

We also expect type equivalence to be well-behaved under substitution. Ultimately, substituting equivalent types or indices into equivalent types ought to produce equivalent types. Proving that result by induction on derivation of equivalence is straightforward except for the REFL case.

**Lemma 4.2.11.** *If  $\models \iota \equiv \iota'$ , then for any index variable  $x$ ,  $\tau[x \mapsto \iota] \cong \tau[x \mapsto \iota']$ .*

*Proof sketch.* This is provable using induction on the structure of  $\tau$ . Only the case for arrays makes direct use of  $\iota$  and  $\iota'$ ; the other cases simply use the induction hypothesis to prove the premises of the derivation of  $\tau[x \mapsto \iota] \cong \tau[x \mapsto \iota']$ .  $\square$

**Lemma 4.2.12.** *If  $\tau_x \cong \tau'_x$ , then for any type variable  $x$ ,  $\tau[x \mapsto \tau_x] \cong \tau[x \mapsto \tau'_x]$ .*

*Proof.* We use induction on the structure of  $\tau$ . The cases for universals, dependent products, and dependent sums require instantiating the induction hypothesis with a substitution of fresh type variables  $x_f \dots$ . For example, when  $\tau = (\forall ((x_u k) \dots) \tau_u)$ , the induction hypothesis promises the equivalence of  $\tau_u$  after substituting in  $x_f \dots$  for  $x_u \dots$  and also  $\tau_x$  or  $\tau'_x$  for  $x$ .  $\square$

**Theorem 4.2.2.** *If  $\tau \cong \tau'$  and  $\tau_x \cong \tau'_x$ , then for any type variable  $x$ ,  $\tau[x \mapsto \tau_x] \cong \tau'[x \mapsto \tau'_x]$ .*

*Proof sketch.* We use induction on the derivation of  $\tau \cong \tau'$ . In each case, the induction hypothesis provides equivalence derivations for corresponding fragments of  $\tau[x \mapsto \tau_x]$  and  $\tau'[x \mapsto \tau'_x]$ , which can then be used to prove the substituted types themselves equivalent.  $\square$

Having defined the typing judgment and the type-equivalence relation on which it builds, we can now prove the usual results about typing in Remora. The T:EQV rule can allow many types to be ascribed to a single term, but we will prove that an environment and term can only map to a single equivalence class.

**Theorem 4.2.3** (Uniqueness of typing, up to equivalence). *If  $\Theta; \Delta; \Gamma \vdash t : \tau$  and  $\Theta; \Delta; \Gamma \vdash t : \tau'$ , then  $\tau \cong \tau'$ .*

*Proof sketch.* This can be proven by induction on  $t$ , showing that all derivations of  $\Theta; \Delta; \Gamma \vdash t : \tau'$  must end with the same non-T:EQV rule (chosen according to the structure of  $t$ ) followed by 0 or more T:EQV instances, which keeps the result in the same equivalence class as  $\tau$ .  $\square$

We also require guarantees about substitution in terms: replacing an index variable with an appropriately sorted index, a type variable with an appropriately kinded type, or a term variable with an appropriately typed expression should not change the type of the original term. If substitution turns a term  $t$  with type  $\tau$  into  $t'$  with type  $\tau'$ , where  $\tau \cong \tau'$ , we can add a T:EQV at the end of the new type derivation to conclude  $t'$  has type  $\tau$ . As such, we do not need to include an “up to equivalence” caveat when stating the preservation of typing lemmas.

**Lemma 4.2.13** (Preservation of types under index substitution). *Given  $\Theta, x :: \gamma; \Delta; \Gamma \vdash t : \tau$  and  $\Theta \vdash \iota_x :: \gamma$  then  $\Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash t[x \mapsto \iota_x] : \tau[x \mapsto \iota_x]$ .*

*Proof sketch.* This is straightforward induction on the type derivation  $\Theta, x :: \gamma; \Delta; \Gamma \vdash t : \tau$ .  $\square$

**Lemma 4.2.14** (Preservation of types under type substitution).

*Given  $\Theta; \Delta, x :: k; \Gamma \vdash t : \tau$  and  $\Theta; \Delta \vdash \tau_x :: k$ , then  $\Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash t[x \mapsto \tau_x] : \tau[x \mapsto \tau_x]$ .*

*Proof sketch.* This is straightforward induction on the type derivation  $\Theta; \Delta, x :: k; \Gamma \vdash t : \tau$ .  $\square$

**Lemma 4.2.15** (Preservation of types under term substitution). *Given  $\Theta; \Delta; \Gamma, x : \tau_x \vdash t : \tau$  and  $\Theta; \Delta; \Gamma \vdash e_x : \tau_x$  then  $\Theta; \Delta; \Gamma \vdash t[x \mapsto e_x] : \tau$ .*

*Proof sketch.* We use induction on the derivation of  $\Theta; \Delta; \Gamma, x : \tau_x \vdash t : \tau$ .  $\square$

We call an environment well-formed, written as  $\Theta; \Delta \vdash \Gamma$ , if for every binding  $x : \tau \in \Gamma$ , we can derive  $\Theta; \Delta \vdash \tau :: \text{Array}$ . This is the expected case, rather than permitting  $\tau$  to have kind `Atom`, because a lone variable is an expression and ought to stand for an array value.

When we show that the typing judgment only ascribes types of the appropriate kind, the case for the `T:EQV` rule relies on the earlier lemma that the type equivalence relation respects kinding, *i.e.*, two equivalent types will have the same kind when checked in the same environment.

**Theorem 4.2.4** (Ascription of well-kinded types). *Given  $\Theta; \Delta; \Gamma \vdash t : \tau$  where  $\Theta; \Delta \vdash \Gamma$ :*

- *If  $t$  is an expression, then  $\Theta; \Delta \vdash \tau :: \text{Array}$*
- *If  $t$  is an atom, then  $\Theta; \Delta \vdash \tau :: \text{Atom}$*

*Proof sketch.* This follows by induction on the derivation of  $\Theta; \Delta; \Gamma \vdash t : \tau$ . It is not sufficient to point out that each typing rule ascribes a type whose form matches the appropriate kind. Elimination-form cases call for a little extra work. For `UNBOX`, the kind check on the result type is necessary to ensure that existentially quantified variables do not leak out. The `APP` case must ensure that index variables in the ascribed type actually appear in the environment. This is guaranteed because the principal frame is always chosen to be one of the function- or argument-position frames.  $\square$

### 4.3 DYNAMIC SEMANTICS

In the dynamic semantics for Remora, the way function application is lifted to work on aggregate data depends on the types of the function and argument terms. Consulting type information avoids a “hole” in the semantics of untyped array-oriented code, where a frame whose shape includes a 0 dimension evaluates to an array with indeterminate shape—there are no concrete cells whose shape can be used to determine the overall shape of the resulting array. Instead, the function’s type tells us the shape of the resulting cells, even when there are zero such cells.

The small-step operational semantics is the compatible closure of the reduction rules given in Figure 4.11, using the evaluation contexts  $\mathbb{V}$  defined in Figure 4.2. It assumes every atom and expression has

$$\begin{aligned}
& \text{Split}_n \llbracket (a_1, \dots, a_m) \rrbracket \\
&= ((a_1, \dots, a_n), (a_{n+1}, \dots, a_{2n}), \dots, (a_{m-n+1}, \dots, a_m)) \\
& \text{Rep}_n \llbracket (a_1, \dots, a_m) \rrbracket \\
&= (a_{1,1}, \dots, a_{1,n}, \dots, a_{m,1}, \dots, a_{m,n}) \quad \text{where } a_{i,j} = a_i \\
& \text{Concat} \llbracket ((a_{1,1}, \dots, a_{1,n}), \dots, (a_{m,1}, \dots, a_{m,n})) \rrbracket \\
&= (a_{1,1}, \dots, a_{1,n}, \dots, a_{m,1}, \dots, a_{m,n}) \\
& \text{Transpose}((a_{1,1}, \dots, a_{1,n}), \dots, (a_{m,1}, \dots, a_{m,n})) \\
&= ((a_{1,1}, \dots, a_{m,1}), \dots, (a_{1,n}, \dots, a_{m,n}))
\end{aligned}$$

Figure 4.10: List-processing metafunctions

been tagged with its type. For example,  $\beta$ -reduction requires that each atom in the function-position array have input types  $\tau \dots$  and that the argument arrays' types also match  $\tau \dots$ . This matching is still subject to the type-equivalence rules described in §4.2, *e.g.*, a function tagged as having input type  $(A \text{ Int } (++) (\text{shape } 3) (\text{shape } 4))$  can be applied to an argument tagged with type  $(A \text{ Int } (\text{shape } 3 \ 4))$ . Because every term now has type annotations attached, we drop the “empty” array and frame syntactic forms. Their replacements use the standard array and frame syntax with an empty list of atoms or cells, and the atom or cell type is implied by the expression's type annotation.

Several list-processing metafunctions are used in defining the reduction rules. These metafunctions are defined in Figure 4.10.  $\text{Split}_n$  turns a list into a list of lists, made up of the consecutive length- $n$  pieces of the original list. For example,  $\text{Split}_3 \llbracket (1 \ 2 \ 3 \ 4 \ 5 \ 6) \rrbracket$  is  $((1 \ 2 \ 3) \ (4 \ 5 \ 6))$ .  $\text{Concat}$  flattens a list of lists into a single list, effectively undoing a  $\text{Split}$ .  $\text{Rep}_n$  constructs a new list by repeating each element of the original list  $n$  times.  $\text{Rep}_2 \llbracket (0 \ 1) \rrbracket$  is  $(0 \ 0 \ 1 \ 1)$ . Used on nested lists, the inner lists are treated atomically:  $\text{Rep}_2 \llbracket ((1 \ 2 \ 3) \ (4 \ 5 \ 6)) \rrbracket$  is  $((1 \ 2 \ 3) \ (1 \ 2 \ 3) \ (4 \ 5 \ 6) \ (4 \ 5 \ 6))$ .  $\text{Transpose}$  takes a list of lists, where the inner lists all have the same length, and produces a new list of lists whose  $i^{\text{th}}$  element contains the  $i^{\text{th}}$  elements of each original inner list.

The reduction rules themselves are given in Figure 4.11. Remora's function application is split into stages for replicating cells to make frame shapes match (*lift*), mapping the functions to corresponding argument cells (*map*), and gathering the result cells back into an array (*collapse*).

Performing a *lift* step identifies the function array's frame, the sequence  $[n_f \dots]$ , and each argument's frame,  $[n_a \dots]$ . Then the sequence  $[n_p \dots]$  is chosen to be the largest frame according to prefix ordering. We require that at least one function or argument frame be different from

the principal frame—otherwise, a *map* step would be appropriate instead. Each argument’s cell size  $n_{ac}$  is the product of the dimensions  $[n_{in} \dots]$  of the function’s input type at that position; the function array’s cell size is always 1. The number of replicas needed for each cell ( $n_{fe}$  for the function and  $n_{fa}$  for each argument) is determined by multiplying the dimensions that must be added to each corresponding frame to produce the principal frame, *i.e.*, the principal frame minus whatever prefix was already present in the original array’s shape. Given these numbers, we split each array’s atom list into its cells, replicate those cells to match the new array shapes, and then concatenate each array’s replicated cells to produce the new function and argument arrays. Type annotations on the individual arrays update to reflect their new shapes, but the application form’s type remains unchanged.

A *map* step is possible when every piece of a function application has the same frame shape. Then the application becomes a frame of application forms, which themselves all have scalar principal frame. This requires breaking each argument array’s atom list into its individual cells’ atom lists, then transposing to match the first cell of each argument with the first function, the second cell of each argument with the second function, and so on.

When function application has a scalar in function position, and every argument array matches the function’s corresponding input type, then we can  $\beta$ -reduce or  $\delta$ -reduce.  $\beta$ -reduction performs conventional  $\lambda$ -calculus substitution. The  $\delta$  rule uses a family of metafunctions, each associated with a primitive operator. No *f* frame construct is necessary in either result, as this is the degenerate case of function lifting—the principal frame is scalar.

Applying type and index abstractions is handled by the  $t\beta$  and  $i\beta$  rules. The application frame is the shape of the array of type or index abstractions, since there are no argument *arrays*. Every  $\top\lambda$  or  $\Gamma\lambda$  is applied to the full list of type or index arguments. Substitution into the body of each abstraction should be read as affecting type annotations as well as subterms: if we are replacing the type variable  $\top$  with  $\text{Int}$ , then  $\chi^{(\text{A } \top \text{ (shape 3)})}$  becomes  $\chi^{(\text{A } \text{Int} \text{ (shape 3)})}$ .

Once a *f* frame has every one of its cells reduced to an array literal, the nested representation can be merged into a single literal. In the case where one of  $n \dots$  is 0, there will be no cells to examine to determine the cell dimensions  $n' \dots$ , so this information is taken from the type annotation on the *f* frame form. The type annotation itself passes through unchanged. The atom lists from the cells are concatenated to produce the collapsed array’s atom list.

Destructing a *box* with an *unbox* form behaves like a conventional *let*. The result is the body  $e$ , where the index variables  $x_i \dots$  are replaced with the *box*’s indices  $\iota \dots$ , and the term variable  $x_e$  is replaced with the contained array  $v$ .

$$\begin{aligned}
& ((\text{array } (n_f \dots) \mathbf{v}_f \dots)^{(A \rightarrow ((A \tau_i (\text{shape } n_i \dots)) \dots) \tau_o) (\text{shape } n_f \dots)}) \\
& (\text{array } (n_a \dots n_i \dots) \mathbf{v}_a \dots)^{(A \tau_i (\text{shape } n_a \dots n_i \dots)) \dots}) \\
& \mapsto_{\text{lift}} \\
& ((\text{array } (n_p \dots) \\
& \quad \text{Concat} \llbracket \text{Rep}_{n_{fe}} \llbracket \text{Split}_1 \llbracket \mathbf{v}_f \dots \rrbracket \rrbracket \rrbracket \rrbracket)^{(A \rightarrow ((A \tau_i (\text{shape } n_i \dots)) \dots) \tau_o) (\text{shape } n_p \dots)}) \\
& (\text{array } (n_p \dots n_i \dots) \\
& \quad \text{Concat} \llbracket \text{Rep}_{n_{ae}} \llbracket \text{Split}_{n_{ac}} \llbracket \mathbf{v}_a \dots \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket)^{(A \tau_i (\text{shape } n_p \dots n_i \dots)) \dots}) \\
& \text{where} \\
& \text{Not all of } (n_f \dots), (n_a \dots) \dots \text{ are equal}
\end{aligned}$$

$$\begin{aligned}
n_p \dots &= \sqcup \llbracket (n_f \dots) (n_a \dots) \dots \rrbracket & n_{fe} &= \frac{\prod (n_p \dots)}{\prod (n_f \dots)} \\
n_{ae} \dots &= \frac{\prod (n_p \dots)}{\prod (n_a \dots)} \dots & n_{ac} \dots &= \left( \prod (n_i \dots) \right) \dots
\end{aligned}$$

$$\begin{aligned}
& ((\text{array } (n_f \dots) \mathbf{v}_f \dots)^{(A \rightarrow ((A \tau_i (\text{shape } n_i \dots)) \dots) \tau_o) (\text{shape } n_f \dots)}) \\
& (\text{array } (n_f \dots n_i \dots) \mathbf{v}_a \dots)^{(A \tau_i (\text{shape } n_f \dots n_i \dots)) \dots}) \\
& \mapsto_{\text{map}} \\
& (\text{frame } (n_f \dots) \\
& \quad ((\text{array } () \mathbf{v}_f)^{(A \rightarrow ((A \tau_i (\text{shape } n_i \dots)) \dots) \tau_o) (\text{shape})}) \\
& \quad (\text{array } (n_i \dots) \mathbf{v}_c \dots)^{(A \tau_i (\text{shape } n_i \dots)) \dots} \tau_o \dots)
\end{aligned}$$

where

$$\begin{aligned}
n_c \dots &= \left( \prod n_i \dots \right) \dots \\
((\mathbf{v}_c \dots) \dots) \dots &= \text{Transpose} \llbracket \text{Split}_{n_c} \llbracket \mathbf{v}_a \dots \rrbracket \dots \rrbracket \\
\text{Length} \llbracket n_f \dots \rrbracket &> 0
\end{aligned}$$

$$\begin{aligned}
& ((\text{array } () (\lambda ((x \tau) \dots) e)) v^\tau \dots) \\
& \mapsto_{\beta} e[x \mapsto v^\tau, \dots]
\end{aligned}$$

$$\begin{aligned}
& (\text{t-app } (\text{array } (n \dots) (\text{T}\lambda ((x k) \dots) e) \dots) \tau \dots) \\
& \mapsto_{t\beta} (\text{frame } (n \dots) e[x \mapsto \tau, \dots] \dots)
\end{aligned}$$

$$\begin{aligned}
& (\text{i-app } (\text{array } (n \dots) (\text{I}\lambda ((x \gamma) \dots) e) \dots) \iota \dots) \\
& \mapsto_{i\beta} (\text{frame } (n \dots) e[x \mapsto \iota, \dots] \dots)
\end{aligned}$$

$$\begin{aligned}
& (\text{frame } (n \dots) (\text{array } (n' \dots) \mathbf{v} \dots) \dots)^{(A \tau (\text{shape } n \dots n' \dots))} \\
& \mapsto_{\text{collapse}} (\text{array } (n \dots n' \dots) \text{Concat} \llbracket (\mathbf{v} \dots) \dots \rrbracket \rrbracket)^{(A \tau (\text{shape } n \dots n' \dots))}
\end{aligned}$$

$$\begin{aligned}
& (\text{unbox } (x_i \dots x_e (\text{array } (n_s \dots) (\text{box } \iota \dots v \tau) \dots)) e) \\
& \mapsto_{\text{unbox}} (\text{frame } (n_s \dots) e[x_i \mapsto \iota, \dots, x_e \mapsto v])
\end{aligned}$$

Figure 4.11: Dynamic semantics for Remora

## 4.4 TYPE SOUNDNESS

The value of a type-soundness theorem for Remora is not only assurance that well-typed programs do not suffer from shape-mismatching errors. It also ensures that the types ascribed to program terms accurately describe the shapes of the data those terms compute. That is the guarantee that justifies a compiler's use of the type system as a static analysis for array shape.

With supporting lemmas such as canonical forms and substitution already taken care of, we now establish progress and preservation lemmas. Since we have not committed to a collection of primitive operators that are all total functions, the progress lemma acknowledges the possibility of non-shape errors, such as division by zero. However, we do assume that any value returned by a primitive operator inhabits that operator's output type.

**Lemma 4.4.1** (Progress). *Given an expression  $e$  such that  $\cdot; \cdot \vdash e : \tau$ , one of the following holds:*

- $e$  is a value  $v$
- There exists  $e'$  such that  $e \mapsto e'$
- $e$  is  $\mathbb{W}[(\text{array } () \ \mathfrak{d}) \ v \ \dots]$  where  $\mathfrak{d}$  is a partial function applied to appropriately typed values outside its domain.

*Proof sketch.* We use induction on the derivation of  $\cdot; \cdot \vdash e : \tau$ . We consider only cases for typing rules which apply to expressions (as opposed to atoms). Since we do not reduce under a binder, our assumed type derivation ensures that the reducible subexpression of  $e$  is also typable using an empty environment.

An array form which is not already a value must have some non-value atom. That atom must itself contain a non-value expression, with its own type derivation. So the induction hypothesis implies that it can take a reduction step or is a mis-applied primitive operator. Similar reasoning applies to frame forms: either we have a *collapse* redex, or some cell subexpression in the frame can make progress.

An unbox form can either make progress in the box position (via the induction hypothesis) or take an *unbox* step. Similarly, a type or index application can make progress in function position or take a  $t\beta$  or  $i\beta$  step.

The function-application case splits into subcases depending on whether the function and argument arrays are fully reduced and if so what their frame shapes are. If they are all value forms, we have all scalar frames (a  $\beta$  or  $\delta$  redex) or all identical non-scalar frames (a *map* redex), or non-identical prefix-compatible frames (a *lift* redex). Prefix-incompatible frames are ruled out by the type derivation.  $\square$



**Lemma 4.4.2** (Preservation). *Let  $\Theta, \Delta, \Gamma$  be a well-formed environment, i.e.,  $\Theta; \Delta \vdash \Gamma$ . If  $\Theta; \Delta; \Gamma \vdash e : \tau$  and  $e \mapsto e'$  then  $\Theta; \Delta; \Gamma \vdash e' : \tau$ .*

*Proof sketch.* We use induction on the derivation of  $\Theta; \Delta; \Gamma \vdash e : \tau$ . An array form which can take a reduction step must contain a reducible subexpression. Many typing rules give rise to subcases where the  $e$  itself is not a redex but contains some subexpression  $e_r$  which steps to  $e'_r$ . In these situations, the typing derivation for  $e_r$  is included in that for  $e$ , so replacing that sub-derivation with one for  $e'_r$  (deriving the same type, according to the induction hypothesis) produces a derivation of  $\Theta; \Delta; \Gamma \vdash e : \tau$ .

The remaining nontrivial subcases each correspond to particular reduction rules. As in proving Progress, the T:APP case is split into subcases based on the function and argument frames. When frames are non-identical but prefix-compatible, the resulting *lift* reduction produces an application form with the same principal frame and thus the same result type. When we have identical non-scalar frames, the *map* reduction produces a frame form whose frame shape is equal to the application form's principal frame and whose cell shape and atom type is the same as the function's return shape and atom type. This gives it a type equivalent to that of the *map* redex. With a scalar principal frame, we have a  $\delta$  redex (trivial) or  $\beta$  redex (follows from Lemma 4.2.15, preservation of types under term substitution). Reasoning for type- and index-application forms is similar (via Lemma 4.2.13 and Lemma 4.2.14 respectively). A reducible *unbox* form also substitutes a value in for a variable which is intended to have the same type, so Lemma 4.2.15 again ensures the result type is  $\tau$ .  $\square$

**Theorem 4.4.1** (Type soundness). *If  $\cdot; \cdot; \cdot \vdash e : \tau$ , then either  $e$  diverges,<sup>19</sup> there exists  $v$  such that  $e \mapsto^* v$  and  $\cdot; \cdot; \cdot \vdash v : \tau$ , or there exist partial function  $\mathfrak{v}$  and appropriately typed arguments  $v \dots$  such that  $e \mapsto^* \mathbb{V}[(\text{array } () \mathfrak{v}) v \dots]$  and  $v \dots$  are outside the domain of  $\mathfrak{v}$ .*

*Proof.* We argue coinductively using the sequence of reduction steps from  $e$ . For any well-typed  $e$ , Progress (Lemma 4.4.1) implies that either  $e$  has the form  $v$ ,  $e$  has the form  $\mathbb{V}[(\text{array } () \mathfrak{v}) v \dots]$ , or  $e \mapsto e'$ . In the first case, the reduction sequence terminates in a value, so we have  $e \mapsto^* v$ . Furthermore, Preservation (Lemma 4.4.2) implies that  $\cdot; \cdot; \cdot \vdash v : \tau$ . In the second case, the reduction sequence terminates in a mis-applied primitive operator. In the third case, the Preservation lemma implies that  $\cdot; \cdot; \cdot \vdash e' : \tau$ .  $\square$

<sup>19</sup> I conjecture that divergence is not possible because Remora effectively extends System F with more detailed types and an iteration construct bounded by the size of actual data. However, the statement of this theorem must account for the possibility of divergence because I have not proven that Remora is normalizing.



Part II

TYPE INFERENCE



## BACKGROUND

Remora’s type inference builds on three major lines of prior work, Handling implicit instantiation of polymorphism is based on bidirectional type checking, a form of local type inference discussed in Section 5.1. The general theme in bidirectional type inference is recognizing when partial knowledge about a term’s type is available from the immediate surroundings. One can thus turn a conventional typing judgment into algorithmic rules which may treat the ascribed type as either input or output depending on how much the program context reveals about a subterm’s type. Dependent typing brings in its own type inference issues, which are discussed in Section 5.2. The fundamental challenge for the restricted style of dependent typing used by Remora is integration with a solver for the theory of type indices. Reasoning about array shapes—Remora’s type indices—uses the first-order theory of sequences, whose logical presentation and related algorithmic results are described in Section 5.3.

## 5.1 LOCAL TYPE INFERENCE AND BIDIRECTIONAL TYPING

The general technique of bidirectional type checking existed as folklore for some time before Pierce and Turner published a pair of type inference systems based on bidirectional checking [79]. The core of such type inference methods is recognizing that in conventional typing judgments, certain rules would permit a type checking algorithm to identify the type ascribed in the conclusion simply by inspecting the term. We can tell that 5 has type `Int` without being told that we are looking for an `Int`. Other rules, like those for functions and sums, require some information about the whole term’s type in order to properly check subterms on whose types the whole term’s well-typedness depends. Type checking  $(\lambda (x) (+ x 1))$  requires type checking  $(+ x 1)$ , which in turn requires having a type for  $x$ .<sup>20</sup> Since type information might in different situations flow into or out of the type ascribed to some term, bidirectional type checking uses two separate judgment forms. The conventional type judgment form  $\Gamma \vdash e : \tau$  is split into two judgment forms,  $\Gamma \vdash e \Leftarrow \tau$  and  $\Gamma \vdash e \Rightarrow \tau$ , respectively called “checking” and “synthesis.” They represent whether in algorithmic terms the type is an input or an output. Checking is used when a goal type is known and can therefore guide checking of subterms. For example, a function can be checked against a known  $\rightarrow$  type as follows:

$$\frac{\Gamma, x : \tau_i \vdash e \Leftarrow \tau_o}{\Gamma \vdash (\lambda (x) e) \Leftarrow (\rightarrow \tau_i \tau_o)}$$

<sup>20</sup> Presentations of simply typed  $\lambda$ -calculus, such as in Pierce’s *Types and Programming Languages* [78], often require a type annotation on each formal parameter in order to make unidirectional typing rules algorithmic.

Synthesis is used when there is no particular goal in mind and the term’s type must be discovered by inspecting its subterms.

The two judgment forms are mutually inductively defined, and both might appear as premises in the same rule. To synthesize a type for function application, first *synthesize* a type for the term in function position, and then *check* the argument against the function input type. Two rules provide the ability to reverse direction. To turn a checking goal into a synthesis goal, a subsumption rule allows a term  $e$  to check at type  $\tau$  by synthesizing the type  $\tau'$  for it and then ensuring that  $\tau' \leq \tau$ . To turn a synthesis goal into a checking goal, an annotation rule allows the annotated term  $(: e \tau)$  to have the type  $\tau$  synthesized as long as  $e$  checks at type  $\tau$ . Pierce and Turner also demonstrate how bidirectional rules can be used to select type arguments for instantiating polymorphic functions in the presence of subtyping and bounded quantification using a constraint solver over the language of types.

The overall design of Remora’s type inference is based on the “Pfenning recipe” for bidirectional type checking, as described by Dunfield and Krishnaswami [21, 23]. As a general principle, the Pfenning recipe calls for checking types of introduction forms and synthesizing types of elimination forms. Choosing which judgment to use in the premises of a rule is driven by what information is known about the types involved. For example, injecting some term into a sum type—the introduction form for sums—calls for a checking rule, where the goal type would identify the summand type we are not using. Since the goal type also identifies the desired type for the term we’re injecting, the premise judgment should be to check the term at that type. To eliminate a sum, via `match`, we would have to synthesize a sum type for the value we are inspecting because even specifying the type of the end result for each branch of the `match` does not say enough about the type of the contents extracted from the sum in those branches.

Dunfield and Krishnaswami’s system for handling higher-rank polymorphism [22] adds some flexibility to the bidirectional checking process by introducing a separate class of variables to represent unresolved portions of a type. For example, with the unsolved type variable  $\hat{t}$  as a goal type, if we check that the literal `10` has type  $\hat{t}$ , we would conclude that  $\hat{t}$  stands for `Int`. This allows synthesis rules to succeed when only partial information about a term’s type is discernible, and checking rules can work with partial information, possibly filling in details about the unresolved portions of a type.

The same authors have worked on a follow-up system which encodes indexed types using generalized algebraic datatypes and existential types [20]. In such a system, the `List` type constructor takes a type-level natural number (defined using Peano-like type constructors `S` and `Z`) as one of its arguments. This then requires index-level operators to be implemented as type-level functions, which becomes ergonomically awkward if the

universe of indices is meant to have a rich equality theory. While `append` can be typed in GADT style as producing a list whose length is the sum of its inputs' lengths, this process only goes smoothly if the type-level `+` function recurs down the same argument as `append`. Otherwise, typing an `append`-like function requires an explicit invocation of the commutativity of `+`, represented as a function which transforms a type constructed with `n + m` into one constructed from `m + n`. Other equalities in the algebra of  $\{\mathcal{N}, +, 0\}$  would require similar equality proofs.

Concoction [76] addresses the task of proving equalities on GADT indices by extending OCaml's type language to include Coq's term language. Recurring on the argument on which `+` doesn't recur can still be addressed by invoking a `plus_comm` lemma, but the full flexibility of Coq proof scripts is also at the programmer's disposal. For a decidable index theory, a proof script could just invoke the corresponding decision procedure tactic (such as `omega` for Presburger arithmetic).

Boxy types [98] and its descendant FPH [99] also build on Pierce and Turner's bidirectional system but with a single judgment form describing both "directions." The abstract syntax tree for types used during inference permits a "boxed" subtree within a type, identifying variable instantiations which remain to be guessed. This is in contrast to the Dunfield-Krishnaswami style, where metavariables stand in for unresolved portions of types, with witnesses to be found during an eventual subtype check.

## 5.2 DEPENDENT TYPE INFERENCE

Dependent ML presents two versions of bidirectional rules. The first set of rules is meant as a declarative (*i.e.*, non-algorithmic) description of how code which uses dependent types implicitly may be rewritten to use them explicitly. Some rules, such as dependent product elimination, require some oracle to identify indices to use for instantiation:

$$\frac{\Gamma \vdash e \Rightarrow \Pi x. \tau \leftrightarrow e \quad \Gamma \vdash \iota}{\Gamma \vdash e \Rightarrow \tau[x \mapsto \iota] \leftrightarrow e[x]}$$

If we synthesize a `Pi` type for `e`, then we can also synthesize any instantiation of that type with a well-formed index `ι`. This rule appearing at some position in a type derivation does not have access to enough information about *which* instantiation is actually needed, so the choice of index must be left to the oracle.

The second set of rules algorithmically generates a constraint formula over type indices. That formula can then be passed off to a standalone constraint solver. If the formula is true, then the program is well-typed. The constraint solver serves the role of the index-selection oracle, though

the constraint-based bidirectional rules do not actually generate an elaborated program. The dependent product elimination rule is quite similar:

$$\frac{\Gamma \vdash e \Rightarrow \Pi x_p. \tau \equiv \exists [\hat{x}_i \dots]. \Phi}{\Gamma \vdash e \Rightarrow \tau[x_p \mapsto \hat{x}_p] \equiv \exists [\hat{x}_i \dots, \hat{x}_p]. \Phi}$$

Rather than nondeterministically choosing an index, this rule marks  $\hat{x}_p$  as a new existential variable which the constraint solver must resolve. The underlying formula  $\Phi$  is untouched by this rule, but other rules can introduce connectives or equalities on types. For example, the product introduction rule generates the conjunction of the two constraints generated for the pair's elements, and a rule for checking application of `cons` equates a synthesized length argument for the `List` type constructor with the goal type's length argument.

There is still some disconnect between Dependent ML's constraint generation and elaboration, which makes Dependent ML's strategy ill-suited for Remora. Since the constraint solver is only asked to sign off on whether suitable type indices exist, it does not generate the actual index-level code for them.

In some cases, the suitable type indices may not be definable within the index language itself. Consider the `pairoff` function in Standard ML, which splits any even-length list into a list of consecutive pairs of elements.

```
fun pairoff [] = []
  | pairoff (x::y::more) = (x, y)::(pairoff more);;
```

In Dependent ML, it could be given the type

$$\Pi n. \forall t. (t) \text{List}[n+n] \rightarrow (t * t) \text{List}[n]$$

That is, for any length  $n$  and any element type  $t$ , this function transforms a list containing  $2n$  of these  $t$ s into a list containing  $n$  pairs of  $t$ s. However, a fully elaborated form would require giving the recursive call an index argument, which must be equal to  $n-1$ . The elaborated form of the pattern `(x::y::more)` will bind index variables equal to  $n+n-1$  and  $n+n-2$ , but the index language must be augmented with subtraction or division, in order to be able to express  $n-1$  itself.

The constraints generated by Dependent ML may also have deeply nested quantifier alternation, depending on the structure of the source program. Even for Presburger arithmetic, which is often used in Dependent ML examples, quantifier alternation makes formulae expensive to solve (triple exponential time in the worst case, as opposed to merely doubly exponential when the number of quantifier alternations is bounded [74, 84]). Quantifier alternation presents more difficulty for Remora, due to its more complex index language.



Besides Dependent ML, there have been other efforts to build a practical language with dependent types. In PIE [97], the type-index language mirrors the term language, but a phase distinction prohibits term variables from being referenced within types. Soundness also relies on an effect system to ensure no effects happen within a type index. The type inference machinery is a bidirectional system inspired by Pierce and Turner [79], but how the technique was adapted for PIE is not described in detail.

Elaboration in Idris [9] is performed via tactic-based proof search, with implicit arguments at a function call resolved via unification. The proof search is a heavier hammer than necessary for Dependent ML, doing extra work to handle additional language features such as identifying type-class constraints and then choosing appropriate type-class instances. Unification alone is also ill-suited for index arguments in Dependent ML, where the universe of type indices is meant to have its own nontrivial equality theory.

Liquid types [86] attach a logical qualifier to base data types, and functions' types may be refined to demand data qualified in a certain way. A naïvely generated whole-program constraint is repeatedly updated until either a scope-respecting solution is apparent or contradiction leads to a dead end. In a related method, Unno and Kobayashi [95] use Craig interpolation [15] to infer possible specifications for a function. Based on its definition and uses, the system searches for formulae that both are implied by the function's definition and themselves imply the success of assert statements which mention the function. The possibility of interpolation is a property of first-order logic, rather than any particular theory, so generating *definable* predicates works as a method for iteratively refining the description of a function's behavior until either a satisfied specification can be found or an input which leads to assertion failure can be generated. The fact that an interpolant will only reference symbols used in both formulae “between” which it sits ensures that candidate specifications for a function only mention its inputs and output.

Neither liquid types nor interpolation-based inference is quite suited to our purposes, since we intend to make more detailed knowledge about program data available for guiding compilation decisions. We wish to know not just whether a program will go wrong and whether functions' arguments meet their effective preconditions, but what particular aspects of these inputs make them acceptable.

### 5.3 THEORY OF SEQUENCES

Type indices, given in program syntax as  $\iota$ , represent individual dimensions, taken from  $\mathbb{N}$ , and array shapes, taken from the free monoid on  $\mathbb{N}$ . The theory of the free monoid on  $\mathbb{N}$  includes as axioms the associativity

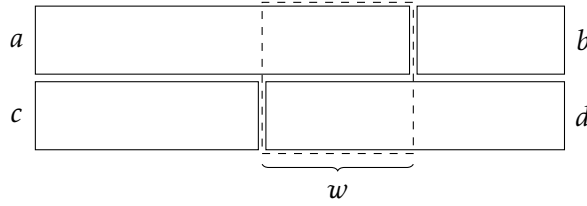


Figure 5.1: Overlap axiom, visualized:  $w$  is the overlapping portion of  $a$  and  $d$ .

of adding naturals and appending sequences as well as unique identity elements for addition (zero) and appending (the scalar shape,  $\square$ ):

$$0 + i = i + 0 = i$$

$$(i + j) + k = i + (j + k)$$

$$\square ++ a = a ++ \square = a$$

$$(a ++ b) ++ c = a ++ (b ++ c)$$

As the *free* monoid, it also follows an equidivisibility rule which states that if two uses of the append operator give the same result, there is some completing subsequence, representing the overlap between each use's larger argument (demonstrated in Figure 5.1):

$$a ++ b = c ++ d \implies$$

$$\exists w. (a ++ w = c \wedge w ++ d = b) \vee (c ++ w = a \wedge w ++ b = d)$$

A free monoid (on any set of generators) also has a homomorphism to the monoid formed by  $\mathbb{N}$  under addition, with the property that only the free monoid's identity element can be mapped to 0. This can be axiomatized with one additional function symbol  $L$ :

$$L(a) = 0 \implies a = \square$$

$$L(a ++ b) = L(a) + L(b)$$

The length function is a suitable candidate for  $L$ , but the most general requirement is that  $L$  take a weighted count of elements in a sequence, where every generator of the monoid is assigned a non-zero weight. In the free monoid on  $\mathbb{N}$ , this includes functions such as summing all numbers in a sequence or adding the sequence's length to its population of even numbers.

Using equidivisibility and the homomorphism to the additive monoid  $\mathbb{N}$ , we can define a partial operator  $\dot{-}$  for prefix subtraction:  $a \dot{-} b = c$  iff  $b ++ c = a$ . For example,  $[3, 4, 5, 6] \dot{-} [3, 4] = [5, 6]$ , whereas  $[3, 4, 5, 6] \dot{-} [4]$  is undefined. It is also possible to define suffix subtraction in a similar manner.

The full theory of sequences is undecidable, as proven by Durnev [25] and Marchenkov [61]. Later work by Durnev [24] tightened the result

to the  $\forall\exists^3(\vee\wedge)$  fragment, *i.e.*, a single universal quantifier and three existential quantifiers with only conjunction and disjunction is enough to be undecidable. Durnev shows that a predicate for valid start-to-finish execution traces of a particular Turing-complete abstract machine can be encoded in the  $\forall\exists^3(\vee\wedge)$  fragment, so this fragment is powerful enough to express halting. Since type inference for Remora treats bound shapes as universal variables and shape arguments to solve as existential variables, a naïve inference strategy would run up against this boundary. Although the mixed quantifier prefix is necessary, we can avoid asking a constraint solver to deal with disjunction.

The first decision procedure for the existential fragment of the theory of strings over a finite alphabet was presented by Makanin [59] and later simplified by Jaffar [45] and Gutiérrez [36]. Solving a string equation gives an assignment for the variables appearing in it, which in turn identifies the entire string denoted by both sides of the equation. Strictly speaking, Makanin’s algorithm handles the existential fragment of the theory of a free *semigroup*. A semigroup is associative, like a monoid, but may not have an identity element. The free semigroup—with no nontrivial identities—therefore does not have one. This is the algebra of *nonempty* sequences, and Makanin’s search procedure relies on the assumption that any named portion of an unknown string must be nonempty. In order to use Makanin’s algorithm for equations in a free monoid instead of a free semigroup, we must identify a subset of existential variables to assume represent the empty sequence and drop them from the equation. The algorithm is described in more detail in Chapter 7.

Makanin uses the *exponent of periodicity*, the number of consecutive repetitions of a substring in some longer string, to bound the depth of the search tree. As such, arguments deriving complexity bounds for Makanin’s algorithm focus on bounding the exponent of periodicity of the string denoted by each side of the equation, which in turn can be used to derive a bound on the depth of the search tree. Makanin initially gave an upper bound on the exponent  $s(n)$  for an initial equation of length  $n$  as  $2 + (6n)^{2^{(2n^4)}}$ .

The search process itself uses a more detailed representation of an equation, which assigns variables and constants to ranges between abstract boundaries within the equation’s full solution.<sup>21</sup> Given the number of bases  $x$  and the number of still possibly distinct boundaries  $y$ , the search depth  $d(x, y)$  is bounded by  $y * (s(2 * (x + 2y) * (y + 2)) + 1)$ . Kościelski and Pacholski tightened the upper bound on the exponent of periodicity to  $s(n) \leq 2^{1.07n}$  [55], leading to a doubly exponential nondeterministic time bound (stated as  $O(2^{2^{cn}})$ , for a constant  $c$ ). While this result only suffices to establish a triply exponential deterministic time bound, further work by Gutiérrez showed an exponential space bound [35]. By Savitch’s theorem, this exponential bound applies to both

<sup>21</sup> This portion of the cost analysis is gathered together neatly by Jaffar [45], though the original derivation of part of it is due to prior authors.

nondeterministic and deterministic machines [88], implying a doubly exponential deterministic time cost.

Despite the extremely heavy worst-case cost, the shape constraints arising from typical rank-polymorphic code avoid the algorithm’s worst case. Deep search is only realized by Makanin’s algorithm in cases where different occurrences of some variable represent overlapping sections of the denoted string, which does not happen with frame shapes. Wide branching requires having many variables on one side of an equation, as the branching corresponds to choosing how to align the boundaries of regions those variables represent. When there is only one variable, Makanin’s algorithm degrades into peeling generators off of the ends until either a conflict is discovered (and we have found a contradiction) or the variable’s region is isolated (and we have found a solution).

Karhumäki *et al* explored the problem of which properties of sequences can be stated by an equation [51]. Of particular importance, within the existential fragment of the theory of sequences with a finite, non-singleton generator set,<sup>22</sup> the conjunction, disjunction, and negation connectives applied to sequence equations can be rewritten as a single sequence equation, possibly introducing fresh variables. Thus any quantifier-free formula can be queried for satisfiability by condensing it to a single equation. The set of languages expressible in the existential fragment is also closed under finite intersection, finite union, Kleene star, concatenation, reverse, and cyclic rotation. Despite these capabilities, some regular languages, such as  $(a|b)^*$  are not definable—trying to use a disjunction like  $x = a \vee x = b$  still requires committing to a specific choice which will be used for *every* repetition of  $(a|b)$ . Disjunction only allows a bounded number of choices, whereas using Kleene star around a union of subterms in a regular expression corresponds to an unbounded number of choices.

Plandowski described an algorithm for solving string equations which works by exploring a graph of equations reachable via satisfiability-preserving transformations [80]. A constant can have all occurrences replaced by a chosen term, a variable in the equation can have a chosen term inserted immediately prior to all its occurrences, and a fresh variable can replace some (or all) occurrences of a chosen subterm in the equation. Following the above rules, a nondeterministic search for a transformation path from a trivial equation  $a = a$  to the original satisfiability query has polynomially bounded space cost. That the transformations preserve satisfiability implies the decision procedure is sound, but the proof of completeness (*i.e.*, that *all* satisfiable equations are reachable from  $a = a$ ) is more involved.

Key to Plandowski’s reduction in asymptotic cost is a more compact representation of intermediate steps in transforming a string equation. The possibly exponential blowup in the size of a solution, compared to the size of the original equation, comes from consecutive repetition

<sup>22</sup> The conjunction and disjunction constructions require selecting two distinct generators, and the negation construction requires enumerating the full set of generators.

of smaller substrings (recall, Makanin’s original termination argument arises from bounding that repetition). Plandowski and Rytter showed that this repetition provides enough of the blowup in solution size that conventional dictionary-based compression asymptotically shrinks the solutions [82]. Plandowski therefore includes exponentiation in the representation of equations and ensures a cubic bound on the size of rewritten forms of the original equation.

Follow up work by Plandowski extends the solution from determining whether an equation is satisfiable to whether the solution set is finite and characterizing maximal exponents of periodicity of solutions to the equation [81]. This extended algorithm still runs in polynomial space.

Jež’s “recompression” algorithm uses a similar strategy of compressing the expression by allowing exponentiation, but Jež’s representation is based on introducing fresh monoid generators which stand for sequences of pre-existing generators, rather than writing numeric exponents in the equation itself [47]. The algorithm is able to produce a description of the full solution set, rather than nondeterministically choosing a single solution, and Jež argues that local decision-making in this algorithm means there is more hinting available to guide the nondeterministic choices essential to equation-transformation search algorithms. Further development of the recompression algorithm achieves a nondeterministic linear space bound [48], corresponding via Savitch’s theorem to quadratic space on a deterministic machine.

A significant difference between Makanin-style boundary alignment search and later algorithms based on searching for sequences of transformations between equations is that the more recent algorithms rely on recognizing which monoid generators (*i.e.*, sequence element symbols) are equal to each other during the search process, whereas Makanin’s algorithm can be tweaked to allow such checks to be delayed until a leaf of the search tree is reached. In the case of type inference for Remora, the equality relation on generators may be uncertain because they are given as terms of Presburger arithmetic rather than simply symbols or numeric literals.



## LOCALLY INFERRING DEPENDENT TYPES

---

The explicitly typed language presented in Chapter 4 has implicit frame polymorphism in function application, but cell polymorphism is explicit. Quantifying over and instantiating portions of cell shape using  $\Gamma\lambda$ ,  $I\lambda$ ,  $t$ -app, and  $i$ -app is an excessive annotation burden for a programmer and typically exceeds the boilerplate eliminated by implicit frame polymorphism. The primary goal for a type inference strategy for Remora must be to allow the programmer to call cell-polymorphic functions like `mean` and `reduce`, without stating how they are instantiated.

Automatically generalizing Remora code to take a polymorphic type, in the style of Hindley-Milner type inference [39], is infeasible because Remora lacks principal types. In addition to the usual limitations arising from including parametric polymorphism over atom types (which might themselves be polymorphic functions), even the  $\forall$ -free fragment of Remora includes functions which have no most general type. Consider defining a function `app+` as follows:

```
(λ ((x 1) (y 1))
  (+ (append x y)
     [1 2 3 4 5]))
```

The length of `x` can be anything from  $\{0, \dots, 5\}$ , which then determines the exact length of `y`. So the function has six possible monomorphic types, and no  $\Pi$  type includes all of and only those six as its possible instantiations. Lacking principal types, a more robust option is to require the programmer to specify when a function is meant to be polymorphic. Code comprehensibility is often mentioned as mitigating the cost of requiring polymorphic functions to be so annotated, including by Pierce and Turner’s paper which forms the foundation for much work in bidirectional type checking [79]. Including a type annotation explains to other programmers how a polymorphic function is meant to be used without requiring them to reconstruct its type themselves.

In both the untyped and typed forms of Remora, a function’s formal parameters must be marked with a description of the expected cells, which determines how applying the function to aggregate arguments will break those arguments into cells. In the untyped version, those annotations are numeric cell ranks, or `all` for a function which is polymorphic in cell rank. The explicitly typed form replaces these rank annotations—both numeric and `all`—with type annotations.

For a human-facing implicitly typed language, rank annotations are preferable. First, they are more concise, reducing the programmer’s

annotation burden. The prevalence of reranking (recall, a form of  $\eta$ -expansion) means that describing cell shapes is common. Second, they retain some flexibility which would be lost by specifying a particular cell type. For example, specifying cell rank instead of cell type allows `app+` to be written without specifying the lengths of `x` and `y`. It would be up to type inference to examine the actual arguments passed to `app+` to determine those lengths. That is, one of many possible monomorphic types can be chosen without first identifying a principal polymorphic type. Automatic  $\Pi$ -generalization for `app+` is impossible but also turns out to be unnecessary. Viewing  $\Pi$ - and  $\forall$ -generalization as identifying a “weakest precondition” on some term’s type, a type inference algorithm is allowed to keep track of finer-grained preconditions than can be expressed using the type system itself.

Supporting rank annotations requires converting them into type annotations. This is fairly straightforward for some cases, such as `outer*`:

```
(λ ((x 0) (y 1))
  (* x y))
```

We have `x` whose shape must be scalar and `y` whose shape can be a vector of any length. Other cases are trickier, like `vec+`:

```
(λ ((x 1) (y 1))
  (+ x y))
```

Here, each argument is a vector which may have any length, but both must have the *same* length (otherwise the application of `+` is ill typed). Type inference must recognize that `x`’s and `y`’s lengths are related and ensure that when `vec+` is elaborated into a form with cell type annotations, the arguments’ annotations both use the same dimension. If we imagine the formal parameters to `vec+` being described as `(x [Int $x])` and `(y [Int $y])`, the call to `+` forces the frame shapes `[$x]` and `[$y]` to match, and `[$x] ≐ [$y] ⊢ $x ≐ $y`. In Remora’s type inference, these relationships between pieces of the implicit shapes arising from rank annotations are to be discovered by a constraint solver for array shapes.

The key design points for Remora’s type inference are

1. Elaborate programs to explicitly typed code
2. Explicitly introduce and implicitly eliminate cell polymorphism
3. Discover dependence between portions of rank-annotated arguments’ types

Prior work on bidirectional type checking offers a starting point for automatically instantiating cell-polymorphic functions. The flexibility of Dunfield and Krishnaswami’s bidirectional type system[22] arises partly from the ability to delay selecting unsolved portions of types. So the “checking” judgment can succeed without a fully specified goal



type, and the “synthesis” judgment has the ability to return only partial information about a term’s type. Some new machinery is needed to grow from implicitly instantiating  $\forall$  types to explicit instantiation for both  $\forall$  types and DML-style  $\Pi$  types. Elaboration itself requires constructing instantiation witnesses, and type indices come with a richer theory of equality.

First, emitting an elaborated program requires that each judgment form include either the elaborated term or, for subtyping, code that can be wrapped around the lower-typed term to form the higher-typed term. Second, the subtyping judgment must add a new rule to require equality of dependent type indices. Third, the solver used for checking whether shapes can be equated must be able to point out an equivalence relation on dimensions which would make a shape equation solvable.

The first two requirements are handled within the bidirectional typing and subtyping judgments themselves, which are presented in this chapter. The structure of the solver needed to accommodate rank annotations on formal parameters is discussed in Chapter 7.

The Redex model from Chapter 4<sup>23</sup> also includes bidirectional typing rules for an implicitly typed variant of the core language. The implicitly typed language is defined in `implicit-lang.rkt`, the bidirectional rules in `bidirectional.rkt`, and an adapter layer for linking to the shape theory solver in `makanin-wrapper.rkt`. The bidirectional rules rely on some utility judgments for identifying array types, atom types, shapes, and dimensions, which is found in `well-formedness.rkt`. In order to allow Redex code to handle a mix of implicit and explicit Remora code, a combined language is defined in `elab-lang.rkt`.

<sup>23</sup> <https://github.com/jrslepak/Revised-Remora>

## 6.1 SYNTAX

The grammar for Implicit Remora, a variant without explicit instantiation of type- and index-polymorphic functions, is given in Figure 6.1. As in the explicitly typed grammar (Figure 4.1), *fraktur* typeface is used in nonterminals which stand for fragments—atoms, atom types, and dimensions—and *italic* typeface for composites—expressions, array types, and shapes. An overline is used to distinguish syntactic classes in implicitly typed Remora which differ from the explicitly typed versions.

Beyond eliding type and index application, as is typical in bidirectional systems, a general-purpose annotation form is provided, allowing the programmer to specify when a function is meant to be polymorphic and disambiguate cases where a term’s type is uncertain. To move closer to a concise surface syntax, this variant of Remora uses distinct classes of variables for terms ( $a$ , with no sigil), array types ( $\rho$ , prefixed with  $*$ ), atom types ( $\alpha$ , prefixed with  $\&$ ), shapes ( $\sigma$ , prefixed with  $@$ ), and dimensions ( $\delta$ , prefixed with  $\$$ ). This facilitates syntactic sugar analogous to that used in untyped code. An atom type appearing where an array type is

expected can be implicitly converted to a scalar array containing that atom, and a dimension appearing where a shape is expected can be implicitly converted to a vector shape. For purposes of type inference, it is necessary to distinguish monomorphic types, as they are the only permitted meanings for type variables.

The environment structure used for bidirectional type checking is described in Figure 6.2. Some environment manipulations performed in the bidirectional rules—particularly those which make reference to which variables were brought into scope when—are easier to state using an environment with a single namespace rather than using separate type, kind, and sort environments. Having separate classes of variables means individual environment entries do not need to specify the kind or sort of a type or index variable. During elaboration, an environment will sometimes be treated as a substitution, written as  $\bar{\Gamma}[\cdot]$ , replacing any resolved type or index variables with their solutions.

Three environment-manipulating metafunctions are used to connect the bidirectional type derivations with type derivations for explicitly typed code:  $TB[\cdot]$ ,  $KB[\cdot]$ , and  $SB[\cdot]$  extract variable-to-type, variable-to-kind, and variable-to-sort bindings from a bidirectional environment to produce a typing, kinding, and sorting environment respectively.

Tracking dependence among the type indices which must be inserted into the elaborated program is more difficult due to Remora’s particular index theory. Predicative System F types (as in past work [22]) permit gradually resolving a type by picking a type constructor and then seeking solutions for that type constructor’s arguments. On the other hand, most objects in Presburger arithmetic’s universe cannot be uniquely characterized by a constructor and its arguments. Any type equal to  $(\rightarrow \hat{x} \hat{x})$  must be constructed by  $\rightarrow$  and must have its input type equal to its output type. However  $(+ \hat{m} \hat{m})$  could be equal to  $(+ 5 \hat{n})$ , for many possible  $\hat{n}$ , not only 5. Accommodating such partial information about dimensions and propagating it through an implicitly typed program can complicate the environment somewhat.

For example, an existential dimension variable  $\hat{k}$  might be discovered in one subexpression to be a multiple of two and in another to be a multiple of three. This might come from adding a vector of length  $\hat{k}$  to vectors of length  $2(\hat{m})$  and  $3(\hat{n})$ . Suppose also that the scopes of  $\hat{m}$  and  $\hat{n}$  are nested within that of  $\hat{k}$ . That is, a solution for  $\hat{k}$  cannot mention  $\hat{m}$  and  $\hat{n}$ , and type inference must still remember the requirements they imposed on  $\hat{k}$  after they go out of scope.

If we had only  $\hat{m}$  to deal with, we might generate a fresh “placeholder” variable  $\widehat{\$k/2}$ , resolve  $\hat{k}$  as  $(+ \$k/2 \widehat{\$k/2})$ , and resolve  $\hat{m}$  as  $\widehat{\$k/2}$ . This strategy runs into trouble when we include  $\hat{n}$  and possibly more existential dimension variables. To handle both  $\hat{m}$  and  $\hat{n}$ , we would need a  $\widehat{\$k/6}$  placeholder because the environment cannot resolve  $\hat{k}$  as both  $(+ \$k/2 \widehat{\$k/2})$  and  $(+ \$k/3 \widehat{\$k/3} \widehat{\$k/3})$  (an existential type or index vari-

$\bar{e} \in \overline{Expr} ::=$	$\bar{a}$	<i>Implicitly typed expressions</i>
	(array ( $n \dots$ ) $\bar{a} \dots$ )	<i>Term variable</i>
	(frame ( $n \dots$ ) $\bar{e} \dots$ )	<i>Array</i>
	( $\bar{e}_f \bar{e}_a \dots$ )	<i>Frame</i>
	(unbox ( $x_i \dots x_e \bar{e}_s$ ) $\bar{e}_b$ )	<i>Term application</i>
	( $: \bar{e}_a T$ )	<i>Let-binding box contents</i>
$\bar{a} \in \overline{Atom} ::=$	$\mathbf{b}$	<i>Type annotation</i>
	$\mathbf{o}$	<i>Implicitly typed atoms</i>
	( $\lambda ((a \zeta) \dots) \bar{e}$ )	<i>Base value</i>
	(box $\bar{i} \dots e$ )	<i>Primitive operator</i>
	( $: \bar{e}_a \Upsilon$ )	<i>Term abstraction</i>
$\zeta ::= n \mid \text{all} \mid \tau$		<i>Boxed array</i>
$\bar{\tau} \in \overline{Type} ::= T \mid \Upsilon$		<i>Type annotation</i>
$T \in \overline{ArrayT} ::=$	(A $\Upsilon I$ )	<i>Cell specifier</i>
	$\rho$	<i>Array types</i>
$\Upsilon \in \overline{AtomT} ::=$	B	<i>Atom types</i>
	( $\rightarrow (T \dots) T'$ )	<i>Base type</i>
	( $\forall (\mathcal{X} \dots) \tau$ )	<i>Function</i>
	( $\Pi (\mathcal{S} \dots) \tau$ )	<i>Universal</i>
	( $\Sigma (\mathcal{S} \dots) \tau$ )	<i>Dependent product</i>
	$\alpha$	<i>Dependent sum</i>
$\bar{i} \in \overline{Idx} ::= I \mid \text{dim}$		<i>Atom type variable</i>
$I \in \overline{Shp} ::=$	(shape $\mathbf{I} \dots$ )	<i>Indices</i>
	( $++ I \dots$ )	<i>Shape indices</i>
	$\sigma$	<i>Sequence of dimensions</i>
$I \in \overline{Dim} ::=$	$n$	<i>Appending shapes</i>
	( $+ \text{dim} \dots$ )	<i>Shape variable</i>
	$\delta$	<i>Dimension indices</i>
$x ::= \mathcal{S} \mid \mathcal{X}$		<i>Natural number</i>
$\mathcal{X} ::= \rho \mid \alpha$		<i>Adding dimensions</i>
$\mathcal{S} ::= \sigma \mid \delta$		<i>Dimension variable</i>
$\bar{\mu} \in \overline{Monotype} ::= T \mid \Upsilon$		<i>Variables</i>
$M \in \overline{ArrayM} ::= (A \mathfrak{M} I) \mid \rho$		<i>Type variables</i>
$\mathfrak{M} \in \overline{AtomM} ::= B \mid (\rightarrow (M \dots) M') \mid \alpha$		<i>Index variables</i>

Figure 6.1: Grammar for implicitly typed Remora

$\gamma ::=$	<i>Environment entries</i>
$a : T$	<i>Term variable bound at array type</i>
$\mathcal{X}$	<i>Bound type variable</i>
$\hat{\mathcal{X}}$	<i>Unresolved type</i>
$\hat{\mathcal{X}} \mapsto \tau$	<i>Resolved type</i>
$\mathcal{S}$	<i>Bound index variable</i>
$\hat{\mathcal{S}}$	<i>Unresolved index</i>
$\hat{\mathcal{S}} \mapsto \iota$	<i>Resolved index</i>
$\blacktriangleright_x$	<i>Scope marker</i>
$\bar{\Gamma} ::= \gamma, \dots$	<i>Inference environment</i>
$\Phi ::= I \doteq I', \dots$	<i>Dimension equalities</i>

Figure 6.2: Environment structure for implicitly typed Remora

able gets only one solution in  $\bar{\Gamma}$ ). If we had already substituted  $\widehat{\$k/2}$  into the elaborated program, there would be more work needed when  $\widehat{\$k/6}$  is introduced. Discovering later that  $\widehat{\$k}$  must be equal to  $(+\widehat{\$p}1)$  forces even more placeholders into the environment.

Instead of expanding the variable-scope environment  $\bar{\Gamma}$  in this manner, Remora’s type inference algorithm adds an “archive”  $\Phi$  containing the discovered equalities on dimensions. An existential dimension variable in  $\bar{\Gamma}$  will still have a solution noted, since that is needed for elaboration, but the accumulation of constraints is collected in  $\Phi$ . The archive ensures that constraints involving variables which have gone out of scope are not forgotten. As a demonstrative example, a program of the following form places contradictory constraints on the length of  $x$ :

```
(λ ((x 1))
  (f ((λ ((y 1))
      (+ x (append y y)))
    some-computation)
    ((λ ((z 1))
      (+ x (append z z [1])))
     other-computation)))
```

We need to make sure that the use of  $y$  which requires  $x$  to have even length is remembered when we encounter the use of  $z$  which requires  $x$  to have odd length.

## 6.2 SOLVER INVOCATION

The judgment forms defined later in this chapter require access to a solver for the theory of shapes. The interface to this solver is phrased as

$$\bar{\Gamma}; \Phi \models I \doteq I' \Rightarrow \bar{\Gamma}'; \Phi'$$

This means that given the input environment  $\bar{\Gamma}$  and equation archive  $\Phi$ , the solver equates shapes  $I$  and  $I'$  (which may contain existential index variables) by constructing the output environment  $\bar{\Gamma}'$  and archive  $\Phi'$ .

**Proposition 6.2.1** (Solver specification). *Given  $\bar{\Gamma}; \Phi \models I \doteq I' \Rightarrow \bar{\Gamma}'; \Phi'$ , then all of the following hold:*

- $\bar{\Gamma}'$  preserves all entries of  $\bar{\Gamma}$ , except for solving some existential variables and introducing new existential variables
- $\Phi'$  is  $\Phi$  augmented with new equalities which entail the minimal equivalence relation on dimensions appearing in  $I$  and  $I'$  for some alignment of existential shape variables' boundaries.
- New entries added to  $\bar{\Gamma}'$ , when taken as equations on dimensions and shapes, are consistent with  $\Phi'$ .
- $\Phi' \models \bar{\Gamma}' \llbracket I \rrbracket \doteq \bar{\Gamma}' \llbracket I' \rrbracket$

A description of how to modify Makanin's string equation algorithm to satisfy the above specification is given in Chapter 7.

An equation on appended shapes may have multiple solutions corresponding to different ways that variables in the shape equation might be aligned. For example, the equation

$$(\text{shape } \hat{s}_a \hat{s}_b \hat{s}_c \hat{s}_d \hat{s}_e) \doteq ( ++ \hat{s} \hat{t} \hat{s})$$

gives three possibilities for the length of  $\hat{s}$ : 0, 1, or 2. Each alignment gives rise to a particular minimal equivalence relation—an equivalence class identifies a set of dimensions that are aligned at the same position within some shape variable. If we take 0 as the length of  $\hat{s}$  in the above equation, there is no need to equate any of the existential dimension variables. Choosing a length of 1 means that the first occurrence of  $\hat{s}$  covers  $\hat{s}_a$ , and the second covers  $\hat{s}_e$ . So the minimal equivalence relation for that alignment equates  $\hat{s}_a$  with  $\hat{s}_e$  but no other dimension variables. For a length of 2, we have  $\hat{s}$  equal to both  $(\text{shape } \hat{s}_a \hat{s}_b)$  and  $(\text{shape } \hat{s}_d \hat{s}_e)$ , so the solver should equate  $\hat{s}_a$  and  $\hat{s}_d$  and  $\hat{s}_b$  and  $\hat{s}_e$ . For an equation that allows multiple alignments, there can be multiple possible values for the solver outputs  $\bar{\Gamma}'$  and  $\Phi'$ .

Producing an unsatisfiable  $\Phi'$  indicates a dimension mismatch. Although an internally inconsistent equivalence relation for this particular equation means the chosen alignment is impossible, it is possible for

an internally consistent equivalence relation to be inconsistent with earlier choices. If an earlier shape equation solution included  $\hat{\$}_n = (+ \hat{\$}_m \hat{\$}_l)$  while the new one includes  $\hat{\$}_n = (+ 1 \hat{\$}_l \hat{\$}_l)$ , we have discovered conflicting constraints on  $\hat{\$}_n$ . A practical implementation may choose to proceed anyway. At the end of bidirectional inference, an output archive with multiple disjoint unsatisfiable cores can point to multiple parts of the program which warrant type error messages. The decision of when to filter out a solver result with an inconsistent output archive is beyond the scope of this work.

### 6.3 BIDIRECTIONAL JUDGMENT FORMS

Similar to Dunfield and Krishnaswami’s work [22], Remora uses the usual “synthesis” and “checking” judgments—corresponding to seeking to derive a type and already having a goal type—alongside a third “application” judgment. All three judgments update the environment as new information about unresolved type and index variables is discovered. The judgments themselves are given in Figures 6.3, 6.4, and 6.5. The three are defined mutually inductively. As is typical in bidirectional systems, a subsumption rule allows a goal type to be satisfied by a “lower,” *i.e.*, more polymorphic type, and an annotation rule allows the programmer to provide a hint as to what type ought to be synthesized for an expression or atom. The application judgment must also check arguments’ types against possibly incompletely resolved function input types.

#### 6.3.1 *Synthesis*

The synthesis judgment is written out as

$$\bar{\Gamma}; \Phi \vdash \bar{t} \Rightarrow \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow t$$

The environment structure includes both bindings ( $\bar{\Gamma}$ ) and collected restrictions on dimensions ( $\Phi$ ). The judgment states that in the given environment:

- The term  $\bar{t}$  is discovered to have type  $\tau$
- In making that discovery, the environment is updated to  $\bar{\Gamma}'; \Phi'$
- The implicitly typed  $\bar{t}$  elaborates to the explicitly typed  $t$

Of the synthesis rules, the most straightforward is the rule for variable references. Whichever type is given to  $x$  in the binding environment is the type synthesized for  $x$ , and there is no change to the environment from looking up a variable. A variable reference (absent a particular goal type) elaborates to just the original variable reference.

Following the Pfenning recipe, the elimination forms which require synthesis rules are unboxing and function application. In order to synthesize a type for an `unbox` form, we first must synthesize a type for  $\overline{e_s}$ , the array of boxes being destructed. One tempting next step would be to synthesize a type for  $e_b$ , the body of the `unbox`. However, we already know what type constructor we want to find. We need to get an array type and also be able to refer to its shape. So instead we check against a *partially specified* type:  $(A \widehat{\alpha}_b \widehat{\sigma}_b)$ , with fresh existential atom type variable  $\widehat{\alpha}_b$  and shape variable  $\widehat{\sigma}_b$ . We leave it to the checking derivation to resolve the existential variables. The final result's type uses  $I_s$  (the shape of  $\overline{e_s}$ ) as the frame shape and  $\widehat{\sigma}_b$ , (the shape of  $\overline{e_b}$ ) as the result cells' shape. There is no extra elaboration needed for `unbox` beyond what its subexpressions  $\overline{e_s}$  and  $\overline{e_b}$  require, and the only changes to the environment are those made while typing  $\overline{e_s}$  and  $\overline{e_b}$ .

The major elimination form in Remora is function application, and most of the work in synthesizing a type for an application is pushed into the application judgment. The role of the `SYN:APP` rule is to kick off that process and then report its end result. First, we need to synthesize a type for the function-position expression, but we do not insist on finding an `Arr` type containing `->` atoms. The application judgment handles instantiating polymorphism, so we may proceed using an array of  $\Pi$  or  $\forall$  atoms instead.

Other synthesis rules—`SYN:FN`, `SYN:ARRAY`, and `SYN:FRAME`—are optional according to the Pfenning recipe but still convenient for reducing the annotation burden.

A function has each formal parameter annotated with either a cell type or a cell rank. `SYN:FN` must handle the conversion from cell-rank annotations in implicit Remora to cell-type annotations in explicit Remora. The `NewVars` and `NewType` metafunctions are used for elaborating the cell specification.

`NewVars` behaves like the `new-vars` Racket procedure below. If the specification is a rank  $n$ , `NewVars` produces  $n$  fresh existential dimension variables to include in the environment for the function body. For `all`, it produces a single fresh existential shape variable. Both rank cases must also produce an existential atom type variable. If the cell specification is a type rather than a rank, `NewVars` produces an empty sequence.

```
(define (new-vars var spec)
  (match spec
    [(? natural? _)
     (cons
      (gensym (format "^&~a" var))
      (for/list ([i n])
        (gensym (format "^$~a~a" var i)))))
    ['all (list (gensym (format "^*~a" var)))]
    [(? type? _) '()])))
```

Once we have any fresh atom-type and index variables needed for the function’s formal parameters, *NewType*, following the procedure *new-type*, constructs the type we ascribe to a parameter. If we have only a rank specification, this will be a partially unresolved type. It is up to the premise—synthesizing a type for the function body—and lower parts of the derivation tree to discover further restrictions on the parameters’ underlying atom types and dimensions or shapes. When given a type specification instead of only a rank, we can use the type as is.

```
(define (new-type var spec)
  (match spec
    [(? natural? _)
     '(A ,(first (new-vars var spec))
          ,(rest (new-vars var spec)))]
    ['all (gensym (format "*~a" var))]
    [(? type? _) spec]))
```

Finally, to produce the argument cell type annotations in the elaborated output code, *ElabType* is needed to smooth over the difference between the core language used for the formalism, where type and index variables are annotated with a kind or sort, and implicit Remora’s syntax, where programmer-friendly shorthand for types rests on using different classes of variables for different kinds and sorts. The *ElabType* metafunction is a simple pass over a type adding kind and sort annotations to each type- and index-variable binding. In cases where only a type- or index-variable binding must be converted,  $[\cdot]$  stands for the type or index variable annotated with its appropriate kind or sort. For example,  $[\$q]$  is ( $\$q$  Dim), and  $[*m]$  is ( $*m$  Array).

SYN:ARRAY is more straightforward than the previous rules. Since an array form states its shape directly, synthesizing its type only requires synthesizing an atom type. Since arrays are homogeneous, we must check that all other atoms agree with the synthesized type. The SYN:FRAME rule is similar to SYN:ARRAY, but requires synthesizing a cell type rather than an atom type. Subtyping offers no flexibility about shapes, so all cells in a frame must have equatable shapes. However, success still depends on argument order. Rules like SYN:ARRAY and SYN:FRAME would benefit from computing a least upper bound on the atoms’ or cells’ types, but extending Remora’s language of types to form a lattice, in the style of MLsub [19], is left to future work. The only opportunity for dependence on argument order is if an array or frame contains atoms or cells with differing degrees of polymorphism. The common case of gathering first-order values into an aggregate structure leaves no room for different types to be synthesized for different elements.



$$\boxed{\bar{\Gamma}; \Phi \vdash \bar{t} \Rightarrow \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow t}$$

$$\frac{\bar{\Gamma}_0; \Phi_0 \vdash \bar{t} \Leftarrow \tau \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow t}{\bar{\Gamma}_0; \Phi_0 \vdash (: \bar{t} \tau) \Rightarrow \tau \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow t} \text{ SYN:ANNOT}$$

$$\frac{}{\bar{\Gamma}_0, x : \tau, \bar{\Gamma}_1; \Phi \vdash x \Rightarrow \tau \dashv \bar{\Gamma}_0, x : \tau, \bar{\Gamma}_1; \Phi \hookrightarrow x} \text{ SYN:VAR}$$

$$\frac{\begin{array}{c} \text{with fresh } \widehat{\alpha}_b, \widehat{\sigma}_b \\ \bar{\Gamma}_0 \Phi_0 \vdash \bar{e}_s \Rightarrow (A \text{ (box } x'_i \dots \tau_s) I_s) \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow e_s \\ \bar{\Gamma}_1, \widehat{\alpha}_b, \widehat{\sigma}_b, x_i \dots, x_e : \tau_s[x'_i \mapsto x_i, \dots]; \Phi_1 \vdash \bar{e}_b \Leftarrow (A \widehat{\alpha}_b \widehat{\sigma}_b) \\ \dashv \bar{\Gamma}_2, x_i \dots, \bar{\Gamma}_3; \Phi_2 \hookrightarrow e_b \\ KB \llbracket \bar{\Gamma}_3 \rrbracket; SB \llbracket \bar{\Gamma}_3 \rrbracket \vdash (A \Upsilon_b (++) I_s \widehat{\sigma}_b) :: \text{Array} \end{array}}{\bar{\Gamma}_0; \Phi_0 \vdash (\text{unbox } (x_i \dots x_e \bar{e}_s) \bar{e}_b) \Rightarrow (A \Upsilon_b (++) I_s \widehat{\sigma}_b) \dashv \bar{\Gamma}_2; \Phi_2 \hookrightarrow (\text{unbox } (x_i \dots x_e e_s) e_b)} \text{ SYN:UNBOX}$$

$$\frac{\begin{array}{c} \bar{\Gamma}_0; \Phi_0 \vdash \bar{e}_f \Rightarrow \tau_f \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow e_f \\ \bar{\Gamma}_1; \Phi_1 \vdash (e_f : \tau_f) \bullet [\bar{e}_a \dots] \Rightarrow \tau_r \dashv \bar{\Gamma}_2; \Phi_2 \hookrightarrow (e'_f e_a \dots) \end{array}}{\bar{\Gamma}_0; \Phi_0 \vdash (\bar{e}_f \bar{e}_a \dots) \Rightarrow \tau_r \dashv \bar{\Gamma}_2; \Phi_2 \hookrightarrow (e'_f e_a \dots)} \text{ SYN:APP}$$

$$\frac{\begin{array}{c} \bar{\Gamma}_0, \blacktriangleright_{x_f}, \text{NewVars} \llbracket x, \varsigma \rrbracket \dots, x : \tau_i \dots; \Phi_0 \vdash \bar{e} \Rightarrow \tau_o \\ \dashv \bar{\Gamma}_1, \blacktriangleright_{x_f}, \bar{\Gamma}_2; \Phi_1 \hookrightarrow e \\ \text{where } \tau_i \dots = \text{ElabType} \llbracket x, \varsigma \rrbracket \dots \\ \text{with fresh } x_f \end{array}}{\bar{\Gamma}_0; \Phi_0 \vdash (\lambda ((x \varsigma) \dots) \bar{e}) \Rightarrow \bar{\Gamma}_2 \llbracket (- \blacktriangleright (\tau_i \dots) \tau_o) \rrbracket \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \bar{\Gamma}_2 \llbracket (\lambda ((x \text{ElabType} \llbracket \tau_i \rrbracket) \dots) e) \rrbracket} \text{ SYN:FN}$$

$$\frac{\begin{array}{c} \bar{\Gamma}_0; \Phi_0 \vdash \bar{a} \Rightarrow \Upsilon \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow a \\ \bar{\Gamma}_1; \Phi_1 \vdash \bar{a}' \Leftarrow \Upsilon \dashv \bar{\Gamma}_m; \Phi_m \hookrightarrow a' \dots \\ \text{Length} \llbracket \bar{a} \dots \rrbracket = \prod n \dots \end{array}}{\bar{\Gamma}_0; \Phi_0 \vdash (\text{array } (n \dots) \bar{a} \bar{a}' \dots) \Rightarrow (A \Upsilon (\text{shape } n \dots)) \dashv \bar{\Gamma}_m; \Phi_m \hookrightarrow (\text{array } (n \dots) a a' \dots)} \text{ SYN:ARRAY}$$

$$\frac{\begin{array}{c} \bar{\Gamma}_0; \Phi_0 \vdash \bar{e} \Rightarrow (A \Upsilon \iota) \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow e \\ \bar{\Gamma}_1; \Phi_1 \vdash \bar{e}' \Leftarrow (A \Upsilon \iota) \dashv \bar{\Gamma}_m; \Phi_m \hookrightarrow e' \dots \end{array}}{\bar{\Gamma}_0; \Phi_0 \vdash (\text{frame } (n \dots) \bar{e} \bar{e}' \dots) \Rightarrow (A \Upsilon (++) (\text{shape } n \dots) \iota) \dashv \bar{\Gamma}_m; \Phi_m \hookrightarrow (\text{frame } (n \dots) e e' \dots)} \text{ SYN:FRAME}$$

Figure 6.3: Type synthesis judgment

### 6.3.2 Checking

The checking judgment is written as

$$\bar{\Gamma};\Phi \vdash \bar{t} \Leftarrow \tau \dashv \bar{\Gamma}';\Phi' \hookrightarrow t$$

The meaning is similar to that of synthesis, except that  $\tau$  here is a goal, *i.e.*, input to the judgment, rather than discovered by the judgment.

Analogous to the annotation rule for synthesis, the subsumption rule  $\text{CHK:SUB}$  allows a checking judgment for a term to be reached via a synthesis judgment for that same term. The elaborated code generated by synthesis might be, for example, a polymorphic function where checking had a monomorphic function as the goal type. The subtype judgment's output  $\mathbb{C}$  is an explicit Remora context (*i.e.*, term with a hole) which coerces terms from the low type  $\tau_l$  to the high type  $\tau_h$ . If we discovered that  $\bar{t}$  has type

```
(A (∀ (&e)
      (A (-> ((A &e (shape)))
              (A &e (shape))))
      (shape)))
(shape))
```

*i.e.*, a scalar containing a polymorphic function on scalars, the context  $(\text{t-app } \square \text{ Int})$  would coerce  $\bar{t}$  to have type

```
(A (-> ((A Int (shape)))
        (A Int (shape))))
(shape))
```

For functions, the requirement to include a cell specification turns the Pfenning recipe on its head.  $\text{CHK:FN}$  operates much like  $\text{SYN:FN}$ , except that introducing a goal type means there may be conflict between the goal's input types and the specified input cell ranks. Checking  $(\lambda ((x \ 1)) (+ \ x \ x))$  at type  $(-> ((A \text{Int} \ (\text{shape}))) (A \text{Int} \ (\text{shape})))$  should not succeed because this function requires vector input rather than scalar. The subtyping checks between the desired input types  $\tau_i \dots$  and the input types generated by *ElabType* ensure that the partial shape specification implied by a parameter's cell rank annotation is compatible with the possibly partial specification of that parameter's type.

Typing a `box` requires knowing how the hidden type indices are meant to relate to the overall type. Since we have two classes of index variables, ranging over shapes and dimensions respectively,  $\text{CHK:SIGMA}$  must first make sure that each  $\Sigma$ -bound variable in the goal type is allowed to represent the corresponding index. If so, then we also check that the body of the  $\Sigma$  with those indices substituted in can type the contents of the `box`, just as in checking a `box`'s type in explicit Remora. The syntactic difference between implicit and explicit Remora's `box` forms is

that implicit Remora excludes the type annotation, which can instead be provided using the general-purpose annotation ( $:$ ) form. The elaborated code, however, requires it.

$\text{CHK:PI}$  and  $\text{CHK:FORALL}$ , like analogous rules in explicit Remora, simply add the new type or index variables into the environment and check against the underlying type. Since there is no explicit  $\text{I}\lambda$  or  $\text{T}\lambda$ , the only way to introduce a  $\Pi$  or  $\forall$  is to force a term to be checked at that type. The expectation of a polymorphic type can propagate from other parts of the program by applying a function which expects a polymorphic function as an argument or placing functions in an array alongside another polymorphic function. However, this can be done most directly using an explicit annotation on the polymorphic term itself.

The question of how polymorphic array elements ought to be is not an issue for  $\text{CHK:ARRAY}$  and  $\text{CHK:FRAME}$ , since it is specified by the goal type. Similar to  $\text{CHK:FN}$ , there is the potential for disagreement between the shape specified in the array or frame form and that specified by the goal type, so a shape equality check is required. In the case where the goal shape was partially unresolved, this can resolve shape or dimension variables appearing in the goal.

### 6.3.3 Application

The application judgment, responsible for identifying how both cell and frame polymorphism are used and making the instantiation of cell polymorphism explicit, is written as

$$\bar{\Gamma};\Phi \vdash (e_f : \tau_f) \bullet [\bar{e}_a \dots] \Rightarrow \tau_r \dashv \bar{\Gamma}';\Phi' \hookrightarrow e_r$$

In the given environment, if the explicitly-typed function  $e_f$  has type  $\tau_f$ :

- Applying it to arguments  $\bar{e}_a \dots$  produces a result of type  $\tau_r$
- The environment is updated to  $\bar{\Gamma}';\Phi'$
- The application form elaborates to  $e_r$

The input  $e_f$  is only needed for elaboration. In a non-elaborating system, only its type  $\tau_f$  would be needed.

The invariant to maintain through the application type derivation is that the “input” elaborated function may be partly instantiated but must carry along a type which describes how much more instantiation is needed. In essence, the judgment asks, “If  $e_f$  had type  $\tau_f$ , what type would we get by applying it to the arguments  $e_a \dots$ ?” Each step in the derivation either removes one layer of type or index polymorphism or, after reaching an  $\dashv$ , removes one input type. When a polymorphism layer is removed, elaboration generates unsolved existential variables in order to later solve for the type or index arguments. When an input type is removed, we update what we know about the application form’s principal frame, and

$$\boxed{\bar{\Gamma}; \Phi \vdash t \Leftarrow \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow t}$$

$$\frac{\bar{\Gamma}_0; \Phi_0 \vdash \bar{t} \Rightarrow \tau_l \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow t \quad \bar{\Gamma}_1; \Phi_1 \vdash \tau_l \Leftarrow \tau_h \dashv \bar{\Gamma}_2; \Phi_2 \hookrightarrow \mathbf{C}}{\bar{\Gamma}_0; \Phi_0 \vdash \bar{t} \Leftarrow \tau_h \dashv \bar{\Gamma}_2; \Phi_2 \hookrightarrow \mathbf{C}[t]} \text{CHK:SUB}$$

$$\frac{\bar{\Gamma}_0, \blacktriangleright_{x_f}, \text{NewVars} \llbracket x, \varsigma \rrbracket \dots, x : \text{ElabType} \llbracket x, \varsigma \rrbracket \dots; \Phi_0 \vdash \tau_i \Leftarrow \text{ElabType} \llbracket x, \varsigma \rrbracket \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \mathbb{E} \dots \quad \bar{\Gamma}_n; \Phi_n \vdash \bar{e} \Leftarrow \tau_o \dashv \bar{\Gamma}_{n+1}, \blacktriangleright_{x_f}, \bar{\Gamma}_{n+2}; \Phi_{n+1} \hookrightarrow e \quad \text{with fresh } x_f}{\bar{\Gamma}_0; \Phi_0 \vdash (\lambda ((x \varsigma) \dots) \bar{e}) \Leftarrow (-> (\tau_i \dots) \tau_o) \dashv \bar{\Gamma}_{n+1}; \Phi_{n+1} \hookrightarrow (\lambda ((x \tau_i) \dots) e[x \mapsto \mathbb{E}[x], \dots])} \text{CHK:FN}$$

$$\frac{\bar{\Gamma}_0 \vdash \iota :: \text{Sort} \llbracket x \rrbracket \dots \quad \bar{\Gamma}_0; \Phi_0 \vdash \bar{e} \Leftarrow \tau[x \mapsto \iota, \dots] \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow e}{\bar{\Gamma}_0; \Phi_0 \vdash (\text{box } \iota \dots \bar{e}) \Leftarrow (\Sigma (x \dots) \tau) \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow (\text{box } \iota \dots e \text{ElabType} \llbracket (\Sigma (x \dots) \tau) \rrbracket)} \text{CHK:SIGMA}$$

$$\frac{\bar{\Gamma}_0, x \dots; \Phi_0 \vdash \mathbf{v} \Leftarrow \Upsilon \dashv \bar{\Gamma}_1, x \dots, \bar{\Gamma}_2; \Phi_1 \hookrightarrow \mathbf{v}}{\bar{\Gamma}_0; \Phi_0 \vdash \mathbf{v} \Leftarrow (\Pi (x \dots) (A \Upsilon (\text{shape}))) \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow (\text{I} \lambda ([x] \dots) (\text{array } () \mathbf{v}))} \text{CHK:PI}$$

$$\frac{\bar{\Gamma}_0, x \dots; \Phi_0 \vdash \mathbf{v} \Leftarrow \Upsilon \dashv \bar{\Gamma}_1, x \dots, \bar{\Gamma}_2; \Phi_1 \hookrightarrow \mathbf{v}}{\bar{\Gamma}_0; \Phi_0 \vdash \mathbf{v} \Leftarrow (\forall (x \dots) (A \Upsilon (\text{shape}))) \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow (\text{T} \lambda ([x] \dots) (\text{array } () \mathbf{v}))} \text{CHK:ALL}$$

$$\frac{\prod n \dots = \text{Length} \llbracket \mathbf{a} \dots \rrbracket \quad \bar{\Gamma}_0; \Phi_0 \vdash \bar{\mathbf{a}} \Leftarrow \tau \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \mathbf{a} \dots \quad \bar{\Gamma}_m; \Phi_m \models (\text{shape } n \dots) \doteq \iota \dashv \bar{\Gamma}_{m+1}; \Phi_{m+1}}{\bar{\Gamma}_0; \Phi_0 \vdash (\text{array } (n \dots) \bar{\mathbf{a}} \dots) \Leftarrow (A \tau \iota) \dashv \bar{\Gamma}_{m+1}; \Phi_{m+1} \hookrightarrow (\text{array } (n \dots) \mathbf{a} \dots)} \text{CHK:ARRAY}$$

$$\frac{\prod n \dots = \text{Length} \llbracket e \dots \rrbracket \quad \bar{\Gamma}_0; \Phi_0 \vdash \bar{e} \Leftarrow (A \tau \hat{x}) \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow e \dots \quad \text{with fresh } \hat{x} \quad \bar{\Gamma}_n; \Phi_n \models (++) (\text{shape } n \dots) \hat{x} \doteq \iota \dashv \bar{\Gamma}_{n+1}; \Phi_{n+1}}{\bar{\Gamma}_0; \Phi_0 \vdash (\text{array } (n \dots) e \dots) \Leftarrow (A \tau \iota) \dashv \bar{\Gamma}_{n+1}; \Phi_{n+1} \hookrightarrow (\text{array } (n \dots) e \dots)} \text{CHK:FRAME}$$

Figure 6.4: Type checking judgment

we carry that new frame along as the function expression’s shape. In effect, we are pretending that the  $n$ -ary function  $e_f$  is actually curried and being applied to a spine of arguments one by one. At each step in the quasi-curried application, the shape attributed to  $e_f$  accounts for the frames of all the arguments we have already consumed. This process ends when there are no input types left, and we are treating  $e_f$  as though its type had the form  $(A \ (-> \ () \ \tau_o) \ I_f)$ .

When the function expression is polymorphic, we must monomorphize it before consuming any arguments. A  $\forall$ -typed expression must be wrapped in an appropriate  $t$ -app by the APP:ALL rule and a  $\Pi$ -typed expression in an appropriate  $i$ -app by APP:PI. With Dunfield and Krishnaswami’s trick for delaying resolution [22], we can do this monomorphization by simply turning the  $\forall$ - and  $\Pi$ -bound variables into unsolved existential type and index variables.

The environment used in the premise is constructed by an array-type articulation metafunction, which accounts for the possibility that the type underneath the quantifier might not be explicitly an Arr. *Articulate* converts an existential array-type variable into an Arr built from fresh existential atom-type and shape variable and builds an appropriately extended environment:

$$\begin{aligned} \text{Articulate}_{\bar{\Gamma}_l, \hat{\rho}, \bar{\Gamma}_r} \llbracket \hat{\rho} \rrbracket &= (\bar{\Gamma}_l, \hat{\alpha}, \hat{\sigma}, \hat{\rho} \mapsto (A \ \hat{\alpha} \ \hat{\sigma}), \bar{\Gamma}_r; (A \ \hat{\alpha} \ \hat{\sigma})) \\ &\quad \text{with fresh } \hat{\alpha}, \hat{\sigma} \\ \text{Articulate}_{\bar{\Gamma}} \llbracket \tau \rrbracket &= (\bar{\Gamma}; \tau) \quad \text{otherwise} \end{aligned}$$

Subtype checks which are required later, when checking arguments’ types and using the function’s result type, can resolve the existential variables used for articulation and instantiation arguments. In order to maintain the application judgment’s invariant, we must pass the  $t$ -app or  $i$ -app “upwards” as input to the premise judgment. If we are handed a  $\Pi$  around an  $->$ , the judgment rules which deal with arguments and their frames will need an elaborated term which is typable with an  $->$ . For nested layers of  $\forall$  and  $\Pi$ , instantiating in the upward position ensures that the instantiation layer closest to the original expression corresponds to the outermost layer of polymorphism: a  $\forall$  containing a  $\Pi$  should be  $t$ -apped and then  $i$ -apped.

There are three rules which might be applicable when consuming an argument to an  $->$ -typed function. The *CellPoly* metafunction enforces the disambiguating restriction that a function which is polymorphic in some argument’s *cell rank* must be monomorphic in its frame rank. It is true for all universal and existential type variables, as well as any Arr type whose shape includes a universal or existential shape variable. If the function is polymorphic in the argument’s cell rank, only APP:FN\*C is applicable. No frame matching is needed because of the mandatory scalar frame, so APP:FN\*C looks much like an application rule for a

conventional non-array oriented language. Otherwise, synthesis uses  $\text{APP:FN}^*\text{F}$  or  $\text{APP:FN}^*\text{A}$ .

$\text{APP:FN}^*\text{F}$  describes the case where the function array contributes the largest frame seen so far, and the argument must therefore be frame-extended. Checking the argument against type  $(A \ \tau_i \ ( ++ \widehat{\sigma}_F \ I_i ))$ , *i.e.*, a  $\widehat{\sigma}_F$  frame of  $I_i$  cells, uses the subtype judgment to resolve  $\widehat{\sigma}_F$ . We then identify the argument frame extension by equating the function frame  $I_f$  with the extended argument frame  $( ++ \widehat{\sigma}_F \ \widehat{\sigma}_e )$ . Having consumed the argument  $\overline{e}_a$ , we drop it from the list of remaining arguments, pretend  $e_f$  now lacks its first input type  $(A \ \tau_i \ I_i)$ , and then continue processing the remaining arguments  $\overline{e}'_a \dots$ . Once we have elaborated forms for the remaining arguments  $e'_a \dots$ , we reinsert the elaborated  $e_a$  into the argument list.

$\text{APP:FN}^*\text{A}$  is similar, except that it is applicable when the function frame must extend to match the argument frame. The first difference appears in the shape equality premise, where we equate the argument frame  $\widehat{\sigma}_F$  with an extension of the function expression's shape  $I_f$ , rather than the other way around. The type we pretend  $e_f$  has when processing the remaining arguments now uses the frame shape from  $\overline{e}_a$  instead of the function array's old frame shape. This way, we ensure that all later arguments are still compared for compatibility with the current argument's frame.

When there are no more arguments left to handle,  $\text{APP:FN}0$  says that the result type is cells of the function's output type,  $(A \ \tau_o \ \iota_o)$ , inside whatever frame shape  $\iota_f$  was determined during the pass over the arguments. None of the application judgment rules for dealing with an  $\rightarrow$  in function position need to elaborate the function expression itself. By the time these rules are applicable, it has already been cell-monomorphized.

Forcing frame comparisons to be handled one argument at a time keeps the shape theory solver from having to select which function or argument frame in an application to use as the principal frame. Doing so would require support for disjunction, which raises decidability problems (this is discussed more thoroughly in Chapter 7). Offering these two rules keeps all disjunctive reasoning confined within the bidirectional rules and out of the theory solver.

Although  $\text{APP:FN}^*\text{F}$  and  $\text{APP:FN}^*\text{A}$  are presented formally as two separate rules, they only differ in outcome once one of them fails to solve the frame equation in the second premise. A practical implementation might prefer to simply identify the argument frame and then choose one applicable rule. The PLT Redex model used to develop this presentation faces a separate problem. Since Redex will explore all possible derivations for a judgment, any case where an argument's frame is the same as the largest frame seen so far allows *both* rules to succeed, leading to possible exponential blowup in the number of derivations. This can be

$$\boxed{\bar{\Gamma}; \Phi \vdash (e_f : \tau_f) \bullet [\bar{e}_a \dots] \Rightarrow \tau_r \dashv \bar{\Gamma}' ; \Phi' \hookrightarrow e_r}$$

where  $(\bar{\Gamma}_r; (A \tau_f \iota_f)) = \text{Articulate}_{\bar{\Gamma}_0} \llbracket T_f \rrbracket$   
 $\bar{\Gamma}_a, \hat{x} \dots ; \Phi_0 \vdash$   
 $((t\text{-app } e_f \hat{x} \dots) : (A \tau_f (++) \iota_a \iota_f)) [x \mapsto \hat{x}, \dots]$   
 $\bullet [e_a \dots] \Rightarrow \tau_r \dashv \bar{\Gamma}_1 ; \Phi_1 \hookrightarrow (e'_f e_a \dots)$   


---

 $\bar{\Gamma}_0 ; \Phi_0 \vdash (e_f : (A (\forall (x \dots) T_f) \iota_a))$   
 $\bullet [e_a \dots] \Rightarrow \tau_r \dashv \bar{\Gamma}_1 ; \Phi_1 \hookrightarrow (e'_f e_a \dots)$   
APP:ALL

where  $(\bar{\Gamma}_r; (A \tau_f \iota_f)) = \text{Articulate}_{\bar{\Gamma}_0} \llbracket T_f \rrbracket$   
 $\bar{\Gamma}_a, \hat{x} \dots ; \Phi_0 \vdash$   
 $((i\text{-app } e_f \hat{x} \dots) : (A \tau_f (++) \iota_a \iota_f)) [x \mapsto \hat{x}, \dots]$   
 $\bullet [e_a \dots] \Rightarrow \tau_r \dashv \bar{\Gamma}_1 ; \Phi_1 \hookrightarrow (e'_f e_a \dots)$   


---

 $\bar{\Gamma}_0 ; \Phi_0 \vdash (e_f : (A (\Pi (x \dots) T_f) \iota_a))$   
 $\bullet [e_a \dots] \Rightarrow \tau_r \dashv \bar{\Gamma}_1 ; \Phi_1 \hookrightarrow (e'_f e_a \dots)$   
APP:PI

where  $\neg \text{CellPoly} \llbracket \tau_i \rrbracket$  with fresh  $\widehat{\sigma}_F, \widehat{\sigma}_E$   
 $\bar{\Gamma}_0, \widehat{\sigma}_F, \widehat{\sigma}_E ; \Phi_0 \vdash \bar{e}_a \Leftarrow (A \tau_i (++) \widehat{\sigma}_F I_i) \dashv \bar{\Gamma}_1 ; \Phi_1 \hookrightarrow e_a$   
 $\bar{\Gamma}_1 ; \Phi_1 \models I_f \doteq (++) \widehat{\sigma}_F \widehat{\sigma}_E \equiv \bar{\Gamma}_2 ; \Phi_2$   
 $\bar{\Gamma}_2 ; \Phi_2 \vdash (e_f : (A (-> (\tau'_a \dots) \tau_o) I_f))$   
 $\bullet [e'_a \dots] \Rightarrow \tau_r \dashv \bar{\Gamma}_3 ; \Phi_3 \hookrightarrow (e_f e'_a \dots)$   


---

 $\bar{\Gamma}_0 ; \Phi_0 \vdash (e_f : (A (-> ((A \tau_i I_i) \tau'_a \dots) \tau_o) I_f))$   
 $\bullet [e_a e'_a \dots] \Rightarrow \tau_r \dashv \bar{\Gamma}_3 ; \Phi_3 \hookrightarrow (e_f e_a e'_a \dots)$   
APP:FN\*F

where  $\neg \text{CellPoly} \llbracket \tau_i \rrbracket$  with fresh  $\widehat{\sigma}_F, \widehat{\sigma}_E$   
 $\bar{\Gamma}_0, \widehat{\sigma}_F, \widehat{\sigma}_E ; \Phi_0 \vdash e_a \Leftarrow (A \tau_i (++) \widehat{\sigma}_F I_i) \dashv \bar{\Gamma}_1 ; \Phi_1 \hookrightarrow e_a$   
 $\bar{\Gamma}_1 ; \Phi_1 \models \widehat{\sigma}_F \doteq (++) I_f \widehat{\sigma}_E \equiv \bar{\Gamma}_2 ; \Phi_2$   
 $\bar{\Gamma}_2 ; \Phi_2 \vdash (e_f : (A (-> (\tau'_a \dots) \tau_o) I_f))$   
 $\bullet [e'_a \dots] \Rightarrow \tau_r \dashv \bar{\Gamma}_3 ; \Phi_3 \hookrightarrow (e_f e'_a \dots)$   


---

 $\bar{\Gamma}_0 ; \Phi_0 \vdash (e_f : (A (-> ((A \tau_i I_i) \tau'_a \dots) \tau_o) I_f))$   
 $\bullet [e_a e'_a \dots] \Rightarrow \tau_r \dashv \bar{\Gamma}_3 ; \Phi_3 \hookrightarrow (e_f e_a e'_a \dots)$   
APP:FN\*A

where  $\text{CellPoly} \llbracket \tau_i \rrbracket$   $\bar{\Gamma}_0 ; \Phi_0 \vdash \bar{e}_a \Leftarrow T_i \dashv \bar{\Gamma}_1 ; \Phi_1 \hookrightarrow e'_f$   
 $\bar{\Gamma}_1 ; \Phi_1 \vdash (e_f : (A (-> (T'_i \dots) \tau_o) I_f))$   
 $\bullet [\bar{e}_a \dots] \Rightarrow T_r \dashv \bar{\Gamma}_2 ; \Phi_2 \hookrightarrow (e'_f e'_a \dots)$   


---

 $\bar{\Gamma}_0 ; \Phi_0 \vdash (e_f : (A (-> (T_i T'_i \dots) \tau_o) I_f))$   
 $\bullet [\bar{e}_a ; e'_a \dots] \Rightarrow \tau_r \dashv \bar{\Gamma}_2 ; \Phi_2 \hookrightarrow (e'_f e_a e_a \dots)$   
APP:FN\*C

---

 $\bar{\Gamma} ; \Phi \vdash (e : (A (-> (A \tau_o I_o)) \iota_f))$   
 $\bullet [] \Rightarrow (A \tau_o (++) I_f I_o) \dashv \bar{\Gamma} ; \Phi \hookrightarrow (e)$   
APP:FN0

Figure 6.5: Application synthesis judgment

averted by having one of the two rules fail when the frame extension  $\widehat{\sigma}_E$  is found to be (shape).

#### 6.4 SUBTYPING JUDGMENT FORMS

The purpose of the subtype judgment is to resolve existential type and index variables based on how types which mention them are used. When an array of type  $(A \hat{t} \hat{s})$  is used where an  $(A \text{ Int } (\text{shape } 3))$  is required, the bidirectional rule which discovered that use will attempt to derive  $(A \hat{t} \hat{s}) \leq (A \text{ Int } (\text{shape } 3))$  as a premise. In doing so, the subtype judgment will update the environment to reflect that  $\hat{t}$  stands for  $\text{Int}$  and  $\hat{s}$  for  $(\text{shape } 3)$ . The task of checking subtyping questions in an environment which may contain unresolved variables can be conceptually split into two pieces, for which we use separate judgments: breaking down types to determine which variables instantiate as which pieces and then updating the environment to reflect those instantiations. The environment-update task covers the “base case” of reasoning about subtyping, so we discuss the variable-instantiation judgments first. However, some instantiation rules do involve additional destructuring steps.

While the subtype judgment emits a coercing context, Remora’s term-level syntax, being stratified into atoms and expressions, limits the opportunities for coercing atoms. All computation is stated in expressions, not atoms, so there is no general way to coerce an implicitly typed polymorphic function to an explicitly typed monomorphic function. For example, the polymorphic identity function and the integer increment function ought to be able to coexist in the same array, with the identity function instantiated for integers. However the simple

```
(array (2) (%id %inc))
```

is not well-typed without somehow coercing the polymorphic `%id` to have the monomorphic type

```
(-> ((A Int (shape)))
     (A Int (shape)))
```

We cannot simply insert a `t-app` wrapper because type application requires an expression rather than an atom. This makes the array literal something of a dead end. The only atom form which has the desired type is a  $\lambda$ -abstraction, and producing an appropriate one may require looking several layers down into the original type to find the piece which can match the goal type. The result code would be something like

```
(λ ((x (A Int (shape)))
    ((t-app id Int) x)))
```

Having to generate a  $\lambda$  as the outermost part of the coercion for `id`—or more generally, having to choose the introduction form corresponding



to the goal type—means that subtype rules for  $\forall$  and  $\prod$  types must be specific to the form of goal type. A more easily generalized option is to instead aim to coerce arrays of polymorphic values.

Requiring polymorphic functions which are implicitly instantiated to be wrapped in their own array literal is not a terrible burden due to the nested-vector syntax sugar for large arrays. Since the surface syntax for Remora uses  $[\mathfrak{a} \dots]$  to stand for a `frame` form of the appropriate vector shape, each atom  $\mathfrak{a}$  is implicitly wrapped in a scalar array form. This array can be coerced using `t-app` and `i-app` directly.

It is useful to have a collection of metafunctions which describe how coercions are constructed. The judgments described in this section make use of three: *LiftC* converts a coercion between atom types into a coercion between scalar arrays; *EachC* converts a coercion between array types of a particular shape into a coercion for arrays of some larger shape; and *FnC* builds a coercion for between function types from the coercions for their corresponding input and output types. Since these metafunctions work by constructing explicitly typed functions which perform the appropriate conversion steps, they need to know the input types for those generated functions.

$$\begin{aligned} & \text{LiftC}_{\mathcal{T}} \llbracket \mathbb{A} \rrbracket = \\ & ((\text{array } () \\ & \quad (\lambda ((x \text{ (A } \mathcal{T} \text{ (shape))})) \\ & \quad \quad (\text{array } () \mathbb{A}[x]))) \\ & \quad \square) \end{aligned}$$

$$\begin{aligned} & \text{EachC}_{\mathcal{T}} \llbracket \mathbb{E} \rrbracket = \\ & ((\text{array } () \\ & \quad (\lambda ((x \text{ T})) \\ & \quad \quad \mathbb{E}[x])) \\ & \quad \square) \end{aligned}$$

$$\begin{aligned} & \text{FnC}_{(-\rightarrow (T_{il} \dots) T_{ol}) \rightarrow (-\rightarrow (T_{ih} \dots) T_{oh})} \llbracket (\mathbb{E}_i \dots); (\mathbb{E}_o) \rrbracket = \\ & ((\text{array } () \\ & \quad (\lambda ((x_f \text{ (A } (-\rightarrow (T_{ih} \dots) T_{oh}) \text{ (shape))})) \\ & \quad \quad (\text{array } () \\ & \quad \quad \quad (\lambda ((x_i \text{ T}_{il}) \dots) \\ & \quad \quad \quad \quad (\mathbb{E}_o[(x_f \mathbb{E}_i[x_i] \dots ])))))) \\ & \quad \square) \end{aligned}$$

In the interest of generating smaller elaborated code, all three coercion metafunctions can be augmented with a special case which produces the identity coercion  $\square$  when building entirely from identity coercions.

6.4.1 *Existential Variable Instantiation*

When a subtype goal is articulated enough that one side of the subtyping relation is an unsolved existential variable, it is time to update the environment. We use two judgments for this instantiation, depending on whether the variable to be instantiated is meant to be a subtype or supertype of the goal type:

$$\bar{\Gamma}; \Phi \vdash \hat{\mathcal{X}} : \leq \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbf{C}$$

and

$$\bar{\Gamma}; \Phi \vdash \tau \leq : \hat{\mathcal{X}} \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbf{C}$$

In the simplest cases (the SOLVE rules), we have a monotype  $\mu$  as the goal for  $\hat{\mathcal{X}}$ . As long as  $\bar{\Gamma}_0$ , the portion of the environment to the left of  $\hat{\mathcal{X}}$ , is sufficient to show that  $\mu$  is well-formed at the kind appropriate for the variable  $\mathcal{X}$ ,<sup>24</sup> we update the environment entry for  $\hat{\mathcal{X}}$  to show  $\mu$  as the solution. The goal  $\mu$  will not be properly kinded under  $\bar{\Gamma}_0$  if  $\mu$  mentions universal type variables introduced in  $\bar{\Gamma}_1$ . Such a situation corresponds to a point in the program where explicitly typed code would have to mention a type variable which is not in scope. The program must then be ill-typed.

<sup>24</sup> Recall, using multiple classes of variables means that an atom type can only be represented by an atom-type variable and an array type by an array-type variable.

However, if the goal for type variable instantiation is an *existential* type variable which appears later in the environment, the elaborated code need not mention an out-of-scope variable. That later-bound existential variable is a temporary stand-in for a concrete piece of syntax. We can update the environment to say that  $\hat{x}_0$  and  $\hat{x}_1$  stand for the same type by making whichever one appears later refer to the other (as in the REACH rules).

If that later-bound existential variable appears deep inside the goal type, rather than as the goal itself, we must pick apart the goal and articulate the variable we are instantiating accordingly. Although  $\widehat{\text{@v}}$  cannot stand for  $(\rightarrow ([\text{int}]) [\&\text{m}])$  if  $\&\text{m}$  is bound later, we can freely insert new existential variables to represent pieces of  $\widehat{\text{@v}}$ . Breaking  $\widehat{\text{@v}}$  into  $(\rightarrow (\widehat{\text{@in}}) \widehat{\text{@out}})$  and then  $\widehat{\text{@out}}$  into  $(A \widehat{\text{@out\_a}} \widehat{\text{@out\_s}})$ —i.e.,  $\widehat{\text{@v}}$  now stands for  $(\rightarrow (\widehat{\text{@in}}) (A \widehat{\text{@out\_a}} \widehat{\text{@out\_s}}))$ —gives the finer-grained pieces we need to use a SOLVE or REACH rule for  $\&\text{m}$ . Tracing through all the instantiations for the fresh existential variables, we will resolve  $\widehat{\text{@in}}$  as  $[\text{int}]$ ,  $\widehat{\text{@out\_s}}$  as  $(\text{shape})$ , and  $\&\text{m}$  as  $\widehat{\text{@out\_a}}$ .

Managing this simultaneous breakdown of the goal type and generation of fresh variables for its pieces is handled by the ARRAY rule and FN\* rule (specifically for arrays of functions). For ARRAY, we must generate a fresh atom-type variable  $\hat{a}$ . For FN\*, we need fresh existential variables for input atom types and shapes as well as the output atom type and shape. Both cases require a fresh existential shape variable for the

$$\boxed{\bar{\Gamma}; \Phi \vdash \hat{x} : \leq \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbf{C}}$$

$$\frac{\bar{\Gamma}_0 \vdash \mu :: \text{Kind} \llbracket \mathcal{X} \rrbracket}{\bar{\Gamma}_0, \hat{\mathcal{X}}, \bar{\Gamma}_1; \Phi \vdash \hat{\mathcal{X}} : \leq \mu \dashv \bar{\Gamma}_0, \hat{\mathcal{X}} \mapsto \mu, \bar{\Gamma}_1; \Phi \hookrightarrow \square} \text{ ILOW:SOLVE}$$

$$\frac{\text{Kind} \llbracket \mathcal{X}_0 \rrbracket = \text{Kind} \llbracket \mathcal{X}_1 \rrbracket}{\bar{\Gamma}_l, \hat{\mathcal{X}}_0, \bar{\Gamma}_c, \hat{\mathcal{X}}_1, \bar{\Gamma}_r; \Phi \vdash \hat{\mathcal{X}}_0 : \leq \hat{\mathcal{X}}_1 \dashv} \text{ ILOW:REACH}$$

$$\frac{\bar{\Gamma}_l, \hat{\mathcal{X}}_0, \bar{\Gamma}_c, \hat{\mathcal{X}}_1 \mapsto \mathcal{X}_0, \bar{\Gamma}_r; \Phi \hookrightarrow \square}{\bar{\Gamma}_l, \hat{\mathcal{X}}_0, \bar{\Gamma}_c, \hat{\mathcal{X}}_1 \mapsto \mathcal{X}_0, \bar{\Gamma}_r; \Phi \hookrightarrow \square}$$

$$\frac{\bar{\Gamma}_l, \hat{\alpha}, \hat{\sigma}, \hat{\mathcal{X}} \mapsto (A \hat{\alpha} \hat{\sigma}), \bar{\Gamma}_r; \Phi_0 \vdash \hat{\alpha} : \leq \Upsilon \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \mathbf{A}}{\bar{\Gamma}_1; \Phi_1 \models \hat{\sigma} \doteq I \equiv \bar{\Gamma}_2; \Phi_2 \quad \text{with fresh } \hat{\alpha}, \hat{\sigma}} \text{ ILOW:ARRAY}$$

$$\frac{\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r; \Phi_0 \vdash \hat{\mathcal{X}} : \leq (A \Upsilon I) \dashv \bar{\Gamma}_2; \Phi_2 \hookrightarrow \text{LiftC}_{\hat{\alpha}} \llbracket \mathbf{A} \rrbracket}{\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r; \Phi_0 \vdash \hat{\mathcal{X}} : \leq (A \Upsilon I) \dashv \bar{\Gamma}_2; \Phi_2 \hookrightarrow \text{LiftC}_{\hat{\alpha}} \llbracket \mathbf{A} \rrbracket}$$

$$\frac{\begin{array}{l} \bar{\Gamma}_l, \hat{\mathcal{X}}_i \dots, \hat{\mathcal{X}}_o, \hat{\sigma}, \hat{\mathcal{X}} \mapsto (A (-> (\hat{\mathcal{X}}_i \dots) \hat{\mathcal{X}}_o) \hat{\sigma}), \bar{\Gamma}_r; \Phi_0 \\ \vdash T_i : \leq \hat{\mathcal{X}}_i \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \mathbf{E}_i \dots \\ \bar{\Gamma}_n; \Phi_n \vdash \hat{\mathcal{X}}_o : \leq T_o \dashv \bar{\Gamma}_{n+1}; \Phi_{n+1} \hookrightarrow \mathbf{E}_o \\ \bar{\Gamma}_{n+1}; \Phi_{n+1} \models \hat{\sigma} \doteq I \equiv \bar{\Gamma}_{n+2}; \Phi_{n+2} \\ \text{with fresh } \mathcal{X}_i \dots, \mathcal{X}_o, \sigma \end{array}}{\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r; \Phi_0 \vdash \hat{\mathcal{X}} : \leq (A (-> (T_i \dots) T_o) I) \dashv \bar{\Gamma}_{n+2}; \Phi_{n+2}} \text{ ILOW:FN*}$$

$$\frac{\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r; \Phi_0 \vdash \hat{\mathcal{X}} : \leq (A (-> (T_i \dots) T_o) I) \dashv \bar{\Gamma}_{n+2}; \Phi_{n+2}}{\hookrightarrow \text{FnC}_{(-> (\hat{\mathcal{X}}_i \dots) \hat{\mathcal{X}}_o) \rightarrow (-> (T_i \dots) T_o)} \llbracket (\mathbf{E}_i \dots); (\mathbf{E}_o) \rrbracket}$$

$$\frac{\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r, x_a \dots; \Phi_0 \vdash \hat{\mathcal{X}} : \leq (A \Upsilon (++) I_f I_c) \dashv}{\bar{\Gamma}_1, x_a \dots, \bar{\Gamma}_2; \Phi_1 \hookrightarrow \mathbf{E}} \text{ ILOW:ALL*}$$

$$\frac{\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r; \Phi_0 \vdash \hat{\mathcal{X}} : \leq (A (\forall (x_a \dots) (A \Upsilon I_c)) I_f) \dashv \bar{\Gamma}_1; \Phi_1}{\hookrightarrow \text{EachC}_{(A \Upsilon I_c)} \llbracket (\text{array } () (\text{TL } ([x_a] \dots) \square)) \rrbracket \llbracket \mathbf{E} \rrbracket}$$

$$\frac{\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r, x_a \dots; \Phi_0 \vdash \hat{\mathcal{X}} : \leq (A \Upsilon (++) I_f I_c) \dashv}{\bar{\Gamma}_1, x_a \dots, \bar{\Gamma}_2; \Phi_1 \hookrightarrow \mathbf{E}} \text{ ILOW:PI*}$$

$$\frac{\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r; \Phi_0 \vdash \hat{\mathcal{X}} : \leq (A (\prod (x_a \dots) (A \Upsilon I_c)) I_f) \dashv \bar{\Gamma}_1; \Phi_1}{\hookrightarrow \text{EachC}_{(A \Upsilon I_c)} \llbracket (\text{array } () (\text{IL } ([x_a] \dots) \square)) \rrbracket \llbracket \mathbf{E} \rrbracket}$$

$$\frac{\begin{array}{l} \bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r, \blacktriangleright_{x_f}, \hat{\mathcal{S}}_a \dots; \Phi_0 \vdash \\ \hat{\mathcal{X}} : \leq (A \Upsilon (++) I_f I_c) \llbracket \mathcal{S}_a \mapsto \hat{\mathcal{S}}_a, \dots \rrbracket \\ \dashv \bar{\Gamma}_1, \blacktriangleright_{x_f}, \bar{\Gamma}_2; \Phi_1 \hookrightarrow \mathbf{E} \\ \text{with fresh } x_f \end{array}}{\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r; \Phi_0 \vdash \hat{\mathcal{X}} : \leq (A (\Sigma (\mathcal{S}_a \dots) (A \Upsilon I_c)) I_f) \dashv} \text{ ILOW:SIGMA*}$$

$$\frac{\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r; \Phi_0 \vdash \hat{\mathcal{X}} : \leq (A (\Sigma (\mathcal{S}_a \dots) (A \Upsilon I_c)) I_f) \dashv}{\bar{\Gamma}_1; \Phi_1 \hookrightarrow \text{EachC}_{\bar{\Gamma}_2} \llbracket (A \Upsilon (++) I_f I_c) \llbracket \mathcal{S}_a \mapsto \hat{\mathcal{S}}_a, \dots \rrbracket \rrbracket}$$

$$\llbracket (\text{array } () \bar{\Gamma}_2 \llbracket (\text{box } \hat{\mathcal{S}}_a \square (\Sigma (\mathcal{S}_a \dots) (A \Upsilon I_c))) \rrbracket \rrbracket \llbracket \mathbf{E} \rrbracket$$

Figure 6.6: Instantiating existential type variables as subtypes

$$\boxed{\bar{\Gamma}; \Phi \vdash \tau \leqslant: \hat{x} \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbb{C}}$$

$$\frac{\bar{\Gamma}_0 \vdash \mu :: \text{Kind}[\mathcal{X}]}{\bar{\Gamma}_0, \hat{\mathcal{X}}, \bar{\Gamma}_1; \Phi \vdash \mu \leqslant: \hat{\mathcal{X}} \dashv \bar{\Gamma}_0, \hat{\mathcal{X}} \mapsto \mu, \bar{\Gamma}_1; \Phi \hookrightarrow \square} \text{IHIGH:SOLVE}$$

$$\frac{\text{Kind}[\mathcal{X}_0] = \text{Kind}[\mathcal{X}_1]}{\bar{\Gamma}_l, \hat{\mathcal{X}}_0, \bar{\Gamma}_c, \hat{\mathcal{X}}_1, \bar{\Gamma}_r; \Phi \vdash \hat{\mathcal{X}}_1 \leqslant: \hat{\mathcal{X}}_0 \dashv \bar{\Gamma}_l, \hat{\mathcal{X}}_0, \bar{\Gamma}_c, \hat{\mathcal{X}}_1 \mapsto \mathcal{X}_0, \bar{\Gamma}_r; \Phi \hookrightarrow \square} \text{IHIGH:REACH}$$

$$\frac{\bar{\Gamma}_l, \hat{\alpha}, \hat{\sigma}, \hat{\mathcal{X}} \mapsto (A \hat{\alpha} \hat{\sigma}), \bar{\Gamma}_r; \Phi_0 \vdash \Upsilon \leqslant: \hat{\alpha} \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \mathbb{A} \quad \bar{\Gamma}_1; \Phi_1 \models \hat{\sigma} \doteq I \doteq \bar{\Gamma}_2; \Phi_2 \quad \text{with fresh } \hat{\alpha}, \hat{\sigma}}{\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r; \Phi_0 \vdash (A \Upsilon I) \leqslant: \hat{\mathcal{X}} \dashv \bar{\Gamma}_2; \Phi_2 \hookrightarrow \text{LiftC}_{\hat{\alpha}}[\mathbb{A}]} \text{IHIGH:ARRAY}$$

$$\frac{\bar{\Gamma}_l, \hat{\mathcal{X}}_i \dots, \hat{\mathcal{X}}_o, \hat{\sigma}, \hat{\mathcal{X}} \mapsto (A (-\> (\hat{\mathcal{X}}_i \dots) \hat{\mathcal{X}}_o) \hat{\sigma}), \bar{\Gamma}_r; \Phi_0 \vdash \hat{\mathcal{X}}_i \leqslant: T_i \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \mathbb{E}_i \dots \quad \bar{\Gamma}_n; \Phi_n \vdash T_o \leqslant: \hat{\mathcal{X}}_o \dashv \bar{\Gamma}_{n+1}; \Phi_{n+1} \hookrightarrow \mathbb{E}_o \quad \bar{\Gamma}_{n+1}; \Phi_{n+1} \models \hat{\sigma} \doteq I \doteq \bar{\Gamma}_{n+2}; \Phi_{n+2} \quad \text{with fresh } \mathcal{X}_i \dots, \mathcal{X}_o, \sigma}{\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r; \Phi_0 \vdash (A (-\> (T_i \dots) T_o) I) \leqslant: \hat{\mathcal{X}} \dashv \bar{\Gamma}_{n+2}; \Phi_{n+2} \hookrightarrow \text{FnC}_{(-\> (T_i \dots) T_o) \dashv (-\> (\hat{\mathcal{X}}_i \dots) \hat{\mathcal{X}}_o)}[(\mathbb{E}_i \dots); (\mathbb{E}_o)]} \text{IHIGH:}\rightarrow^*$$

$$\frac{\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r, \blacktriangleright_{\mathcal{X}_a} \hat{\mathcal{X}}_a \dots; \Phi_0 \vdash (A \Upsilon (++) I_f I_c)[\mathcal{X}_a \mapsto \hat{x}_a, \dots] \leqslant: \hat{\mathcal{X}} \dashv \bar{\Gamma}_1, \blacktriangleright_{\mathcal{X}_a} \bar{\Gamma}_2; \Phi_1 \hookrightarrow \mathbb{E}}{\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r; \Phi_0 \vdash (A (\forall (x_a \dots) (A \Upsilon I_c)) I_f) \leqslant: \hat{\mathcal{X}} \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \mathbb{E}[(\text{t-app } \square \bar{\Gamma}_2[\hat{x}_a] \dots)]} \text{IHIGH:ALL}^*$$

$$\frac{\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r, \blacktriangleright_{\mathcal{X}_a} \hat{x}_a \dots; \Phi_0 \vdash (A \Upsilon (++) I_f I_c)[x_a \mapsto \hat{x}_a, \dots] \leqslant: \hat{\mathcal{X}} \dashv \bar{\Gamma}_1, \blacktriangleright_{\mathcal{X}_a} \bar{\Gamma}_2; \Phi_1 \hookrightarrow \mathbb{E}}{\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r; \Phi_0 \vdash (A (\Pi (x_a \dots) (A \Upsilon I_c)) I_f) \leqslant: \hat{\mathcal{X}} \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \mathbb{E}[(\text{i-app } \square \bar{\Gamma}_2[\hat{x}_a] \dots)]} \text{IHIGH:PI}^*$$

$$\frac{\text{KB}[\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r]; \text{SB}[\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r] \vdash (A \Upsilon I_c) :: \text{Array} \quad \bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r, \mathcal{S}_a \dots; \Phi_0 \vdash (A \Upsilon (++) I_f I_c) \leqslant: \hat{\mathcal{X}} \dashv \bar{\Gamma}_1, \mathcal{S}_a \dots, \bar{\Gamma}_2; \Phi_1 \hookrightarrow \mathbb{E}}{\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r; \Phi_0 \vdash (A (\Sigma (\mathcal{S}_a \dots) (A \Upsilon I_c)) I_f) \leqslant: \hat{\mathcal{X}} \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \mathbb{E}[(\text{unbox } (\mathcal{S}_f \dots x_f) \square) x_f]} \text{IHIGH:SIGMA}^*$$

Figure 6.7: Instantiating existential type variables as supertypes

array itself. The use of a rule for arrays of functions—rather than for functions themselves—may seem odd, but it is essential in order to avoid the limits on atom-level coercion.

The eventual underlying source of all nontrivial coercions in instantiating existential type variables is the rules for handling polymorphism. Instantiation involving no polymorphic types at all can safely generate the identity coercion. The only major asymmetry between corresponding ILOW and IHIGH rules in the instantiation judgments appears in handling polymorphism, *i.e.*,  $\forall$ ,  $\Pi$ , and  $\Sigma$  types.

This type inference system only guesses monomorphic types, so if we need  $\hat{\mathcal{X}}$  to be a subtype of a universal type, ILOW:ALL\* must find some monomorphic type which can have quantifiers added on to it to produce the goal type. Instantiation is more restrictive than the subtyping judgment itself: The type of the identity function has a subtype (by reflexivity), but it has no *monomorphic* subtype. This type inference strategy will give up if forced to guess some portion of a type which is polymorphic. Requiring well-formedness means that this can only succeed if the body of the goal type does not mention its  $\forall$ -bound variables. In that case, the coercion performs any necessary atom conversion, then wraps each cell of the input as a scalar containing a type abstraction.

We have more flexibility in finding a monomorphic supertype of a universal, using IHIGH:ALL\*. Any instantiation of the universal—as would be produced by type application in the coercion—is a supertype, so we convert the universal type variables to unsolved existentials. The corresponding coercion is straightforward type application. Choosing the concrete type arguments is deferred to later reasoning about type structure.

The PI\* instantiation rules are nearly identical to the ALL\* rules, using index variables instead of type variables. A subtype of a dependent product type is a type which can have the appropriate  $\Pi$  quantifier added to it, and a subtype is an instantiation of the dependent product.

Dependent sums reverse the intuition from universals and dependent products. The monomorphic subtypes of a dependent sum are the array types which can be turned into that sum by hiding part of their shape information. While substituting particular indices into the body of a  $\Pi$  type produces a supertype, substituting them into the body of a  $\Sigma$  type produces a subtype. So ILOW:SIGMA\* generates existential index variables to represent the  $\Sigma$  type's existential witnesses—dual to a  $\Pi$  type's index arguments. The coercion must first apply whatever coercion turns the eventually chosen solution for  $\hat{\mathcal{X}}$  into the non-boxed form, and then it wraps each cell in a `box` with the appropriate witnesses.

IHIGH:SIGMA\*, tasked with finding a type we are guaranteed to get by unboxing the given dependent sum, must ensure that the shape of the boxes' contents does not depend on the existential witnesses. Like ILOW:ALL\* and ILOW:PI\*, this instantiation rule is more restrictive

than the corresponding subtyping rule because instantiation only guesses monomorphic types. There is no opportunity to `re-box` the extracted contents with shape information hidden.

#### 6.4.2 Subtyping Proper

The subtype judgment, used for premises in several bidirectional rules, is given in Figures 6.8 and 6.9. The judgment form itself is

$$\bar{\Gamma}_0; \Phi_0 \vdash \tau_l \leq \tau_h \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \mathbb{C}$$

This states that with the starting environment  $\bar{\Gamma}_0$  and archive  $\Phi_0$ , a value of type  $\tau_l$  can be used as a value of type  $\tau_h$  by wrapping it in the coercion  $\mathbb{C}$ , and newly discovered information about existential variables and indices gives the output environment  $\bar{\Gamma}_1$  and archive  $\Phi_1$ .

The simplest of these rules serve as the reflexive base cases, comparing matching base types (`SUB:BASE`) or type variables (`SUB:VAR`). No change to the environment or archive is needed, and having the lower and higher types match means we only need the identity coercion, represented by the trivial syntactic context  $\square$ .

Although `SUB:ARRAY` is usable for matching up base types and type variables, it has trouble with nontrivial coercions, such as instantiating polymorphic functions. To get around the limits of atom-level computation, several subtyping rules compare arrays of particular atom types rather than comparing the atom types directly. These rules are identified by the asterisks in their names.

Including similar rules for trivial coercion cases can simplify the generated coercion too. Where `SUB:ARRAY` turns the discovered atom coercion (such as the  $\square$  generated by `SUB:BASE`) into a function to apply to the lower-typed array,<sup>25</sup> `SUB:BASE*` can avoid mapping the identity function.

<sup>25</sup> It is possible for a practical implementation to include a special case for converting the atom coercion  $\square$  into the expression coercion  $\square$ .

Subtype rules for polymorphic types gain some flexibility by being allowed to relate two polymorphic types. While `SUB:ALL*L` and `SUB:ALL*R` look similar to `ILOW:ALL*` and `IHIGH:ALL*`, combining these rules makes it possible to find one polymorphic type to be more polymorphic than another. Consider two polymorphic types (in shorthand):

$$(\forall [\&q \ \&r] \ (-\> (\&q \ \&r) \ \&q))$$

$$(\forall [\&t] \ (-\> (\&t \ \&t) \ \&t))$$

We can relate the two as subtypes by first using `SUB:ALL*R` to introduce the universal type variable  $\&t$  into the environment, with the subgoal of relating the first polymorphic type to the monomorphic  $(-\> (\&t \ \&t))$

$$\boxed{\bar{\Gamma}; \Phi \vdash \tau \leq \tau' \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbf{C}}$$

$$\frac{}{\bar{\Gamma}; \Phi \vdash B \leq B \dashv \bar{\Gamma}; \Phi \hookrightarrow \square} \text{SUB:BASE}$$

$$\frac{}{\bar{\Gamma}; \Phi \vdash \mathcal{X} \leq \mathcal{X} \dashv \bar{\Gamma}; \Phi \hookrightarrow \square} \text{SUB:VAR}$$

$$\frac{}{\bar{\Gamma}; \Phi \vdash \hat{\mathcal{X}} \leq \hat{\mathcal{X}} \dashv \bar{\Gamma}; \Phi \hookrightarrow \square} \text{SUB:EXVAR}$$

$$\frac{\hat{x} \notin \text{FreeVars}[\tau] \quad \bar{\Gamma}_0; \Phi_0 \vdash \hat{\mathcal{X}} \leq \tau \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \mathbf{C}}{\bar{\Gamma}_0; \Phi_0 \vdash \hat{\mathcal{X}} \leq \tau \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \mathbf{C}} \text{SUB:INSTL}$$

$$\frac{\hat{x} \notin \text{FreeVars}[\tau] \quad \bar{\Gamma}_0; \Phi_0 \vdash \tau \leq \hat{\mathcal{X}} \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \mathbf{C}}{\bar{\Gamma}_0; \Phi_0 \vdash \tau \leq \hat{\mathcal{X}} \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \mathbf{C}} \text{SUB:INSTR}$$

$$\frac{\bar{\Gamma}_0; \Phi_0 \vdash \Upsilon_l \leq \Upsilon_h \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \mathbf{A} \quad \bar{\Gamma}_1; \Phi_1 \models I_l \doteq I_h \dashv \bar{\Gamma}_2; \Phi_2}{\bar{\Gamma}_0; \Phi_0 \vdash (\mathbf{A} \Upsilon_l I_l) \leq (\mathbf{A} \Upsilon_h I_h) \dashv \bar{\Gamma}_2; \Phi_2 \quad \hookrightarrow \text{LiftC}_{\Upsilon_l}[\mathbf{A}]} \text{SUB:ARRAY}$$

$$\frac{\bar{\Gamma}_0; \Phi_0 \models I_f \doteq I'_f \dashv \bar{\Gamma}_1; \Phi_1 \quad \bar{\Gamma}_1; \Phi_1 \vdash \tau'_i \leq \tau_i \dashv \bar{\Gamma}_2; \Phi_2 \hookrightarrow \mathbf{E}_i \dots \quad \bar{\Gamma}_n; \Phi_n \vdash \tau_o \leq \tau'_o \dashv \bar{\Gamma}_{n+1}; \Phi_{n+1} \hookrightarrow \mathbf{E}_o}{\bar{\Gamma}_0; \Phi_0 \vdash (\mathbf{A} (-> (\tau_i \dots) \tau_o) I_f) \leq (\mathbf{A} (-> (\tau'_i \dots) \tau'_o) I'_f) \dashv \bar{\Gamma}_{n+1}; \Phi_{n+1} \hookrightarrow \text{FnC}_{(-> (\tau_i \dots) \tau_o) \rightarrow (-> (\tau'_i \dots) \tau'_o)}[\mathbf{E}_i \dots; \mathbf{E}_o]} \text{SUB:FN*}$$

Figure 6.8: Subtype rules, part 1: simple type forms

$$\boxed{\bar{\Gamma}; \Phi \vdash \tau \leq \tau' \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbf{C}}$$

$$\frac{\begin{array}{l} \bar{\Gamma}_0, \blacktriangleright_{x_f}, \hat{\mathcal{X}} \dots; \Phi_0 \vdash (A \Upsilon_l (++) I_f I_c) \\ \leq (A \Upsilon_h I_h) \dashv \bar{\Gamma}_1, \blacktriangleright_{x_f}, \bar{\Gamma}_2; \Phi_1 \hookrightarrow \mathbb{E} \\ \text{with fresh } x_f \end{array}}{\bar{\Gamma}_0; \Phi_0 \vdash (A (\forall (\mathcal{X} \dots) (A \Upsilon_l I_c)) I_f) \leq (A \Upsilon_h I_h) \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \mathbb{E} \left[ (\text{t-app} \square \bar{\Gamma}_2 \left[ \left[ \hat{\mathcal{X}} \right] \dots \right] \right) \right]} \text{SUB:ALL*L}$$

$$\frac{\begin{array}{l} \bar{\Gamma}_0, \mathcal{X} \dots; \Phi_0 \vdash (A \Upsilon_l I_l) \leq (A \Upsilon_h (++) I_f I_c) \\ \dashv \bar{\Gamma}_1, \mathcal{X} \dots, \bar{\Gamma}_2; \Phi_1 \hookrightarrow \mathbb{E} \end{array}}{\bar{\Gamma}_0; \Phi_0 \vdash (A \Upsilon_l I_l) \leq (A (\forall (\mathcal{X} \dots) (A \Upsilon_h I_c)) I_f) \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \text{EachC}_{(A \Upsilon_l I_c)} \left[ \left[ (\text{array} ((\text{T}\lambda ([\mathcal{X}] \dots) \mathbb{E})) \right) \right]} \text{SUB:ALL*R}$$

$$\frac{\begin{array}{l} \bar{\Gamma}_0, \blacktriangleright_{x_f}, \hat{\mathcal{S}} \dots; \Phi_0 \vdash (A \Upsilon_l (++) I_f I_c) \\ \leq (A \Upsilon_h I_h) \dashv \bar{\Gamma}_1, \blacktriangleright_{x_f}, \bar{\Gamma}_2; \Phi_1 \hookrightarrow \mathbb{E} \\ \text{with fresh } x_f \end{array}}{\bar{\Gamma}_0; \Phi_0 \vdash (A (\Pi (\mathcal{S} \dots) (A \Upsilon_l I_c)) I_f) \leq (A \Upsilon_h I_h) \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \mathbb{E} \left[ (\text{i-app} \square \bar{\Gamma}_2 \left[ \left[ \hat{\mathcal{S}} \right] \dots \right] \right) \right]} \text{SUB:PI*L}$$

$$\frac{\begin{array}{l} \bar{\Gamma}_0, \mathcal{S} \dots; \Phi_0 \vdash (A \Upsilon_l I_l) \leq (A \Upsilon_h (++) I_f I_c) \\ \dashv \bar{\Gamma}_1, \mathcal{S} \dots, \bar{\Gamma}_2; \Phi_1 \hookrightarrow \mathbb{E} \end{array}}{\bar{\Gamma}_0; \Phi_0 \vdash (A \Upsilon_l I_l) \leq (A (\Pi (\mathcal{S} \dots) (A \Upsilon_h I_c)) I_f) \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \text{EachC}_{(A \Upsilon_l I_c)} \left[ \left[ (\text{array} ((\text{I}\lambda ([\mathcal{S}] \dots) \mathbb{E})) \right) \right]} \text{SUB:PI*R}$$

$$\frac{\begin{array}{l} \bar{\Gamma}_0, \widehat{\sigma}_h; \Phi_0 \models (++) I_f \widehat{\sigma}_h \doteq I_h \dashv \bar{\Gamma}_1; \Phi_1 \\ \bar{\Gamma}_1, \mathcal{S} \dots; \Phi_1 \vdash (A \Upsilon_l I_c) \leq (A \Upsilon_h \widehat{\sigma}_h) \\ \dashv \bar{\Gamma}_2, \mathcal{S} \dots, \bar{\Gamma}_3 \hookrightarrow \mathbb{E} \\ \text{with fresh } \sigma_h, \mathcal{S}_s \dots, x_s \end{array}}{\bar{\Gamma}_0; \Phi_0 \vdash (A (\Sigma (\mathcal{S} \dots) T_h) I_f) \leq (A \Upsilon_h I_h) \dashv \bar{\Gamma}_2; \Phi_2 \hookrightarrow (\text{unbox} (\mathcal{S}_s \dots x_s \square) \mathbb{E}[x_s])} \text{SUB:SIGMA*L}$$

$$\frac{\begin{array}{l} \text{where } T_h = (A \Upsilon_h I_c) \\ \bar{\Gamma}_0, \blacktriangleright_{x_s}, \hat{\mathcal{S}} \dots; \Phi_0 \vdash (A \Upsilon_l I_l) \\ \leq (A \Upsilon_h (++) I_f I_c) \left[ \mathcal{S} \mapsto \hat{\mathcal{S}}, \dots \right] \dashv \bar{\Gamma}_1, \blacktriangleright_{x_s}, \bar{\Gamma}_2; \Phi_1 \hookrightarrow \mathbb{E} \end{array}}{\bar{\Gamma}_0; \Phi_0 \vdash (A \Upsilon_l I_l) \leq (A (\Sigma (\mathcal{S} \dots) T_h) I_f) \dashv \bar{\Gamma}_1; \Phi_1 \hookrightarrow \bar{\Gamma}_1 \left[ \left[ \text{EachC}_{T_h[\mathcal{S} \mapsto \hat{\mathcal{S}}, \dots]} \right] \left[ (\text{array} ((\text{box } \hat{\mathcal{S}} \dots \square (\Sigma (\mathcal{S} \dots) T_h))) \right] \right] \mathbb{E} \right]} \text{SUB:SIGMA*R}$$

Figure 6.9: Subtype rules, part 2: polymorphic type forms



&t). This subgoal succeeds using SUB:ALL\*L to add existential type variables  $\hat{q}$  and  $\hat{r}$  to the environment and then eventually resolving them both as &t.

Similar results are available from SUB:PI\*R and SUB:PI\*L, relating types such as a function on integer vectors and a function on *nonempty* integer vectors:

$$(\Pi [\$r] \rightarrow ([\text{Int } \$r]) [\text{Int } \$r]))$$

$$(\Pi [\$t] \rightarrow ([\text{Int } (+ 1 \$t)]) [\text{Int } (+ 1 \$t)]))$$

First, SUB:PI\*R adds \$t to the environment. Then the subgoal of relating the first type to  $(\rightarrow ([\text{Int } (+ 1 \$t)]) [\text{Int } (+ 1 \$t)])$  is accomplished by SUB:PI\*L, which generates the existential dimension variable  $\hat{r}$  and resolves it as  $(+ 1 \$t)$ . Reversing the order, trying to conclude the second type is a subtype of the first, ends with the constraint solver failing to find a solution for the existential \$t equal to one less than the universal \$r.

Analogous reasoning allows dependent sums to be ordered by hiding more or less information about their contents' shapes. With existential quantification, the type of integer vectors of completely unknown length is higher than the type of integer vectors of nonzero length:

$$(\Sigma [\$t] [\text{Int } (+ 1 \$t)])$$

$$(\Sigma [\$r] [\text{Int } \$r])$$

We derive this result by using SUB:SIGMA\*L to add the universal variable \$t to the environment and then using SUB:SIGMA\*R to introduce the existential variable  $\hat{r}$ , which can be resolved as  $(+ 1 \$t)$ . Unlike in the instantiation rules, where only a flat array type can be found as a subtype of a dependent sum, the full subtyping judgment is able to relate two arrays of uncertain shape as long as what is known about one shape implies what is known about the other.

The limitation on atom coercions makes it potentially awkward to instantiate an existential atom type variable within the instantiation judgments. We can escape these limitations by merging the atom variable instantiation into what would be the subtype rule for an array containing an unsolved atom type and an array of  $\rightarrow$  type, producing the rules shown in Figure 6.10. Only rules for  $\rightarrow$  types are included here (SUB:INST $\rightarrow$ L and SUB:INST $\rightarrow$ R). No such rules are needed for base types, which use the identity coercion, and guessing only monomorphic types means that unresolved existential type variables do not stand for universals, dependent products, or dependent sums. In a language with additional type constructors for atoms, such as products or sums, equivalent rules for those constructors should be added.

$$\begin{array}{c}
\overline{\Gamma}_l, \widehat{\alpha}_i \dots, \widehat{\sigma}_i \dots, \widehat{\alpha}_o, \widehat{\sigma}_o, \widehat{\mathcal{X}} \mapsto \Upsilon_f, \overline{\Gamma}_r; \Phi_0 \vdash \\
(A \Upsilon_f I_l) \leq (A (-> (\tau_i \dots) \tau_o) I_h) \dashv \overline{\Gamma}_1; \Phi_1 \leftrightarrow \mathbb{E} \\
\text{where } \Upsilon_f = (-> ((A \widehat{\alpha}_i \widehat{\sigma}_i) \dots) (A \widehat{\alpha}_o \widehat{\sigma}_o)) \\
\text{with fresh } \widehat{\alpha}_i \dots, \widehat{\sigma}_i \dots, \widehat{\alpha}_o, \widehat{\sigma}_o \\
\hline
\overline{\Gamma}_l, \widehat{\mathcal{X}}, \overline{\Gamma}_r; \Phi_0 \vdash (A \widehat{\mathcal{X}} I_l) \leq (A (-> (\tau_i \dots) \tau_o) I_h) \\
\dashv \overline{\Gamma}_1; \Phi_1 \leftrightarrow \mathbb{E} \qquad \text{SUB:INST} \rightarrow \text{L}
\end{array}$$
  

$$\begin{array}{c}
\overline{\Gamma}_l, \widehat{\alpha}_i \dots, \widehat{\sigma}_i \dots, \widehat{\alpha}_o, \widehat{\sigma}_o, \widehat{\mathcal{X}} \mapsto \Upsilon_f, \overline{\Gamma}_r; \Phi_0 \vdash \\
(A (-> (\tau_i \dots) \tau_o) I_l) \leq (A \Upsilon_f I_h) \dashv \overline{\Gamma}_1; \Phi_1 \leftrightarrow \mathbb{E} \\
\text{where } \Upsilon_f = (-> ((A \widehat{\alpha}_i \widehat{\sigma}_i) \dots) (A \widehat{\alpha}_o \widehat{\sigma}_o)) \\
\text{with fresh } \widehat{\alpha}_i \dots, \widehat{\sigma}_i \dots, \widehat{\alpha}_o, \widehat{\sigma}_o \\
\hline
\overline{\Gamma}_l, \widehat{\mathcal{X}}, \overline{\Gamma}_r; \Phi_0 \vdash (A (-> (\tau_i \dots) \tau_o) I_l) \leq (A \widehat{\mathcal{X}} I_h) \\
\dashv \overline{\Gamma}_1; \Phi_1 \leftrightarrow \mathbb{E} \qquad \text{SUB:INST} \rightarrow \text{R}
\end{array}$$

Figure 6.10: Supplemental “reach-through” rules for instantiating existential variables

## FIRST-ORDER THEORY OF ARRAY SHAPES

Type checking only requires a very restricted fragment of the theory of the free monoid over the natural numbers. Pairs of indices are only checked for equality in isolation from each other, and no information about an index (other than its sort) is given in the program. So the check is for the validity of a single equality—no connectives or quantifiers needed. This fragment can be decided efficiently by comparing indices written in canonical form. Two `Dims` which are equal must simplify to sums with the same constant component and the same coefficient on corresponding variables. For example,

$$(+ x y 5 x) = (+ (+ x x) 5 y)$$

is valid because both simplify to  $2x + y + 5$ , whereas

$$(+ q 5 y) = (+ (+ x x) 5 y)$$

is false for any interpretation which does not assign  $q$  to twice the value assigned to  $x$  (and thus is not valid).

To decide the validity of an equality on Shapes (*i.e.*, sequences of naturals), we can again test by conversion to a canonical form: a sequence is written out as the concatenation of single `Dims` and Shape variables. Sorting rules guarantee that the individual elements of a sequence are natural numbers, and associativity permits nested appends to be collapsed away. Thus the index

$$(++ (\text{shape } 2 \ (+ x \ 5 \ x)) \ (++ \ d \ (\text{shape } 3)))$$

canonicalizes to

$$(++ (\text{shape } 2) \ (\text{shape } (+ x \ x \ 5)) \ d \ (\text{shape } 3))$$

To show that this process does produce a canonical form, consider two shapes in this form which differ, and focus on the leftmost differing position in their respective lists of appended components. If they are syntactically different singleton shapes—their respective contents are two different canonicalized naturals—then an assignment under which those naturals differ will also make the full shapes differ at this position. If one is a singleton (`shape  $\iota$` ) and the other a variable  $s$  (of sort `Shape`), then an interpretation which assigns the variables in  $\iota$  such that its components sum to  $n$  may also assign  $s$  to be the shape (`shape  $(+ n \ 1)$` ). Again, an interpretation forces the shapes to be unequal. Finally, if this position has variables  $s$  and  $t$ , choose an interpretation mapping  $s$  to (`shape 1`) and  $t$  to (`shape 2`) to produce unequal interpretations of the whole shapes.

Although type checking itself only requires this canonicalization process, constraint-based type inference would call for a more sophisticated solver due to the use of existential variables for choosing pieces of an index.

In order to describe when arguments’ shapes are compatible, it is useful to impose a lattice structure on the universe of shapes. The lattice is built with the order  $\sqsubseteq$  meaning that one shape is a prefix of another; a  $\top$  is added to represent the join of incompatible shapes (we already have  $\perp = \square$ , as the empty shape is a prefix of every shape). For shapes  $s_0$  and  $s_1$ , we have  $s_0 \sqcup s_1 \neq \top$  if and only if  $s_0 \sqsubseteq s_1$  or  $s_1 \sqsubseteq s_0$ . Generalizing to arbitrary finite joins,  $\bigsqcup\{s \dots\} \neq \top$  implies that the shapes  $s \dots$  are totally ordered, and the lattice structure means the shapes’ join is one of the shapes themselves.

## 7.1 STRUCTURE OF SOLVER QUERIES

The particular form of constraint problem associated with Remora’s type inference strategy asks, “How can existential variables be written in terms of universal variables in the environment so that these two shapes are equal?” This differs from the usual mode of use for constraint solvers. Instead of a satisfying assignment for a formula, which effectively treats *all* variables as existential, we want Skolem functions which show how to construct the existential variables’ values from the universals’ values.

Furthermore, instead of seeking concrete values or Skolem functions for all existential variables, we often want the solver to give a less specific answer about the values of existential dimension variables. When a function is written with a rank specifier such as  $(\lambda ((x \ 1) (y \ 2)) (+ \ x \ y))$ <sup>26</sup>, we shouldn’t necessarily infer a different argument for *every* input dimension. In this example, the length of the vector  $x$  must be the same as the leading dimension of the matrix  $y$ . Although it seems at first glance to have three dimension arguments—perhaps  $\$x1$ ,  $\$y1$ , and  $\$y2$ —it really ought to have only two. It is not enough for a shape equality solver to say whether  $\{++ \ [\$x1] \ f\} \doteq [\$y1 \ \$y2]$ . Those two shapes are not equal in general. Instead, we need the solver to account for the possibility that  $\$x1 \doteq \$y1$  and report that the shape equation is solved by  $f \mapsto [\$y2]$  as long as  $\$x1 \doteq \$y1$  (but it is unsolvable otherwise). Rather than viewing unsolved existential dimension variables as sequence elements to select, the solver must defer that selection.

The “satisfying assignment” produced by the shape theory solver then is a pair: a sequence of dimensions and universal shape variables corresponding to each existential shape variable and the minimal equivalence relation on dimensions that makes the offered solution correct. These are then used to construct the output environment and archive. The Redex model of Remora<sup>27</sup> calls out to a separate shape-theory solver, provided as a Racket package at <https://github.com/jrslepak/makanin-algo>.

<sup>26</sup> Recall, this is the expanded form of  $\sim(1 \ 2)+$ .

<sup>27</sup> <https://github.com/jrslepak/Revised-Remora>

Recall that it is not always safe to forget restrictions on existential variables that have gone out of scope because multiple different existential variables might be aliased to a single one. For example, suppose we have a local existential variable  $\hat{x}$ , introduce a new existential variable  $\hat{y}$ , and then discover via a shape constraint that  $\hat{y} + \hat{y} \doteq \hat{x}$ . This implies that  $\hat{x}$  must be an even number. If the function which introduced  $\hat{x}$  is found to be polymorphic over  $\hat{x}$ , then its elaborated form must instead parameterize over some fresh variable  $\$d$  which serves as the “witness” to the fact that  $\hat{x}$  is even (uses of  $\hat{x}$  will have to be rewritten as  $(+ \$d \$d)$ ). So we must remember for a time that  $\hat{x}$  is even. However, another variable which comes into scope after  $\hat{y}$  goes out of scope might introduce more restrictions on  $\hat{x}$ . Perhaps we eventually discover  $\hat{z} + \hat{z} + \hat{z} \doteq \hat{x}$ . It is not safe to conclude that  $\hat{x}$  must actually be thrice some universal variable. We need to remember the earlier result that  $\hat{x}$  is even as well. Combining these facts tells us instead that  $\hat{x}$  is 6 times some universal variable. We might continue finding more such restrictions until  $\hat{x}$  goes out of scope.

Equations on sequences may have multiple or even infinitely many solutions.

## 7.2 STRING EQUATIONS MODULO THEORIES

From the perspective of the bidirectional typing rules, invocation of the solver happens through a premise of the form

$$\bar{\Gamma}; \Phi \models \iota \doteq \iota' \Rightarrow \bar{\Gamma}'; \Phi'$$

The solver’s obligation, as described in Proposition 6.2.1, is to construct  $\bar{\Gamma}'$  and  $\Phi'$  by adding new solutions and existential variable entries to  $\bar{\Gamma}$  and  $\Phi$  so that the minimal equivalence relation which is compatible with decisions made during the solver’s internal search process is encoded in  $\bar{\Gamma}'$  and  $\Phi'$ .

Since we must account for the possibility that some existential shape variables generated during type inference stand for the empty shape, we must precede the semigroup solution search by nondeterministically choosing<sup>28</sup> a subset of the sequence variables to remove. In some cases it is immediately evident that a particular variable should stand for  $[\ ]$ , such as identifying the frame shape when applying a rank-1 function to rank-1 input, but this does not hold in the general case for equations on sequences. Although the worst case for this adaptation strategy is choosing from  $O(2^n)$  possible sets of variables to drop, typical queries for type inference purposes do not have many existential sequence variables.

<sup>28</sup> Most literature on solving equations on strings makes heavy use of nondeterminism, and Makanin’s algorithm is no exception. What is stated here as nondeterministic choice can be considered intuitively as branching in a backtracking search.

7.2.1 *A Sketch of Makanin's algorithm*

The algorithm for a free semigroup follows a search procedure which considers the ways that appended components of equated sequences might align with each other. Since both sides of an equation are meant as descriptions of the same sequence, Makanin's algorithm tries to find which contiguous region of the unknown sequence is described by each component. A potential alignment is described by stating which boundaries between appended components are located at the same place in the actual underlying sequence. For example, suppose the goal is to solve  $(++ \ x \ [3]) \doteq (++ \ [3] \ x)$ . The left-hand side of the equation can be broken into two components  $x$  and  $[3]$ , and the right-hand side breaks into  $[3]$  and  $x$ . Consider laying the equation's two sides out parallel to each other: the width of  $x$  is not yet certain. In our search for a solution to the equation, we must decide whether the left-hand side's component boundary occurs before, after, or simultaneous with the right-hand side's component boundary. That is, we decide whether the left-hand  $x$  is shorter than, longer than, or the same length as the right-hand  $[3]$ . The first case turns out to be impossible. Since  $x$  must stand for a nonempty sequence, it cannot end *before* the singleton sequence  $[3]$ .

The other two cases lead to actual solutions. In the second case, the left side's  $x$  is longer than  $[3]$ , and it must overlap with the  $x$  on the right. To try out a possible alignment of component boundaries, Makanin's algorithm will "transport" the left  $x$  to the right  $x$ 's position. In doing so, it must carry along all components that appear within that  $x$ 's boundaries. Since the left side's  $x$  begins with the right side's singleton  $[3]$ , the updated version of the equation tells us that  $x \doteq (++ \ [3] \ [3]) \doteq [3 \ 3]$ . So we choose  $[3 \ 3 \ 3]$  as the sequence described by each side of the original equation.

The third case proceeds more simply. Since the left  $x$  and the right  $[3]$  share both left and right boundaries, transporting the left  $x$  to the right  $x$ 's location moves the right  $[3]$  to cover the entire span of the right  $x$ . We get the equation  $x \doteq [3]$ . The resulting sequence for the original equation is  $[3 \ 3]$ .

The transport procedure which forms the core of Makanin's search algorithm is analogous to the unit propagation step of a DPLL satisfiability solver [16, 17]. After we discover that some sequence variable's meaning contains a particular sequence component at a particular position within it, we trace that implication through to subsequent occurrences of the same variable. However, specifying the "particular position" raises a problem. When choosing how components are aligned, we may have set things up so that different occurrences of one variable don't span the same number of abstract boundaries. For example, we might have equated  $x$  with  $[1 \ 2 \ 3]$  in one place and  $(++ \ y \ z)$  in another. We know

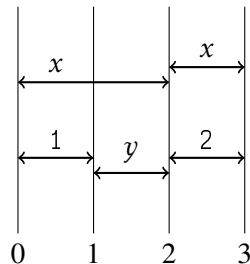
that  $y$  must start with 1 and that  $z$  must with 3, but whether 2 is the end of  $y$  or the beginning of  $z$  comes down to nondeterministic choice.

### 7.2.2 Makanin's algorithm

Makanin's algorithm operates on a "generalized equation," a data structure representing decisions made during the search process about which components of the equated sequence terms stand for which portions of the sequence they denote. For example, in the equation  $(++ [1] x) \doteq (++ x [1])$ , one occurrence of  $x$  represents some prefix of the sequence, and the other represents some suffix. Whether that prefix and suffix overlap is a decision the constraint solver must make while searching for a solution. In lining up the components of each side of the equation, a generalized equation decomposes the denoted sequence into "columns" which contain consecutive nonempty subsequences. Every boundary between components in a term on one side of the equation is also a boundary between columns. Since columns must be linearly ordered, we will index them by natural numbers.

The essential information a generalized equation tracks is a conjunction of atomic statements of the form, "This variable or sequence element spans this interval of columns." The basic search step in Makanin's algorithm is to find a variable which spans two different intervals (due to occurring multiple times in the equation) and copy the contents of one interval into the other. Each of these atomic statements is called a "base" in the generalized equation, and is represented by a record containing the column boundaries that mark the base's edges, the variable or sequence element constant spanning that interval, and a tag which will be used to indicate whether information from this base's range has already been propagated elsewhere. To keep the set of elements abstract, we will refer to individual elements as  $e_i$  with  $i$  ranging over  $\mathbb{N}$ . The programmatic representation of a base containing a specific element will store the identification number  $i$ . So individual bases take the form  $(v, [n, n'])$  meaning the variable  $v$  spans the interval from boundary  $n$  to boundary  $n'$  or  $(i, [n, n'])$  meaning that element  $e_i$  spans that interval. A generalized equation is then represented as a collection of bases.

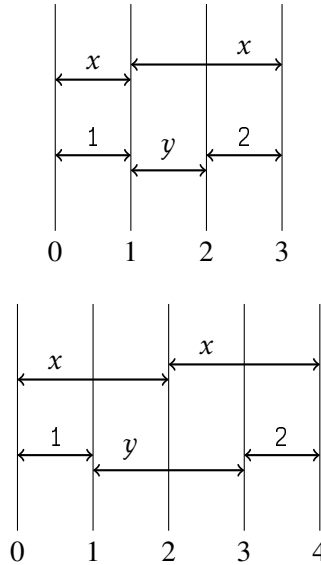
A generalized equation can be visualized as a table of columns with each base drawn with its label across its spanned columns:



In a serial notation, this generalized equation might be written as

$$\{(x, [0, 2]), (x, [2, 3]), (1, [0, 2]), (y, [1, 2]), (2, [2, 3])\}$$

There are often several generalized equations corresponding to a single equation. The equation which produced the above generalized equation,  $(++ \ x \ x) \doteq (++ \ [1] \ y \ [2])$ , has two other ways its components' boundaries might be aligned:



In serial notation, these might respectively be expressed as

$$\{(x, [0, 1]), (x, [1, 3]), (1, [0, 1]), (y, [1, 2]), (2, [2, 3])\}$$

and

$$\{(x, [0, 2]), (x, [2, 4]), (1, [0, 1]), (y, [1, 3]), (2, [3, 4])\}$$

The three generalized equations correspond to a decision about how the boundary between the two occurrences of  $x$  lines up with the boundaries of  $y$ : Does it coincide with the right boundary, coincide with the left boundary, or fall somewhere between them? Only the last of these options leads to a solution. The first two both require  $x$  to span exactly one column in order to be equal to the one constant base and span at least two columns in order to cover the other constant base and all of  $y$ .

Makanin's algorithm therefore begins with a nondeterministic choice of how to align the components on opposite sides of the equation. Since we intend to compile Remora on a deterministic machine, nondeterministic choice is encoded as backtracking search. The left-hand and right-hand sides' leftmost boundaries must coincide, as must their rightmost boundaries, but intermediate boundaries might be sometimes interleaved and sometimes simultaneous. A component alignment, which generates a generalized equation, can be viewed as a sequence of decisions

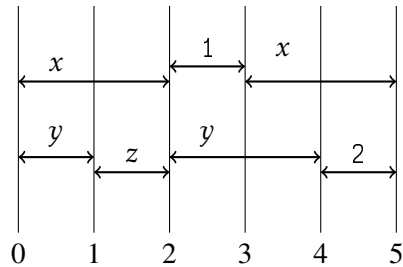


about which side of the equation will have an intermediate component boundary at a given point in the equation's denoted sequence. To generate alignments, make a series of nondeterministic choices of left-hand side, right-hand side, or both—as long as both sides still have unused component boundaries—and after each choice, place the chosen side's (or sides') next component boundary. Once all of one side's component boundaries have been placed, follow them with the other side's remaining boundaries.

As an example, suppose we have the equation

$$(++ x [1] x) \doteq (y z y [2])$$

We need to decide where to place two component boundaries from the left-hand side (between the first  $x$  and  $[1]$  and then between  $[1]$  and the other  $x$ ) and three component boundaries from the right-hand side (between the first  $y$  and  $z$ , between  $z$  and the second  $y$ , and between the second  $y$  and  $[2]$ ). The leftmost component boundary must be column boundary 0. Our first choice is whether column boundary 1 is a component boundary for the left, right, or both. If we first choose to place a right-hand side boundary, we have a base  $(y, [0, 1])$ . We might then choose to place a component boundary for both sides at column boundary 2, generating bases  $(x, [0, 2])$  and  $(z, [1, 2])$ . Placing a left-hand side boundary at column boundary 3 would produce  $(1, [2, 3])$ . We would then be out of component boundaries on the left, so column boundary 4 would have to have only a right-hand side component boundary, producing the base  $(y, [2, 4])$ . Finally, column boundary 5 finishes both sides of the equation, with bases  $(x, [3, 5])$  and  $(2, [4, 5])$  completing the generalized equation:

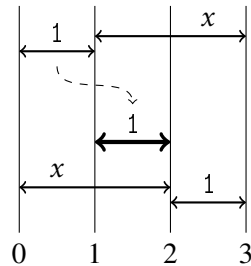


Generating the possible component alignments is analogous to generating monotonic lattice paths—moving from one grid point to another using only northward and eastward movements. Unlike the North-East lattice path problem which is solved by counting combinations, these lattice paths also may include diagonal Northeast steps. This generalization of N-E lattice paths to N-E-NE was investigated by Delannoy as the number of monotonic paths a queen can take across a chess board [18], which are now known as Delannoy numbers.

Once we have a generalized equation, Makanin's search algorithm proceeds using a subroutine called `transport`, which involves finding

two occurrences of the same variable and copying the bases contained within the column span of one base, called the *carrier*, into the position of the other, called the *dual*. In order to ensure full propagation, the conventional ordering is to always copy from the leftmost-reaching variable base which has not yet been copied from; ties are resolved by choosing the widest variable base. So a base  $(x, [1, 3])$  would take precedence over  $(x, [2, 8])$  due to extending further to the left, but  $(y, [1, 4])$  would take precedence over  $(x, [1, 3])$  due to being wider while having the same left boundary. Any variable base copied in this manner can itself be removed from the worklist. When we copy  $(x, [1, 3])$  as part of handling  $(y, [1, 4])$ , that must include copying any bases within  $x$ 's span because they are also contained in  $y$ 's span.

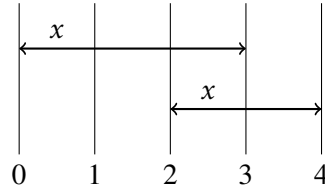
The simple case for transport is when the carrier and dual bases span the same number of columns. Then each base copied from the  $m^{\text{th}}$  through  $n^{\text{th}}$  columns of the carrier can move to the  $m^{\text{th}}$  through  $n^{\text{th}}$  columns of the dual. Consider the example of  $(++ [1] x) \doteq (++ x [1])$  with the variable bases overlapping. The carrier base is  $(x, [0, 2])$ , and the dual is  $(x, [1, 3])$ . The base  $(1, [0, 1])$ , occupying the first column of the carrier, is copied to produce  $(1, [1, 2])$ , which occupies the first column of the dual.



The situation is trickier if the carrier and dual have different column widths. Perhaps one occurrence of  $x$  spans three columns, while another spans four. In order to unify the four components of one occurrence with the three components of the other, we must choose a finer-grained column partition of that other occurrence. If we have already identified three component boundaries, it is safe to keep them; we only need to choose one of the three columns to split into two columns. The general case of inserting new boundaries into one base to match the width of another one requires a nondeterministic choice similar to constructing the initial generalized equation. The choice is how to interleave new column boundaries among the column boundaries already present within the narrower base. Since there is no possibility of coinciding old and new boundaries, this corresponds to choosing from N-E lattice paths, which are counted by combinations, rather than the N-E-NE paths.

We may encounter one further complication: The carrier and dual may partially overlap in addition to having different widths. Then some of the boundaries we introduce in order to accommodate boundaries between

copied components might fall within the intersection of the carrier's and dual's intervals. For example, a generalized equation with two  $x$  bases arranged as below has two possible ways to map the carrier's interval to the dual's interval.



One of those ways maps boundary 2, the carrier's second intermediate column boundary, to boundary 3, the only already existing intermediate column boundary in the dual. Then boundary 1, the carrier's other intermediate boundary, must therefore be mapped to a *new* boundary created by splitting the first column of the dual. Since the first column of the dual is also the last column of the carrier, splitting that column widens the dual from 2 to 3 columns but also widens the carrier from 3 to 4. Since they still have different widths, we need to introduce one more column boundary, this time in the last column of the dual. In the general case, the number of extra boundaries we introduce must match the number we initially put into the intersection, adding another choice with options counted by combinations.

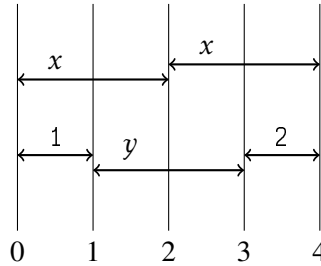
Pruning the search tree arising from the nondeterministic choice of boundary alignments is done by considering the population of each sequence element mentioned in the equation in each individual column of the generalized equation. For each variable  $e_i$  and column spanning the interval  $[j, j + 1]$ , we consider a natural-valued variable  $C_{i,j}$  to represent how many times  $e_i$  appears in column  $j$ .

A sequence variable with multiple occurrences identifies two or more intervals which must agree as to the population of each element. If we have bases  $(x, [0, 3])$  and  $(x, [2, 4])$ , the number of times  $e_1$  appears in the interval  $[0, 3]$  must be the same as the number of times it appears in the interval  $[2, 4]$ —that is,  $C_{1,0} + C_{1,1} + C_{1,2} = C_{1,2} + C_{1,3}$ . The same holds for  $e_2, e_3, \text{etc.}$  (as many elements as appear as literals in the original equation). For each single-element base  $(e_i, [j, j + 1])$  we can also require that  $C_{i,j} = 1$  and for each  $i' \neq i$ ,  $C_{i',j} = 0$ . We also require the sum of all elements' populations in a single column to be at least one because the column must be occupied by something. For a sequence equation using elements  $e_0$  through  $e_k$ , we require in each column  $j$  that  $\sum_{i=1}^k C_{i,j} \geq 1$ .

Since the above are all linear constraints on natural-valued variables, the pruning check is an integer linear programming problem. Once we have completed all transport steps, there is no need to introduce any additional column boundaries. We can further restrict each column's total population to be exactly 1 by requiring  $\sum_{i=1}^k C_{i,j} = 1$ . A satisfying

assignment for this ILP instance must choose for each  $j$  exactly one  $i$  so that  $C_{i,j} = 1$ . The rightmost occurrence of each variable contains all extra information discovered via transport, so those columns' ILP variables identify the elements that make up that sequence variable's assignment.

Revisiting the example equation  $(++ \ x \ x) \doteq (++ \ [1] \ y \ [2])$ , the only solvable alignment places the boundary between the two  $x$  occurrences somewhere in the middle of  $y$ , producing this arrangement:



Constructing the ILP instance for this generalized equation requires the following variables:

- $C_{1,0}$  through  $C_{1,3}$ : occurrences of 1 in each column
- $C_{2,0}$  through  $C_{2,3}$ : occurrences of 2 in each column
- $C_{1,x}$  and  $C_{1,y}$ : occurrences of 1 in  $x$  and  $y$
- $C_{2,x}$  and  $C_{2,y}$ : occurrences of 2 in  $x$  and  $y$

We know that any column must be populated by either 1 or 2, but a single column may turn out to represent multiple sequence elements in the eventual solution. So we include equations

$$C_{1,0} + C_{2,0} \geq 1 \quad = \quad C_{1,1} + C_{2,1} \geq 1$$

$$C_{1,2} + C_{2,2} \geq 1 \quad = \quad C_{1,3} + C_{2,3} \geq 1$$

The fact that  $x$  spans the first two columns means the 1-population of  $x$  must be the total 1-populations of the first two columns; the same holds for 2. We also have analogous constraints arising from the second occurrence of  $x$ , which spans the last two columns.

$$C_{1,0} + C_{1,1} = C_{1,2} + C_{1,3} = C_{1,x}$$

$$C_{2,0} + C_{2,1} = C_{2,2} + C_{2,3} = C_{2,x}$$

The element-population equations for  $y$  turn out not to constrain the ILP solution because there is only one occurrence of  $y$ . Finally, the first and last columns each contain a single-element base, constraining their populations for all elements. Column 0 must contain exactly one 1 and no 2, while column 3 must contain exactly one 2 and no 1.

$$C_{1,0} = 1 \quad C_{2,0} = 0$$

$$C_{1,3} = 0 \quad C_{2,3} = 1$$

This check rules out obviously unsolvable situations such as two different sequence element bases coinciding. It also rules out cases where one occurrence of a variable completely engulfs another occurrence of the same variable with room to spare. The extra columns spanned by the wider base must contain some element, which will then have different populations in the narrow and wide bases' intervals.

This ILP instance is satisfiable, so Makanin's algorithm considers this alignment of variable boundaries to be acceptable. In fact it has multiple solutions, although Makanin's algorithm will not explore them all. After a round of transport, the 1 base is copied from column 0 into column 2, and the updated ILP instance fixes  $C_{1,2}$  and  $C_{2,2}$ , leaving no room for choice in  $C_{1,1}$  and  $C_{2,1}$ .

### 7.2.3 Integrating a Dimension-Theory Solver

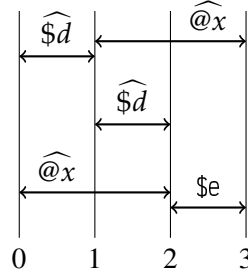
Although Makanin's algorithm was originally meant for finitely generated semigroups, any single equation can only mention finitely many generators. So it is not much trouble to use the algorithm with an infinite generator set. The real snag in Remora's use case is that array shapes may perform arithmetic on individual dimensions. While the previous example  $(++ \ x \ [3]) \doteq (++ \ [3] \ x)$  can be solved with any  $x \in 3^*$ , solving equations on shapes requires a way to recognize that an equation like  $(++ \ x \ [(+ \ q \ 2)]) \doteq (++ \ [(+ \ 1 \ r)] \ x)$  needs both  $x \in r^*$  and  $q + 1 \doteq r$ . Furthermore, the shape constraints that arise in type inference have both universal and existential variables. The above equation is unsolvable if  $r$  is universally quantified (there is no suitable  $q$  in the case where  $r$  is 0). On the other hand, if  $x$  and  $r$  are existentially quantified while  $q$  is universally quantified, the equation is solvable, and we would want the solver to tell us that  $r$  can be instantiated with  $(+ \ 1 \ q)$ .

Because Makanin's algorithm selects an alignment of equation components and then traces through the implications of that alignment, it offers a convenient hook for integrating a separate decision procedure for dealing with generator elements themselves. Keeping the generator decision procedure separate from the sequence decision procedure relies on strict stratification between sequences and sequence elements, like constructing Shapes from Dims in Remora.

The result is  $\text{Makanin}(T)$ , a sequence-equation decision procedure parameterized over a theory of equality for sequence elements, which can more flexibly integrate with a  $\text{DPLL}(T)$  solver or ILP modulo theories solver [60, 71]. Although Remora chooses the theory  $T$  to be Presburger arithmetic, the reasoning which identifies aliased columns in a generalized equation is agnostic to the particular choice of  $T$ . Similar to a lazy  $\text{DPLL}(T)$  solver querying a  $T$ -solver after constructing a plausible propositional model of an input formula,  $\text{Makanin}(T)$  uses string-equation

reasoning to construct a candidate model of a string equation over  $T$  terms and then asks whether that model is  $T$ -consistent.

To generalize from Makanin's algorithm to  $\text{Makanin}(T)$ , we augment Makanin's algorithm with a pass over the generalized equation which looks for sequence-element bases occurring at equivalent locations *within* different occurrences of the same sequence variable. For example, after one round of transport, the shape equation  $(++ [\widehat{\$d}] \widehat{@x}) \doteq (++ \widehat{@x} [\$e])$  is updated to



Rather than creating separate ILP variables for the populations of  $\widehat{\$d}$  and  $\widehat{\$e}$  in various portions of the shape equation, we conclude that  $\widehat{\$d}$  and  $\widehat{\$e}$  must themselves be equal. That is, the minimal equivalence relation which agrees with the boundary alignment we have chosen includes  $\widehat{\$d} \doteq \$e$ . When constructing an ILP instance to check viability, each population variable is associated with an *equivalence class* of sequence generators rather than an individual generator term.

Makanin's transport procedure thus leads to a conjunction of equalities on sequence generators which must be valid in order for the original equation on sequences to be valid. For Remora's type inference, this equivalence relation is interpreted as a conjunction of linear equations on  $\text{Dim}s$ , but we still have a mix of universally and existentially quantified  $\text{Dim}$  variables. Presburger's procedure for eliminating the universal variables rewrites each existential in terms of the universals, effectively constructing the Skolem functions needed in order to elaborate the Remora program.

After this rewriting, we have a strictly universal constraint which can be handed off as a validity query to an integer linear programming solver. This is actually encoded as an unsatisfiability query about the negation of the quantifier-free portion of the constraint. If the ILP solver says the constraint is valid (*i.e.*, its negation is unsatisfiable), then the Skolem functions derived from eliminating the existential  $\text{Dim}$  variables give their proper instantiation in the elaborated Remora program. Otherwise, the ILP solver returns a satisfying assignment for the negated constraint, which corresponds to a counterexample to the original validity query and thus to that particular alignment of the original sequence equation.

## 7.3 GENERALIZING TO A MIXED-PREFIX FRAGMENT

Additional work is needed to handle Remora’s use case. Makanin’s algorithm is meant for solving an equation with only existential quantification, but a Remora function may be polymorphic in cell shape. When synthesizing a type for the body of a cell-polymorphic function, we may encounter shape constraints that mention the function’s shape arguments, which are universal variables, and frame shapes, which are existential. We must account for the possibility that an existential variable stands for something built from a universal variable. Consider the following function:

```
(λ ((n [Int $d @s]))
  (add1 (reverse n)))
```

Both function applications require resolving an existential shape variable using the universal variable  $s$ . For `reverse`, which expects an argument of type `[%t $l @r]`—*i.e.*, an array of `%t` with leading axis  $l$  and remaining axes  $@r$ —we will have to find how the argument type `[Int %d s]` can match the desired type. First, an existential atom type `%t` must resolve to `Int`. We then ask the shape theory solver to solve:

$$\forall \$d, @s. \exists \hat{\$l}, \hat{@r}. \{++ [\hat{\$l}] \hat{@r}\} \doteq \{++ [\$d] @s\}$$

This formula is of course provable by the witness  $\hat{\$l} \mapsto \$d$  and  $\hat{@r} \mapsto @s$ , but we will need to adapt Makanin’s algorithm so that it can work with universal sequence variables like  $@s$  and also invoke a dimension theory solver to discover the solution using  $\$d$ . With the latter problem handled by Subection 7.2.3, we focus now on the universal shape variables.

Accommodating the universal shape variables in a single equation is straightforward if the quantifier prefix is limited to several universals followed by several existentials. This works because the key decisions made during Makanin’s search are about how to split variables’ meanings into subsequences that are partially or fully covered by other variables. If a candidate solution depends at all on the *internal* structure of the sequence a universal variable happens to stand for then that candidate is ruled out: the actual value given to a universal variable might not have that required structure. The treatment we give to sequence element bases upholds exactly that requirement. By never allowing a column boundary to appear inside a certain base, we ensure that any variable base which overlaps it must overlap it completely. Since Makanin’s algorithm is agnostic as to the size of the alphabet from which sequences are generated, we can freely add new element IDs to represent universal sequence variables.

A similar principle applies in generalizing from the  $\forall^*\exists^*$  fragment to the  $(\forall\exists)^*$ . We ignore the ordering of quantifiers during the Makanin search and afterward filter out potential solutions based on the associated

dimension equivalence relations. If an equation involving an existential dimension variable makes it depend on a universal which came into scope after that existential, then we cannot produce a valid solution for that existential. The Skolem function for an existential cannot take as input the values of universal variables bound farther down.

With this encoding, we do sacrifice some expressive power. In the  $\exists^*$  (existential-only) fragment of the theory of sequences, any boolean combination of equations can be encoded as a single equation. This encoding does not work in the  $(\forall\exists)^*$  fragment. Although a single equation is invalid if its truth depends on the internals of a universal, a disjunction of several such equations may be valid. For example, in sequences over the alphabet  $\{0, 1\}$ , the formula

$$\forall x. \exists y. x \doteq \{++ [0] y\} \vee x \doteq \{++ [1] y\} \vee x \doteq []$$

is valid even though the individual equations are not. In order to avoid the undecidability of the full mixed-prefix theory, we must limit shape constraints to the  $(\forall\exists)^*. \wedge$  fragment.



## EVALUATION

## 8.1 ELABORATION SOUNDNESS

Since there is no dynamic semantics for implicit Remora, and explicit Remora’s type-driven semantics is not readily adapted to a language without explicit types, the only behavior attributed to implicitly typed code is the result of elaborating to explicitly typed code and then executing it. This does not offer a clear way to define correctness of elaboration: rather than *preserving* the semantics of implicitly typed code, elaboration *is* the semantics.

However, the important decisions elaboration makes are about the types of functions’ input cells, the types of actual arguments, and how cell-polymorphic functions are instantiated. Giving some function the wrong cell type, whether by elaborating a cell-rank annotation incorrectly or by choosing the wrong type or index arguments for instantiation, produces a function with different lifting behavior than it ought to have. Some combinations of arguments that ought to be acceptable will be rejected as ill-typed. Many aggregation and reduction functions with `all`-ranked input cells produce differently shaped (*i.e.*, differently typed) output if instantiated incorrectly.

Since the tricky bookkeeping handled by elaboration has observable effects during type checking (according to the typing rules of Chapter 4) and even during execution, we can feel confident that elaboration behaves as desired based on the fact that an elaborated term generated for any arbitrary well typed implicit Remora term not only is well typed but has the *same* type ascribed to the original implicitly typed term. Establishing that fact is the goal of this section.

One potential worry appears when we consider stray unsolved existential type and index variables appearing in elaborated code. They identify situations in which multiple possible types could be given to some implicit Remora program. Excepting terms with multiple possible types from the above guarantee would make too few promises about useful terms. Even `(1 1)+` has as many types as there are vector lengths.<sup>29</sup> That is too few promises, not just for user confidence but too few for a useful induction hypothesis. A term which uses code with multiple possible types might rely on it having some particular type. So the elaboration soundness claim in this section states that any interpretation of existential variables which is consistent with the output environment and constraint archive gives elaborated code whose type matches that of the original implicit Remora code.

<sup>29</sup> It should ideally have  $\aleph_0$  possible types, according to the formal semantics, but a realistic implementation of Remora would likely have to limit the number of possibilities to roughly the `SIZE_MAX` of a reasonably modern C implementation.

In order to prove this, we need some lemmas ensuring good behavior in managing the environment and constraint archive. It wouldn't do to have existential variables' eventual chosen meanings or algebraic constraints on those meanings being forgotten too soon. We would be unable to guarantee that the value chosen as the solution for some dimension variable satisfies all the requirements that the surrounding elaborated code places on it. We also do not want any universal variables leaking into the output environment in a way that allows explicit Remora's kinding and sorting rules to be subverted.

After we have established that the environment and archive are managed properly, we move on to investigating the bidirectional rules' elaboration method itself. The coercing contexts generated by the subtyping and instantiation judgments as witnesses to the conclusion that some type is usable where some other is required must be proven to actually witness that conclusion. Most of the decisions about instantiating cell-polymorphic functions are found in the subtyping rules' use of type and index application as coercions.

Finally, with the generated coercions shown to be true witnesses of subtyping, the final theorem of this section establishes that interpretation of the existential variables remaining in an elaborated term that obeys the output environment and constraint archive gives the original implicit Remora term the same type as the elaborated explicit Remora term.

### 8.1.1 Environment bookkeeping

The “growth” relation for well-formed environments, written as  $\bar{\Gamma} \leq \bar{\Gamma}'$ , describes the idea of adding new information to the environment as it is discovered during type inference, while preserving any discoveries already made. This includes introducing new variables and adding solutions to already existing variables. It can be generated as the reflexive, transitive closure of the following rules:

$$\begin{array}{l}
\bar{\Gamma}_1, \bar{\Gamma}_2 \leq \bar{\Gamma}_1, \mathcal{X}, \bar{\Gamma}_2 \\
\bar{\Gamma}_1, \bar{\Gamma}_2 \leq \bar{\Gamma}_1, \mathcal{S}, \bar{\Gamma}_2 \\
\bar{\Gamma}_1, \bar{\Gamma}_2 \leq \bar{\Gamma}_1, \hat{\mathcal{X}}, \bar{\Gamma}_2 \\
\bar{\Gamma}_1, \bar{\Gamma}_2 \leq \bar{\Gamma}_1, \hat{\mathcal{S}}, \bar{\Gamma}_2 \\
\bar{\Gamma}_1, \hat{\mathcal{X}}, \bar{\Gamma}_2 \leq \bar{\Gamma}_1, \hat{\mathcal{X}} \mapsto \tau, \bar{\Gamma}_2, \quad \text{where } \bar{\Gamma}_1 \vdash \tau :: \text{Kind}[\mathcal{X}] \\
\bar{\Gamma}_1, \hat{\mathcal{S}}, \bar{\Gamma}_2 \leq \bar{\Gamma}_1, \hat{\mathcal{S}} \mapsto \iota, \bar{\Gamma}_2, \quad \text{where } \bar{\Gamma}_1 \vdash \iota :: \text{Sort}[\mathcal{S}]
\end{array}$$

Further, we extend the growth relation to define a “completion” relation, which specifies that the enlarged environment must be self-contained in its solutions for existential type and index variables. In some situations, we can allow  $\hat{f} \mapsto (-\> (\hat{i}) \hat{o})$  on its own, but in other situa-

tions, we also require full solutions for  $\widehat{\textcircled{1}}$  and  $\widehat{\textcircled{0}}$ . We say  $\bar{\Gamma}'$  completes  $\bar{\Gamma}$ , written as  $\bar{\Gamma} \widehat{\leq} \bar{\Gamma}'$ , if  $\bar{\Gamma} \leq \bar{\Gamma}'$  and  $\bar{\Gamma}'$  contains a solution for every existential variable appearing in it.

The above rules all preserve the order of environment entries. Since variables are not reused, there can be no duplicate entries. An environment can therefore be “split” at a particular entry, with each growth step belonging to one side of the split. Being able to reason about this splitting of environments will be important for ensuring that dropping environment entries in the course of a judgment derivation does not affect earlier portions of the environment.

**Lemma 8.1.1** (Environment splitting). *If  $\bar{\Gamma}_l, \gamma, \bar{\Gamma}_r \leq \bar{\Gamma}'_l, \gamma, \bar{\Gamma}'_r$ , then  $\bar{\Gamma}_l \leq \bar{\Gamma}'_l$  and  $\bar{\Gamma}_r \leq \bar{\Gamma}'_r$ .*

*Proof.* Because  $\leq$  is the reflexive, transitive closure of the above rules, for any two related environments  $\bar{\Gamma}_1 \leq \bar{\Gamma}_2$ , it must be possible to convert  $\bar{\Gamma}_1$  into  $\bar{\Gamma}_2$  by applying some number of transformation steps corresponding to those rules. We can thus argue inductively on the sequence of growth steps which would turn  $\bar{\Gamma}_l, \gamma, \bar{\Gamma}_r$  into  $\bar{\Gamma}'_l, \gamma, \bar{\Gamma}'_r$ .

For the base case, zero steps are needed, implying that  $\bar{\Gamma}_l, \gamma, \bar{\Gamma}_r = \bar{\Gamma}'_l, \gamma, \bar{\Gamma}'_r$ . Since no variables are reused,  $\gamma$  can only appear once. Therefore  $\bar{\Gamma}_l = \bar{\Gamma}'_l$ , and  $\bar{\Gamma}_r = \bar{\Gamma}'_r$ .

In the inductive case, each of the above rules modifies one entry or adds one entry. For a modified entry, if the original appears in  $\bar{\Gamma}_l$ , then the new version appears in  $\bar{\Gamma}'_l$ . Then  $\bar{\Gamma}_l \leq \bar{\Gamma}'_l$ , and  $\bar{\Gamma}_r = \bar{\Gamma}'_r$ . Similarly, if the original entry appears in  $\bar{\Gamma}_r$ , then the new version appears in  $\bar{\Gamma}'_r$ , implying  $\bar{\Gamma}_l = \bar{\Gamma}'_l$ , and  $\bar{\Gamma}_r \leq \bar{\Gamma}'_r$ . For a newly added entry, it must either appear in  $\bar{\Gamma}'_l$  while not appearing in  $\bar{\Gamma}_l$ , so  $\bar{\Gamma}_l \leq \bar{\Gamma}'_l$ , and  $\bar{\Gamma}_r = \bar{\Gamma}'_r$ , or appear in  $\bar{\Gamma}'_r$  while not appearing in  $\bar{\Gamma}_r$ , so  $\bar{\Gamma}_l = \bar{\Gamma}'_l$ , and  $\bar{\Gamma}_r \leq \bar{\Gamma}'_r$ .  $\square$

**Lemma 8.1.2** (Environment monotonicity for instantiation). *Let  $\bar{\Gamma}; \Phi \vdash \tau :: k$ . Given one of*

- $\bar{\Gamma}; \Phi \vdash \hat{\mathcal{X}} :: \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbf{C}$
- $\bar{\Gamma}; \Phi \vdash \tau :: \hat{\mathcal{X}} \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbf{C}$

*then  $\bar{\Gamma} \leq \bar{\Gamma}'$ , and  $\Phi \subseteq \Phi'$ .*

*Proof sketch.* We use induction on the instantiation judgment derivation.  $\square$

**Lemma 8.1.3** (Environment monotonicity for subtyping). *If  $\bar{\Gamma}; \Phi \vdash \tau_l \leq \tau_h \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbf{C}$ , then  $\bar{\Gamma} \leq \bar{\Gamma}'$ , and  $\Phi \subseteq \Phi'$ .*

*Proof sketch.* We use induction on the subtyping judgment derivation.  $\square$

**Lemma 8.1.4** (Environment monotonicity for bidirectional judgments). *Given one of*

- $\bar{\Gamma}; \Phi \vdash \bar{t} \Leftarrow \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow t$
- $\bar{\Gamma}; \Phi \vdash \bar{t} \Rightarrow \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow t$
- $\bar{\Gamma}; \Phi \vdash (\bar{t}_f : \tau_f) \bullet [\bar{t}_a \dots] \Rightarrow \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow t$ ,  
where  $\bar{t} = (\bar{t}_f \ \bar{t}_a \dots)$

then  $\bar{\Gamma} \leq \bar{\Gamma}'$ , and  $\Phi \subseteq \Phi'$ .

*Proof sketch.* We use induction on the bidirectional type derivation.  $\square$

Showing the proper behavior of the generated coercion code will require some more environment bookkeeping. In addition to the above results showing that we can only gain—not lose—information about existential variables, we need to guarantee good behavior regarding term variables as well as universal type and index variables (*i.e.*, those which arise from  $\forall$  and  $\Pi$  types in the program). Specifically, we need to ensure that these variables do not leak into the output environment, which would allow free term, type, and index variables to appear in generated code. Since we have previously proven monotonic growth—*i.e.*, no variables are removed—we now only need to show that no non-existential variables are added.

**Lemma 8.1.5** (Variable scoping for instantiation). *Given one of*

- $\bar{\Gamma}; \Phi \vdash \mathcal{X} : \leq \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbf{C}$
- $\bar{\Gamma}; \Phi \vdash \tau : \leq \mathcal{X} \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbf{C}$

then all of the following hold:

- $TB \llbracket \bar{\Gamma} \rrbracket = TB \llbracket \bar{\Gamma}' \rrbracket$
- $KB \llbracket \bar{\Gamma} \rrbracket = KB \llbracket \bar{\Gamma}' \rrbracket$
- $SB \llbracket \bar{\Gamma} \rrbracket = SB \llbracket \bar{\Gamma}' \rrbracket$

*Proof sketch.* We use induction on the instantiation judgment derivation.  $\square$

**Lemma 8.1.6** (Variable scoping for subtyping). *If  $\bar{\Gamma}; \Phi \vdash \tau_l \leq \tau_h \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbf{C}$ , then all of the following hold:*

- $EVars \llbracket \bar{\Gamma} \rrbracket = EVars \llbracket \bar{\Gamma}' \rrbracket$
- $TVars \llbracket \bar{\Gamma} \rrbracket = TVars \llbracket \bar{\Gamma}' \rrbracket$
- $IVars \llbracket \bar{\Gamma} \rrbracket = IVars \llbracket \bar{\Gamma}' \rrbracket$

*Proof sketch.* We use induction on the subtyping judgment derivation.  $\square$

**Lemma 8.1.7** (Variable scoping for bidirectional judgments). *Given one of*

- $\bar{\Gamma}; \Phi \vdash \bar{t} \Leftarrow \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow t$
- $\bar{\Gamma}; \Phi \vdash \bar{t} \Rightarrow \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow t$
- $\bar{\Gamma}; \Phi \vdash (\bar{t}_f : \tau_f) \bullet [\bar{t}_a \dots] \Rightarrow \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow t$

*then all of the following hold:*

- $EVars \llbracket \bar{\Gamma} \rrbracket = EVars \llbracket \bar{\Gamma}' \rrbracket$
- $TVars \llbracket \bar{\Gamma} \rrbracket = TVars \llbracket \bar{\Gamma}' \rrbracket$
- $IVars \llbracket \bar{\Gamma} \rrbracket = IVars \llbracket \bar{\Gamma}' \rrbracket$

*Proof sketch.* We use induction on the bidirectional typing derivation.  $\square$

### 8.1.2 Elaborated code

The phrasing of elaboration soundness includes some trickery related to underdetermined type annotations. Consider applying a curried function structured like  $\lambda x. \lambda y. x$  to a single argument. This is a polymorphic function with two type arguments, but only one is actually used. The elaborated code will thus have a stray existential type variable representing the underdetermined type argument. Any type works, but the stray existential does not appear in the output environment. So we cannot just promise that the output environment will suffice to type check elaborated code—we *know* it won't be enough in certain cases. Instead, the coercion soundness theorems must only deal with environments which include solutions for those stray existential variables. To do so, instead of claiming that the output environment  $\bar{\Gamma}'$  contains everything needed to type check the elaborated term, the soundness theorem states that any completion of  $\bar{\Gamma}'$  will do so. Specifying that the environment must solve all existential variables in the output code also ensures that only syntactically valid explicit terms are claimed as acceptable final output (recall, existential type and index variables are not part of explicitly typed Remora).

For elaboration soundness theorems, where the output is a term  $t$  instead of a coercion  $C$ , we require that  $\bar{\Gamma}''$  leave no existential variables in  $t$ . Under that assumption,  $\bar{\Gamma}''$  must also solve all existential variables in any coercion used to build  $t$ . So we are free to invoke the coercion soundness theorems in cases where a premise produces a coercion rather than a full term.

Proving the desired soundness result calls for formally stating the behavior of the coercion code generated by the metafunctions used in

the instantiation and subtyping rules. We say that  $\mathbf{C}$  coerces from type  $\tau_l$  to type  $\tau_h$  under the environment  $\bar{\Gamma}$  if for any explicit term  $t$ ,

$$TB \llbracket \bar{\Gamma} \rrbracket ; KB \llbracket \bar{\Gamma} \rrbracket ; SB \llbracket \bar{\Gamma} \rrbracket \vdash \bar{\Gamma} \llbracket t \rrbracket : \bar{\Gamma} \llbracket \tau_l \rrbracket$$

implies

$$TB \llbracket \bar{\Gamma} \rrbracket ; KB \llbracket \bar{\Gamma} \rrbracket ; SB \llbracket \bar{\Gamma} \rrbracket \vdash \bar{\Gamma} \llbracket \mathbf{C}[t] \rrbracket : \bar{\Gamma} \llbracket \tau_h \rrbracket$$

**Lemma 8.1.8** (Lift coercion). *If  $\mathbb{A}$  coerces from  $\mathfrak{T}_l$  to  $\mathfrak{T}_h$  under  $\bar{\Gamma}$ , with  $KB \llbracket \bar{\Gamma} \rrbracket ; SB \llbracket \bar{\Gamma} \rrbracket \vdash \bar{\Gamma} \llbracket \mathfrak{T}_l \rrbracket :: \text{Atom}$  and  $KB \llbracket \bar{\Gamma} \rrbracket ; SB \llbracket \bar{\Gamma} \rrbracket \vdash \bar{\Gamma} \llbracket \mathfrak{T}_h \rrbracket :: \text{Atom}$  then for any shape  $I$  which is well-formed under  $\bar{\Gamma}$ ,  $\text{LiftC}_{\mathfrak{T}_l} \llbracket \mathbb{A} \rrbracket$  coerces from  $(A \mathfrak{T}_l I)$  to  $(A \mathfrak{T}_h I)$ .*

*Proof.* T-LAM ascribes the type

$$\bar{\Gamma} \llbracket (-> ((A \mathfrak{T}_l (\text{shape}))) (A \mathfrak{T}_h (\text{shape}))) \rrbracket$$

to the function constructed for the lift coercion. With T-ARRAY and T-APP, applying such a function to an argument of type  $\bar{\Gamma} \llbracket (A \mathfrak{T}_l I) \rrbracket$  will have frame shape  $I$ , producing a result of type  $\bar{\Gamma} \llbracket (A \mathfrak{T}_h I) \rrbracket$ .  $\square$

**Lemma 8.1.9** (Each coercion). *If  $\mathbb{E}$  coerces from  $T_l = (A \mathfrak{T}_l I_l)$  to  $T_h = (A \mathfrak{T}_h I_h)$  under  $\bar{\Gamma}$ , with  $KB \llbracket \bar{\Gamma} \rrbracket ; SB \llbracket \bar{\Gamma} \rrbracket \vdash \bar{\Gamma} \llbracket T_l \rrbracket :: \text{Array}$  and  $KB \llbracket \bar{\Gamma} \rrbracket ; SB \llbracket \bar{\Gamma} \rrbracket \vdash \bar{\Gamma} \llbracket T_h \rrbracket :: \text{Array}$ , then for any shape  $I_f$  which is well-formed under  $\bar{\Gamma}$ ,  $\text{EachC}_{T_l} \llbracket \mathbb{E} \rrbracket$  coerces from type  $(A \mathfrak{T}_l ( ++ I_f I_l ))$  to type  $(A \mathfrak{T}_h ( ++ I_f I_h ))$ .*

*Proof.* T-LAM ascribes the type  $\bar{\Gamma} \llbracket (-> ((A \mathfrak{T}_l I_l)) (A \mathfrak{T}_h I_h)) \rrbracket$  to the function constructed for the coercion. With T-ARRAY and T-APP, applying such a function to an argument of type  $\bar{\Gamma} \llbracket (A \mathfrak{T}_l ( ++ I_f I_l )) \rrbracket$  will have frame shape  $I_f$ . So the result type is  $\bar{\Gamma} \llbracket (A \mathfrak{T}_h ( ++ I_f I_h )) \rrbracket$ .  $\square$

**Lemma 8.1.10** (Function coercion). *If the contexts  $\mathbb{E}_i \dots$  respectively coerce from  $T'_i \dots$  to  $T_i \dots$  under  $\bar{\Gamma}$  and  $\mathbb{E}_o$  from  $T_o$  to  $T'_o$ , with all of  $T_i \dots, T'_i \dots, T_o, T'_o$  kinded as Array under  $\bar{\Gamma}$ , then for any shape  $I_f$  which is well-formed under  $\bar{\Gamma}$ , the coercion*

$$\text{FnC}_{(-> (\tau_i \dots) \tau_o) \rightarrow (-> (\tau'_i \dots) \tau'_o)} \llbracket (\mathbb{E}_i \dots); (\mathbb{E}_o) \rrbracket$$

*coerces from  $(A (-> (\tau_i \dots) \tau_o) I_f)$  to  $(A (-> (\tau'_i \dots) \tau'_o) I_f)$ .*

*Proof.* The outer function in the generated code binds  $x_f$  at the type  $(A (-> (\tau_i \dots) \tau_o) (\text{shape}))$ . Its result is a scalar containing a function with input types  $\tau'_i \dots$ . In order to show that the entire term has the correct type, we must show that the body of this inner function has type  $\tau_o$  in the environment extended with  $x_f, x_i \dots$ . Since the variables  $x_i \dots$  are bound at types  $\tau'_i \dots$ , applying the input coercions  $\mathbb{E}_i \dots$  to them

produces results whose respective types are  $\tau_i \dots$ . The application of  $x_f$  to these coerced inputs produces a result with type  $\tau_o$  (*n.b.*, the frame shape here is scalar). Then coercing that application form's output with  $\mathbb{E}_o$  produces a result with type  $\tau'_o$ .

So the outer function has input type  $(A (-> (\tau_i \dots) \tau_o) (\text{shape}))$  and output type  $(A (-> (\tau'_i \dots) \tau'_o) (\text{shape}))$ . Filling the hole in the context with a term typed as  $(A (-> (\tau_i \dots) \tau_o) I_f)$  produces an application form whose principal frame shape is  $I_f$  and output cell type is  $(A (-> (\tau'_i \dots) \tau'_o) (\text{shape}))$ . Therefore the entire term built by applying the function coercion as type  $(A (-> (\tau'_i \dots) \tau'_o) I_f)$ .  $\square$

Since limitations on atom-level computation often require subtyping rules that match on arrays containing atoms of a particular form, we occasionally need to chain multiple layers of context in order to get the right coercion behavior. Intuitively, coercing contexts compose like functions: as long as the output type of one matches the input type of the next, combining them produces a coercion from the first input type to the last output type.

**Lemma 8.1.11** (Coercion composition). *Under a given environment  $\bar{\Gamma}$ , if  $\mathbf{C}_0$  which coerces from  $\tau_0$  to  $\tau_1$  and  $\mathbf{C}_1$  which coerces from  $\tau_1$  to  $\tau_2$ , then  $\mathbf{C}_1[\mathbf{C}_0]$  coerces from  $\tau_0$  to  $\tau_2$ .*

*Proof.* If  $TB \llbracket \bar{\Gamma} \rrbracket ; KB \llbracket \bar{\Gamma} \rrbracket ; SB \llbracket \bar{\Gamma} \rrbracket \vdash \bar{\Gamma} \llbracket t \rrbracket : \bar{\Gamma} \llbracket \tau_0 \rrbracket$ , then  $\mathbf{C}_0$ 's coercion behavior means that  $TB \llbracket \bar{\Gamma} \rrbracket ; KB \llbracket \bar{\Gamma} \rrbracket ; SB \llbracket \bar{\Gamma} \rrbracket \vdash \bar{\Gamma} \llbracket \mathbf{C}_0[t] \rrbracket : \bar{\Gamma} \llbracket \tau_1 \rrbracket$ . So  $\mathbf{C}_1$ 's coercion behavior then implies

$$TB \llbracket \bar{\Gamma} \rrbracket ; KB \llbracket \bar{\Gamma} \rrbracket ; SB \llbracket \bar{\Gamma} \rrbracket \vdash \bar{\Gamma} \llbracket \mathbf{C}_1[\mathbf{C}_0[t]] \rrbracket : \bar{\Gamma} \llbracket \tau_2 \rrbracket$$

$\square$

With utility lemmas in place, we now move on to the elaboration rules themselves. We finally have a use for the index equality archive  $\Phi$ . Many branches of the upcoming case analysis rely on the solver to produce an environment in which solutions of index variables entails the equality of particular shapes. If the archive returned from the solver is unsatisfiable, then there exists no assignment which satisfies all shape equalities required for the type derivation we are constructing. Failing to ensure shape equality means failing to ensure type equivalence. An instantiation or subtype rule in question cannot guarantee that it produces a coercion to the right type, and a bidirectional rule cannot guarantee that it produces an explicit term of the right type.

**Theorem 8.1.1** (Instantiation coercion). *Given all of*

- $\bar{\Gamma}; \Phi \vdash \hat{\mathcal{X}} :: \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbf{C}_t$
- $KB \llbracket \bar{\Gamma} \rrbracket ; SB \llbracket \bar{\Gamma} \rrbracket \vdash \tau :: k$

- $\bar{\Gamma}' \preceq \bar{\Gamma}''$
- $SAT \llbracket \Phi' \rrbracket$

then  $\mathbf{C}_t$  coerces from  $\bar{\Gamma}'' \llbracket \hat{\mathcal{X}} \rrbracket$  to  $\bar{\Gamma}'' \llbracket \tau \rrbracket$  and given all of

- $\bar{\Gamma}; \Phi \vdash \tau \leq: \hat{\mathcal{X}} \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbf{C}_t$
- $KB \llbracket \bar{\Gamma} \rrbracket; SB \llbracket \bar{\Gamma} \rrbracket \vdash \tau :: k$
- $\bar{\Gamma}' \preceq \bar{\Gamma}''$
- $SAT \llbracket \Phi' \rrbracket$

then  $\mathbf{C}_t$  coerces from  $\bar{\Gamma}'' \llbracket \tau \rrbracket$  to  $\bar{\Gamma}'' \llbracket \hat{\mathcal{X}} \rrbracket$ .

*Proof sketch.* We use induction on the instantiation derivation.  $\square$

**Theorem 8.1.2** (Subtyping coercion). *Given all of*

- $KB \llbracket \bar{\Gamma} \rrbracket; SB \llbracket \bar{\Gamma} \rrbracket \vdash \tau_l :: k$
- $KB \llbracket \bar{\Gamma} \rrbracket; SB \llbracket \bar{\Gamma} \rrbracket \vdash \tau_h :: k$
- $\bar{\Gamma}; \Phi \vdash \tau_l \leq \tau_h \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbf{C}_t$
- $\bar{\Gamma}' \preceq \bar{\Gamma}''$
- $SAT \llbracket \Phi' \rrbracket$

then  $\mathbf{C}_t$  coerces from  $\bar{\Gamma}'' \llbracket \tau_l \rrbracket$  to  $\bar{\Gamma}'' \llbracket \tau_h \rrbracket$ .

*Proof sketch.* We use induction on the subtyping derivation.  $\square$

The final step in proving soundness is the proof for the bidirectional judgments themselves.

The phrasing of the application judgment case is slightly odd. Rules for handling a function application step through the arguments one at a time, imagining that the function is curried. After each argument, the type we pretend the function has gets its arity decreased by one and its shape updated to the largest frame seen so far. This controlled self-deception in the bidirectional typing rules is to be justified by counterfactual reasoning in the upcoming proof: If the function application we are in the middle of typing really did have the function partially applied to the already handled arguments, then applying it to the remaining arguments would introduce no new problems.

In the proof itself, this is phrased as taking the  $e_f : \tau_f$  from the application judgment as a premise of the type derivation even though we know that it may be impossible to derive  $\tau_f$  as a type for  $e_f$ . The only time we do know this to be possible is when we have not yet considered



any arguments. Then  $\tau_f$  has the right input types for  $e_f$ . This arrangement gives the application judgment the stronger induction hypothesis it needs while allowing SYN:APP, the synthesis rule which invokes that judgment, to show that  $e_f : \tau_f$  is derivable and that the implication provided by that induction hypothesis has a useful consequent.

Alternatively, this proof could be done by stating the application judgment portion as  $e_f : \tau_f$  implying in this external logic that  $e_r : \tau_r$ . Of course, no function in Remora can have multiple arities. We could therefore throw out arity mismatch subcases by appealing to the meta-level principle *ex falso quodlibet*, but that's no fun.

**Theorem 8.1.3** (Elaboration soundness). *Given  $\bar{\Gamma}' \lesssim \bar{\Gamma}''$  with no existential variables appearing in  $\bar{\Gamma}'' \llbracket t \rrbracket$  and  $\text{SAT} \llbracket \Phi' \rrbracket$ , the following all hold:*

- $\bar{\Gamma}; \Phi \vdash \bar{t} \Leftarrow \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow t$  implies that  
 $TB \llbracket \bar{\Gamma}'' \rrbracket; KB \llbracket \bar{\Gamma}'' \rrbracket; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket t \rrbracket : \bar{\Gamma}'' \llbracket \tau \rrbracket$
- $\bar{\Gamma}; \Phi \vdash \bar{t} \Rightarrow \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow t$  implies that  
 $TB \llbracket \bar{\Gamma}'' \rrbracket; KB \llbracket \bar{\Gamma}'' \rrbracket; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket t \rrbracket : \bar{\Gamma}'' \llbracket \tau \rrbracket$
- $\bar{\Gamma}; \Phi \vdash (\bar{e}_f : \tau_f) \bullet [\bar{e}_a \dots] \Rightarrow \tau_r \dashv \bar{\Gamma}'; \Phi' \hookrightarrow e_r$  implies that  
 from  $TB \llbracket \bar{\Gamma}'' \rrbracket; KB \llbracket \bar{\Gamma}'' \rrbracket; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket e_f \rrbracket : \bar{\Gamma}'' \llbracket \tau_f \rrbracket$  we can  
 derive  $TB \llbracket \bar{\Gamma}'' \rrbracket; KB \llbracket \bar{\Gamma}'' \rrbracket; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket e_r \rrbracket : \bar{\Gamma}'' \llbracket \tau_r \rrbracket$

*Proof sketch.* We use induction on the bidirectional typing derivation.  $\square$

## 8.2 PRACTICAL USE

The plan laid out in Chapter 6 for inferring types of Remora programs calls for automatically generating the t-app and i-app forms needed to instantiate cell-polymorphic functions and also converting argument rank specifiers into cell types which reflect any aliasing of dimensions. The soundness theorem establishes that Remora's bidirectional typing does not do anything ill-typed, but it does not establish that it does anything at all. The following demonstration of type synthesis serves as complementary evidence for the bidirectional rules' effectiveness.

The corpus of code samples includes some synthetic examples—contrived specifically to highlight either tricky reasoning paths type synthesis must take or limitations of type synthesis requiring a type annotation—and adaptations of typical APL or J code for tasks like stencil computation or matrix multiplication.

Type inference's fundamental purpose is for user-written code to be more concise than the big, ugly terms demanded by explicit Remora. We cannot expect a completely annotation-free language, but the annotation burden turns out to be small in practice. In order to show the kind of code

a human is meant to write, each implicit Remora code sample is also presented alongside a version which uses the syntactic sugar for array construction, array types, and reranking. After a sketch of the process for synthesizing a type for the code sample, the fully elaborated version is shown.

<sup>30</sup> Recall, these are prefixed by a percent sign, borrowing a convention for marking symbols reserved for use by the internals of a language implementation.

While Remora’s formal semantics uses atoms to represent primitive operators,<sup>30</sup> here we use term-level variables assumed to be bound in a base environment to scalar arrays containing the corresponding primitive operators. So the variable `+` stands for the value `(array () %+)`. The function types associated with these variables are given in Figure 8.1, with implicit  $\forall$  and  $\Pi$  quantification for type and index variables appearing in the types.

### 8.2.1 Simple Application

```
(+ 1 2)                                (+ (array () 1)
                                       (array () 2))
```

All inference has to do in this example is ensure that argument types match up and identify the principal frame. We have an application form, which fits the `SYN:APP` rule. The first premise synthesizes a type for the function. Since `+` is bound in the base environment, `SYN:VAR` identifies its type. The second premise of `SYN:APP` invokes the application judgment, with `+` as the partly elaborated function—with the type discovered by `SYN:VAR`—and scalars `1` and `2` as arguments.

We already have a monomorphic function, so we can use `APP:FN*A`. The function’s first input type is a scalar integer, so we generate fresh existential shape variables  $\widehat{@af}$  and  $\widehat{@fe}$  to represent the argument’s frame and the frame extension the function will need in order to match the argument’s frame. We must check the argument against the frame-augmented version of the input type,  $(A \text{ Int } \widehat{@af})$ . `CHK:ARRAY` fits this goal. We check that the atom within, `1`, is an `Int`. Then we equate the frame-augmented input shape  $(++ \widehat{@af} (\text{shape}))$  with the array form’s shape  $(\text{shape})$ . In doing so, we discover that  $\widehat{@af}$  must be  $(\text{shape})$ , so we update its environment entry with that solution.

Having confirmed that the argument’s type is acceptable, we then find the function’s frame extension by constructing the expanded function array’s shape  $(++ \widehat{@fe} (\text{shape}))$  and equating it with the argument’s frame  $\widehat{@af}$ , which we just resolved as  $(\text{shape})$ . The equation  $(++ \widehat{@fe} (\text{shape})) \doteq (\text{shape})$  can only be satisfied by resolving  $\widehat{@fe}$  as  $(\text{shape})$ , so we update the environment accordingly. So the first argument is accepted, and its contribution to the application’s principal frame is known.

The final premise of `APP:FN*A` imagines that the function expects one argument less—as though it were curried and partially applied—and

```

+, -, * (-> ((A Int (shape))
            (A Int (shape)))
         (A Int (shape)))
+., *. (-> ((A Float (shape))
           (A Float (shape)))
        (A Float (shape)))
sqrt (-> ((A Float (shape))
         (A Float (shape)))
length (-> ((A &t (++ (shape $l) @c)))
          (A Int (shape)))
fst (-> (*t *t) *t)
transpose (-> ((A &t (shape $a $b)))
             (A &t (shape $b $a)))
iota/w (-> ((A &t @s)
           (A Int @s))
iota0 (-> ((A Int (shape))
          (A ( $\Sigma$  ($l)
             (A Int (shape $l))) (shape))))
rotate (-> ((A Int (shape))
           (A &t (++ (shape $l @c))))
        (A &t (++ (shape $l @c))))
reverse (-> ((A &t (++ (shape $l @c)))
            (A &t (++ (shape $l @c))))
reduce/0 (-> ((A (-> ((A &t @c)
                    (A &t @c))
                  (A &t @c)) (shape))
            (A &t @c)
            (A &t (++ (shape $l) @c)))
            (A &t @c))
reduce/L0 (-> ((A (-> ((A &t @c)
                    (A &t @c))
                  (A &t @c)) (shape))
            (A &t @c)
            (A &t (++ (shape $l) @f @c)))
            (A &t @f @c))

```

Figure 8.1: Base environment entries used for type synthesis and elaboration of sample code

then proceeds with the rest of the argument list. It will proceed just like analyzing the previous argument did, since we have the same expected input type and actual argument type. After the second argument has been handled, we are down to imagining  $+$  as a nullary function.  $\text{APP:FN0}$  gives the result type for the application as the function's stated output type, augmented with the function's own frame (which is still  $(\text{shape})$ ). That output type is propagated back through the outputs of  $\text{APP:FN}^*A$  to give the result type  $(A \text{ Int } (\text{shape}))$  to the whole application term.

### 8.2.2 Vector-Scalar Addition

$$\begin{array}{ll} (+ [4 \ 5 \ 6] \ 2) & (+ (\text{array } (3) \ 4 \ 5 \ 6) \\ & (\text{array } () \ 2)) \end{array}$$

This proceeds like the simple application case until we reach the point of checking the first argument's type. We still have  $\text{Int}$  as the atom type, but the shape equation to solve is now  $(++ \widehat{\text{@af}} (\text{shape})) \doteq (\text{shape } 3)$ , which is satisfied by resolving  $\widehat{\text{@af}}$  as  $(\text{shape } 3)$ . Then the equation for the function's frame extension is  $(++ \widehat{\text{@fe}} (\text{shape})) \doteq (\text{shape } 3)$ , so we must also resolve  $\widehat{\text{@fe}}$  as  $(\text{shape } 3)$ . The final premise of  $\text{APP:FN}^*A$ , invoking the application judgment again, now has a different type for the partially applied function. Instead of a scalar frame around the function, we give it the augmented frame  $(++ \widehat{\text{@fe}} (\text{shape}))$ , which is equivalent to  $(\text{shape } 3)$ .

In handling the second argument, the application judgment now gets the principal frame from the *function* position, so the derivation must use  $\text{APP:FN}^*F$  rather than  $\text{APP:FN}^*A$  ( $\text{APP:FN}^*A$  will fail to discover a suitable function-frame extension because  $(\text{shape } 3)$  is not a prefix of  $(\text{shape})$ ). The existential shape variables we generate now are  $\widehat{\text{@af}}$  to represent the argument frame and  $\widehat{\text{@ae}}$  for the argument's frame extension. Checking the argument type will resolve  $\widehat{\text{@af}}$  as  $(\text{shape})$ , and equating the extended argument frame  $(++ \widehat{\text{@ae}} (\text{shape}))$  with the function frame  $(\text{shape } 3)$  will resolve  $\widehat{\text{@ae}}$  as  $(\text{shape } 3)$ .

When we reach  $\text{APP:FN0}$ , we are treating the function-position array as a 3-vector of functions, each producing a scalar integer as output. So the result type is  $(A \text{ Int } ( ++ (\text{shape } 3) (\text{shape})) )$ —a  $(A \text{ Int } (\text{shape}))$  cell inside a  $(\text{shape } 3)$  frame—or equivalently  $(A \text{ Int } (\text{shape } 3))$ . This result type is propagated back down the derivation tree to type the full application term as  $(A \text{ Int } (\text{shape } 3))$ .

8.2.3 *Vector-Matrix Addition*

```

(+ [100 200 300]      (+ (array (3)
  [[1 2 3 4]         100 200 300)
   [5 6 7 8]        (array (3 4)
   [9 10 11 12]])   1 2 3 4
                       5 6 7 8
                       9 10 11 12))

```

This derivation follows the same path as vector-scalar addition until we get to handling the second argument. Recall that at that point, we act as though the array in function position has type  $(A \rightarrow ((A \text{ Int (shape)})) (A \text{ Int (shape)})) (\text{shape } 3))$ , as though we had partially applied  $+$  to produce a 3-vector of functions on integer scalars. Fresh existential shape variables represent the matrix argument's frame shape and the function array's frame extension. Solving the relevant shape equations identifies the argument frame shape as  $(\text{shape } 3 \ 4)$  and then the function frame extension as  $(\text{shape } 4)$ . When we derive the last premise using APP:FN0, the function array is treated as having type  $(A \rightarrow () (A \text{ Int (shape)})) (\text{shape } 3 \ 4)$ , so the eventual result type is  $(A \text{ Int (shape } 3 \ 4))$ .

8.2.4 *Multiple Instruction, Single Data*

```

([+ * -] 10 5)      ((frame (3) + * -)
                    (array () 10)
                    (array () 5))

```

Type synthesis proceeds much like the previous application examples, but we start with a function frame of  $(\text{shape } 3)$ . Then we must solve for a frame extension for each argument, both of which will also be  $(\text{shape } 3)$ .

8.2.5 *Mismatched Vector-Vector Addition*

```

(+ [4 5 6]          (+ (array (3) 4 5 6)
  [1 2 3 4])        (array (4) 1 2 3 4))

```

After tracing through some examples of successfully typing function application, here we try *ill-typed* code to demonstrate where the bidirectional typing rules reject it. We again skip forward to after handling the first argument. When we consider the function position to

have type  $(A \ (-\> \ ((A \ \text{Int} \ (\text{shape}))) \ (A \ \text{Int} \ (\text{shape}))) \ (\text{shape} \ 3))$ . The first shape equation in  $\text{APP:FN}^*A$  or  $\text{APP:FN}^*F$ —finding the new argument’s frame—succeeds with  $(\text{shape} \ 4)$  as the frame. However the second equation fails.  $\text{APP:FN}^*A$  cannot find  $\widehat{\text{@fe}}$  which solves  $(++ \ (\text{shape} \ 3) \ \widehat{\text{@fe}}) \doteq (\text{shape} \ 4)$ , and  $\text{APP:FN}^*F$  cannot find  $\widehat{\text{@ae}}$  which solves  $(++ \ (\text{shape} \ 4) \ \widehat{\text{@ae}}) \doteq (\text{shape} \ 3)$ .

### 8.2.6 Applying Scalar Identity Function

$$\begin{array}{l} ((\lambda \ ((x \ 0)) \ x) \ [\#t \ \#f]) \quad ((\text{array} \ () \\ \quad \quad \quad (\lambda \ ((x \ 0)) \ x)) \\ \quad \quad \quad (\text{array} \ (2) \ \#t \ \#f)) \end{array}$$

As directed by  $\text{SYN:APP}$ , we must synthesize a type for a user-written function and then ensure the argument is of an acceptable type. We still uphold the principle that values which will be used polymorphically must be explicitly marked. No such mark appears in the source program, so the bidirectional rules will not generate a polymorphic type. This is fine because the function is only used at one type.

$\text{SYN:FN}$  begins by generating fresh existential variables for the input element type and dimensions. Since this function is declared to operate on scalar cells, we need only the element type. The environment is extended with a fresh existential atom type variable  $\widehat{x}$  and the binding for  $x$  at type  $(A \ \widehat{x} \ (\text{shape}))$ . Then the function body, just a reference to  $x$ , has its type synthesized as  $(A \ \widehat{x} \ (\text{shape}))$ . So the type synthesized for the function-position expression is  $(A \ (-\> \ ((A \ \widehat{x} \ (\text{shape}))) \ (A \ \widehat{x} \ (\text{shape}))) \ (\text{shape}))$ .

Proceeding with  $\text{SYN:APP}$ , we invoke the application judgment, and  $\text{APP:FN}^*A$  is the applicable rule. We must check the argument  $(\text{array} \ () \ \#t)$  against the input type  $(A \ \widehat{x} \ (\text{shape}))$ . The first premise for  $\text{CHK:ARRAY}$  checks whether the number of atoms supplied matches the amount required for the stated shape. In this case, we have a 2-vector shape and the required two atoms. The second premise for  $\text{CHK:ARRAY}$  requires atoms with the right type. So we check the boolean literals  $\#t$  and  $\#f$  against the type  $\widehat{x}$ . The only applicable rule here is  $\text{CHK:SUB}$ , which in turn requires  $\text{Bool} \leq \widehat{x}$ . The subtyping judgment will confirm this via  $\text{SUB:INSTR}$  and  $\text{IHIGH:SOLVE}$ , with  $\widehat{x}$  now resolved as  $\text{Bool}$  in the resulting environment.

The third premise of  $\text{CHK:ARRAY}$  equates the array literal’s annotated shape against the shape in the goal type, the unresolved argument frame appended to the scalar cell shape. This succeeds by resolving the argument frame as  $(\text{shape} \ 2)$ . We thus have a scalar principal frame, so the result type is  $(A \ \text{Bool} \ (\text{shape} \ 2))$ .

The elaborated version of the program gives the cell type instead of cell rank:

```
((array () (λ ((x (A Bool (shape)))) x))
 (array (2) #t #f))
```

### 8.2.7 Applying Cell-Rank Polymorphic Identity

```
((λ ((x all)) x) [#t #f])      ((array ()
                                (λ ((x all)) x))
 (array (2) #t #f))
```

We make a slight change to the scalar identity function from the previous example: the specified rank is now `all`. This means that `SYN:FN` will use  $\hat{x}$ , an existential *array* type variable, for `x`'s type. An array type variable as the cell type indicates that the function is written to be polymorphic in this argument's cell rank, so the frame for this argument is required to be scalar. When checking the argument against the input type  $\hat{x}$ , subtyping resolves  $\hat{x}$  as `(A Bool (shape 2))`

The elaborated code is similar to the scalar case, but the argument's shape has propagated to the formal parameter's type annotation:

```
((array () (λ ((x (A Bool (shape 2)))) x))
 (array (2) #t #f))
```

### 8.2.8 Type-Quantified Scalar Identity Function

```
((λ ((x 0)) x)
 : (∀ (&t)
    (-> (&t) &t)))
 [#t #f])      (((array ()
                  (λ ((x 0)) x))
 : (A (∀ (&t)
       (A (->
            (A &t
              (shape)))
          (A &t (shape)))
        (shape))) (shape)))
 (array (2) #t #f))
```

In order to demonstrate instantiation of a polymorphic function, we use a version of the identity function annotated with a polymorphic type. When `SYN:APP` synthesizes a type for the function, the presence of an annotation means that we use `SYN:ANNOT` to switch from synthesis to checking. First, `CHK:ALL` removes the quantifier and adds the universal type variable `&t` to the environment. An explicit type abstraction is inserted into the elaborated code at this point. Despite having a prescribed type for the function, we still must elaborate the rank specifier into a type specifier. A subtype check compares the cell type derived from the

$\emptyset$  rank specifier with the cell type given in the goal. Since  $\emptyset$  is turned into  $(A \ \widehat{x} \ (\text{shape}))$ , subtyping just resolves the existential variable as the universal variable  $\&t$ . Checking the function body's type must use `CHK:SUB` and `SYN:VAR`, looking up the type for the bound variable  $x$  and comparing it against the intended output type  $\&t$ . Since  $x$  is bound at that type, subtyping succeeds with no update to the environment.

The application judgment invoked by `SYN:APP` must begin by using `APP:ALL`, which unwraps the universal type and converts its quantified type variable into an unsolved existential type variable. The function is then elaborated to a type application, instantiating the polymorphic function at the still unknown argument type:  $(t\text{-app} \ \dots \ \widehat{t})$ . We continue with the application derivation treating the function produced by `t-app` as operating on  $\widehat{t}$  scalars. Checking the argument  $[\#t \ \#f]$  against an unknown frame around that atom type resolves  $\widehat{t}$  as `Bool`. The whole term's synthesized type is  $(A \ \text{Bool} \ (\text{shape} \ 2))$ . The elaborated code is explicit about quantification and instantiation:

```
((t-app (array ())
        (Tλ ((&t Atom))
            (array ()
              (λ ((x (A &t (shape)))) x))))
  Bool))
(array (2) #t #f))
```

### 8.2.9 *Type-Quantified Cell-Rank Polymorphic Identity*

```
((λ ((x all)) x)
 : (∀ (*t)
    (-> (*t) *t)))
[#t #f])

(((array ()
   (λ ((x all)) x))
 : (A (∀ (*t)
        (A (-> (*t) *t)
              (shape)))
      (shape)))
 (array (2) #t #f))
```

Instead of quantifying over the input element type, we now quantify over the entire input type—element and shape—at once by using an array type variable. The generated type application must instantiate with the argument's full type rather than the type of its atoms, but this is still determined by a straightforward subtype check. The elaborated code is similar to the previous example:



```

((t-app
  (array ()
    (Tλ ((*t Array))
      (array ()
        (λ ((x *t)) x))))
  (A Bool (shape 2))))
(array (2) #t #f))

```

Note that the rank specifier `all` is required in order for the source program to be well-typed. The annotated type promises to accept arguments of any cell shape, whereas any other rank specifier would fix the cell rank. Recall that subtyping will not relate functions with differing rank specifiers because reranking can change a function’s behavior (although this particular function happens to behave the same).

Although we are able to find `(A Bool (shape 2))` as the result type, we narrowly miss a pitfall: If we had some other argument provide a non-scalar frame, the application judgment would be unable to construct the result type. We would need a `*t` cell inside the appropriate frame, but we have no way to state the result’s atom type and full shape. So although quantifying over array types offers some notational convenience, it is more robust to quantify separately over atom type and shape.

#### 8.2.10 Type- and Cell-Shape Quantified Identity

```

((λ ((x all)) x)
  : (∀ (&t)
    (Π (@s)
      (-> ([&t @s])
          [&t @s])))
 [#t #f])

(((array ()
  (λ ((x all)) x))
  : (A (∀ (&t)
    (A (Π (@s)
      (A (->
        ((A &t @s))
        (A &t @s))
        (shape)))
      (shape)))
    (shape)))
  (array (2) #t #f))

```

Checking the function’s type, as called for by `CHK:ANNOT`, now has two layers of polymorphism to unwrap. We add an element type variable and a shape variable to the environment before we reach the  $\lambda$  itself. The eventual application judgment also must unwrap both layers, adding unsolved existentials  $\hat{t}$  and  $\hat{s}$  to the environment. The shape equality in the check that  $(A \text{ Bool (shape 2)}) \leq (A \hat{t} \hat{s})$  will resolve  $\hat{s}$  as `(shape)`, and the premise which handles the element type will resolve  $\hat{t}$  as `Bool`. The elaborated code includes both index and type application to produce a result of type `(A Bool (shape 2))`.

```

((i-app
 (t-app
  (array ()
   (Tλ ((&t Atom))
    (array ()
     (Iλ ((@s Shape))
      (array ()
       (λ ((x (A &t @s)))
        x))))))
   Bool)
 (shape 2))
 (array (2) #t #f))

```

### 8.2.11 Outer Product

```

(~(0 1)*
 [10 20 30]
 [5 6])
      ((array ()
       (λ ((x 0) (y 1))
        (* x y)))
 (array (3) 10 20 30)
 (array (2) 5 6))

```

Much of the structure of this bidirectional type derivation is similar to that for vector-scalar addition, but the  $\eta$ -expanded function introduces a slight twist. SYN:APP calls for synthesizing the function array's type, so we must elaborate the rank specifiers 0 and 1 into types made from fresh existential variables:  $(A \hat{x})$  and  $(A \hat{y} \text{ (shape } \hat{y}_0))$ . The application of  $*$  in the function body checks whether  $x$  is a frame-extended integer scalar, which resolves  $\hat{x}$  as  $\text{Int}$ . Next, checking whether  $y$  is also a frame-extended integer scalar updates the environment to show  $\hat{y}$  as  $\text{Int}$  as well. Although we do not know the value of  $\hat{y}_0$ , it must be true that  $(\text{shape } \hat{y})$  is a prefix of  $(\text{shape } \hat{y}_0)$ . It is important that the solver is not overeager about choosing values for dimensions here, or it might choose one that conflicts with the value later passed in as  $y$ .

The function body thus has type  $(A \hat{y} \text{ (shape } \hat{y}_0))$ , *i.e.*, a  $(\text{shape } \hat{y}_0)$  frame around the  $(A \text{ Int } (\text{shape } \hat{y}))$  cells produced by  $*$ . When we finish synthesizing the function's type, the existential variables representing frame shapes for  $x$  and  $y$  are no longer needed and drop out of the environment.

In the outer application form, the frame discovered by APP:FN\*A for the first argument is  $(\text{shape } 3)$ . Like the vector-scalar addition case, we continue to the next argument as though the function array itself had shape  $(\text{shape } 3)$ . Checking the second argument against the frame-extended input type  $(A \text{ Int } (++) \hat{\text{af}} \text{ (shape } \hat{y}_0))$  finally forces  $\hat{y}_0$  to resolve to a particular value: it must be 2. The argument frame  $\hat{\text{af}}$

is found to be (shape), which is compatible with the function's frame (shape 3). Wrapping that frame around the function's output type (A Int (shape 2)) gives (A Int (shape 3 2)) as the result type.

Again, although the function  $\sim(0\ 1)*$  appears to be polymorphic in the second argument's cell length, no generalization is asked for (or required). So we have a monomorphic type for this function in the elaborated code:

```
((array ()
  (λ ((x (A Int (shape)))
      (y (A Int (shape 2))))
    (* x y)))
 (array (3) 10 20 30)
 (array (2) 5 6))
```

### 8.2.12 Major Axis Length

```
(length
 [[ [ 1 2 3] [ 4 5 6]]
  [[ 7 8 9] [10 11 12]]
  [[13 14 15] [16 17 18]]
  [[19 20 21] [22 23 24]]])
(length
 (array (4 2 3)
  1 2 3 4 5 6
  7 8 9 10 11 12
  13 14 15 16 17 18
  19 20 21 22 23 24))
```

This example isolates type and index instantiation, but unlike the cell-polymorphic identity function, this requires selecting a dimension and a shape. The length function gives the size of an array's major axis as a term-level integer. This is only possible if the array has a major axis. Matching an argument's shape to that specified by length's type names that particular dimension  $\$l$  and the entire sequence of remaining dimensions  $@c$ .

The argument type includes a shape variable, so no frame lifting is permitted—length is treated as an all-ranked function. After peeling off the polymorphism layers via APP:ALL and APP:PI, we check the argument array against the function input type (A  $\hat{t}$  ( $++$  (shape  $\$l$ )  $@c$ )). SUB:ARRAY's premise for atom types leads to resolving  $\hat{t}$  as Int, and the shape equation ( $++$  (shape  $\$l$ )  $@c$ )  $\doteq$  (shape 4 2 3) is solved by  $\hat{l} = 4$  and  $@c =$  (shape 2 3). When these are substituted in during elaboration, the resulting code is:

```
((i-app (t-app length Int) 4 (shape 2 3))
 (array (4 2 3)
  1 2 3 4 5 6
  7 8 9 10 11 12
  13 14 15 16 17 18
  19 20 21 22 23 24))
```

8.2.13 *Vector Norm*

<pre>(λ ((v 1))   (sqrt    (reduce/0     +. 0 (*. v v))))</pre>	<pre>(λ ((v 1))   (sqrt    (reduce/0     +. (array () 0)       (*. v v))))</pre>
---	--

Synthesizing a type for this function becomes interesting once we get to the application of `reduce/0`. The `APP:ALL` and `APP:PI` rules put  $\hat{t}$ ,  $\hat{l}$ , and  $\hat{c}$  into the environment. The cell type for the first argument is now a scalar containing a binary function on  $(A \ \hat{t} \ \hat{c})$ . Matching the frame-extended version of that with type of `+` must go through the subtype rules for functions. Since `+` consumes floating-point scalars, we need to ensure that the goal input type is a subtype of  $(A \ \text{Float} \ (\text{shape}))$ . This is solved via `SUB:ARRAY` with  $\hat{t}$  as `Float` and  $\hat{c}$  as  $(\text{shape})$ . Since the cell type for the next argument—the alternative zero case—has its existentials resolved, nothing interesting happens there.

The final index argument  $\hat{l}$  is only resolved once we check the third argument. Lifting `*` over two vectors gives a vector of the same length. For the function’s argument `v`, the rank specifier of `1` would be elaborated into a cell type  $(A \ \hat{v} \ (\text{shape} \ \hat{v}0))$ . Passing `v` to `*` is enough to determine that  $\hat{v}$  is `Float`. The third argument to `reduce/0` thus has type  $(A \ \text{Float} \ (\text{shape} \ \hat{v}0))$ , but  $\hat{v}0$  remains unresolved. Equating the expected input shape  $(\text{shape} \ \hat{l})$  and the argument shape  $(\text{shape} \ \hat{v}0)$  still does not give a concrete value, it does require that  $\hat{l}$  and  $\hat{v}0$  be equal. So the solver updates the environment to give  $\hat{v}0$  as the solution for  $\hat{l}$  (and adds  $\hat{l} \doteq \hat{v}0$  to the constraint archive).

The output type of `reduce/0` has solutions for all of its existentials, so we know this application form will produce a float scalar. That scalar is then passed to `sqrt`, with a scalar frame. The final synthesized type is  $(A \ (-> ((A \ \text{Float} \ \hat{v}0)) (A \ \text{Float} \ (\text{shape}))) (\text{shape}))$ , a function from a float vector to a float scalar, but the vector length is still undetermined. This is a monomorphic vector-norm function, but *which* of the  $\mathbb{N}$ -sized family of vector-norm functions won’t be decided until we apply it. The code we have can be applied to a vector of any length, but it must be explicitly generalized in order to be used on vectors of different lengths. The elaborated code still contains the existential variable for the argument’s unspecified length:

```
(λ ((v (A Int (shape  $\hat{v}0$ ))))
  (sqrt ((i-app (t-app reduce/0 Int)  $\hat{v}0$  (shape))
        + (array () 0) (* v v))))
```

8.2.14 *Vector Sum*

$$\sim(1\ 1)+ \quad (\lambda ((x\ 1)\ (y\ 1))\ (+\ x\ y))$$

Synthesizing a type for this function relies on the ability to recognize aliasing of argument dimensions. The function body is only safe to execute if both  $x$  and  $y$  have the same length. We get that ability from having the shape theory solver identify an equivalence relation on dimensions. Elaborating the rank specifiers into cell types gives  $(A\ \hat{x}\ (\text{shape}\ \hat{\$x0}))$  and  $(A\ \hat{y}\ (\text{shape}\ \hat{\$y0}))$ . Each existential atom variable will be identified as  $\text{Int}$  when checking the respective term variable as an argument for  $+$ .

Dimension aliasing is discovered by frame analysis. First,  $\text{APP:FN}^*A$  checks whether  $x$ 's type is a frame extension of an integer scalar, discovering the frame shape  $(\text{shape}\ \hat{\$x0})$ . Then we proceed to the second argument with  $(\text{shape}\ \hat{\$x0})$  as the function array's shape. When the next use of  $\text{APP:FN}^*A$  compares  $y$ 's frame against the function frame, it is forced to equate  $(\text{shape}\ \hat{\$x0})$  and  $(\text{shape}\ \hat{\$y0})$ . The minimal equivalence relation under which that equation is satisfiable has  $\hat{\$x0} \doteq \hat{\$y0}$ , so the solver will update the environment to reflect that  $\hat{\$y0}$  stands for  $\hat{\$x0}$ .

Like the vector-norm function, the elaborated code for vector sum still contains the unresolved length  $\hat{\$x0}$ :

$$\begin{aligned} &(\lambda ((x\ (A\ \text{Int}\ (\text{shape}\ \hat{\$x0}))) \\ &\quad (y\ (A\ \text{Int}\ (\text{shape}\ \hat{\$x0})))) \\ &\quad (+\ x\ y)) \end{aligned}$$
8.2.15 *Transpose and Add*

$$\begin{aligned} &(\lambda ((x\ 2)) \\ &\quad (+\ x\ (\text{transpose}\ x))) \end{aligned} \quad \begin{aligned} &(\lambda ((x\ 2)) \\ &\quad (+\ x\ (\text{transpose}\ x))) \end{aligned}$$

$\text{SYN:FN}$  generates two separate existential dimension variables for  $x$ , but adding a matrix to its own transpose only works if the matrix is square. We have another case of dimension aliasing. This time, the aliasing is discovered when matching argument frames for  $+$ . The application of transpose must resolve its dimension arguments  $\hat{\$a}$  and  $\hat{\$b}$ . The frame-matching shape equation is  $(++\ \hat{x}f\ (\text{shape}\ \hat{\$a}\ \hat{\$b})) \doteq (\text{shape}\ \hat{\$x0}\ \hat{\$x1})$ . Its solution must map  $\hat{x}f$  to  $(\text{shape})$ ,  $\hat{\$a}$  to  $\hat{\$x0}$ , and  $\hat{\$b}$  to  $\hat{\$x1}$ . The result type for transpose is then  $(A\ \text{Int}\ (\text{shape}\ \hat{\$x1}\ \hat{\$x0}))$ .

When typing the application of  $+$ ,  $\text{APP:FN}^*A$  discovers that the first argument has frame  $(\text{shape}\ \hat{\$x0}\ \hat{\$x1})$ . Propagating that to the function

array’s pretended shape, handling the second argument requires solving  $(\widehat{\text{shape}} \ \$x1 \ \$x0) \doteq (++) \ \widehat{\text{te}} \ (\widehat{\text{shape}} \ \$x0 \ \$x1)$ . The second argument’s frame need not expand, so  $\widehat{\text{te}}$  is resolved to  $(\widehat{\text{shape}})$ . The remaining dimensions align such that  $\widehat{\$x0} \doteq \widehat{\$x1}$ , so the environment returned by the solver must have  $\widehat{\$x0}$  as the solution for  $\widehat{\$x1}$ . The function’s input and output types are thus found to be  $(A \ \text{Int} \ (\widehat{\text{shape}} \ \widehat{\$x0} \ \widehat{\$x0}))$ , with the elaborated code as follows:

```
(λ ((x (A Int (shape  $\widehat{\$x0}$   $\widehat{\$x0}$ ))))
  (+ x ((i-app (t-app transpose Int)  $\widehat{\$x0}$   $\widehat{\$x0}$ ) x)))
```

### 8.2.16 1D Stencil

```
(λ ((w 1) (s 1))
  (reduce/0
    ~ (1 1)+
    ((λ ((k 0)) 0.) s)
    (* w (rotate
      (iota/w w) s))))

(λ ((w 1) (s 1))
  (reduce/0
    (array ()
      (λ ((x 1) (y 1))
        (+ x y)))
    ((λ ((k 0))
      (array () 0.)) s)
    (* w (rotate
      (iota/w w) s))))
```

The earlier reduction in the vector-norm function was meant to add up individual numbers in the vector, but here we add up all the vectors in a matrix. This requires a vector-addition function, (typed in an earlier example). The zero case also needs a vector instead of a scalar, which we can construct by lifting the constant-zero function over the scalar cells of  $s$ . Constructing the time-shifted versions of  $s$  lifts  $\text{rotate}$  over a vector of rotation amounts. That vector is constructed by the “iota with shape witness” function:  $\text{iota}/w$  takes an array of any shape and produces an array of matching shape whose atoms are enough natural numbers to fill out the array.

There are several instances of selecting type and index arguments (for  $\text{reduce}/0$ ,  $\text{rotate}$ , and  $\text{iota}/w$ ), elaborating function’s input cell dimensions ( $w$ ,  $s$ ,  $x$ , and  $y$ ), and spotting how an iteration space is built (by  $\text{iota}/w$  and  $\text{rotate}$ ) and then collapsed (by  $\text{reduce}/0$ ).

Through the entire type synthesis process, the lengths of  $w$  and  $s$  remain undetermined, but they are used in the elaborated code to select the lengths of  $x$  and  $y$ , the length of the rotation vector, and the lengths of the reduced and preserved matrix axes.

```

(λ ((w (A Int (shape  $\widehat{s}_0$ )))
    (s (A Int (shape  $\widehat{s}_0$ ))))
  ((i-app (t-app reduce/0 Int)  $\widehat{s}_0$  (shape  $\widehat{s}_0$ ))
    (array () (λ ((x (A Int (shape  $\widehat{s}_0$ )))
                  (y (A Int (shape  $\widehat{s}_0$ ))))
              (+ x y)))
    ((array () (λ ((k (A Int (shape)))
                  (array () 0))))
      s)
    (* w ((i-app (t-app rotate Int)  $\widehat{s}_0$  (shape))
          ((i-app (t-app iota/w Int) (shape  $\widehat{s}_0$ )) w)
          s))))

```

### 8.2.17 1D Stencil with Lifting Reduce

<pre> (λ ((w 1) (s 1))   (reduce/L0     + 0.     (* w (rotate           (iota/w w) s)))) </pre>	<pre> (λ ((w 1) (s 1))   (reduce/L0     +     (array () 0.)     (* w (rotate           (iota/w w) s)))) </pre>
---	--

Having to manage the lifting of a reducing function is awkward for the programmer. A more flexible “lifting reduce” function `reduce/L0` can simplify the code. Instead of requiring the programmer to rerank the reducing function and construct a full cell of zeroes, `reduce/L0` expects a function which can lift to operate along the array’s major axis. How much it will lift is represented by an additional shape argument.

An input type with *multiple* shape variables may look risky due to the potentially high cost of branching in the shape solver’s search. Having multiple possible alignments for shape variables’ boundaries can lead to bad decisions and backtracking within the solver or even multiple distinct solutions to an equation (and backtracking in the subtyping or even bidirectional rules). However, we expect that by the time we reach the argument with multiple shape variables, one of those will already have a solution.

For this particular code, checking `+` against the frame-extended version of `reduce/L0`’s first input type resolves  $\widehat{c}$  as `(shape)`. Then the third argument’s shape equation is  $(++ (\widehat{s}_1) \widehat{f}) \doteq (\widehat{s}_0 \widehat{s}_0)$ , forcing  $\widehat{s}_1 \doteq \widehat{s}_0$  and  $\widehat{f} \doteq (\widehat{s}_0)$ . The elaborated code is smaller as well for not needing the explicit lifting management.

```

(λ ((w (A Int (shape  $\widehat{w0}$ )))
    (s (A Int (shape  $\widehat{s0}$ ))))
  ((i-app (t-app reduce/L0 Int)  $\widehat{w0}$  (shape  $\widehat{s0}$ ) (shape))
   +
   (array () 0)
   (* w ((i-app (t-app rotate Int)  $\widehat{s0}$  (shape))
          ((i-app (t-app iota/w Int) (shape  $\widehat{w0}$ ) w)
           s))))))

```

### 8.2.18 Matrix Product

```

(λ ((a 2) (b 2))
  (~(0 0 2)reduce/L0
   + 0
   ~(1 2)* a b)))

```

```

(λ ((a 2) (b 2))
  ((array ()
    (λ ((f 0)
        (z 0)
        (x 2))
      (reduce/L0 f z x)))
   + (array () 0)
  ((array ()
    (λ ((q 1) (r 2))
        (* q r)))
   a b)))

```

Matrix multiplication is another opportunity to use `reduce/L0`, and the shape equations work out similar to the stencil computation example. It also has dimension aliasing: The second dimension of `a` must match the first dimension of `b`. Identifying this requirement starts from synthesizing the type of the reranked `*` function. Frame matching demands that `q` and `r` share the same first dimension. Then lifting the reranked `*` over arguments `a` and `b` uses `a`'s minor axis as `q`'s sole axis. Thus  $\widehat{r0}$  is resolved as  $\widehat{q0}$ , which in turn is resolved as  $\widehat{a1}$ . At the same time,  $\widehat{r0}$  and  $\widehat{r1}$  are also forced to be equal to  $\widehat{b0}$  and  $\widehat{b1}$  respectively. The elaborated code shows  $\widehat{a1}$  and  $\widehat{b1}$  driving the dimension selection for reranked functions— $\widehat{a0}$  is the leading dimension of the three-dimensional intermediate structure, so it is only used as part of a frame shape.



```

(λ ((a (A Int (shape  $\widehat{a0}$   $\widehat{a1}$ )))
    (b (A Int (shape  $\widehat{a1}$   $\widehat{b1}$ ))))
  ((array ()
    (λ ((f (A (-> ((A Int (shape))
                    (A Int (shape)))
                    (A Int (shape))) (shape)))
      (z (A Int (shape)))
      (x (A Int (shape  $\widehat{a1}$   $\widehat{b1}$ ))))
    ((i-app (t-app reduce/L0 Int)
       $\widehat{a1}$  (shape  $\widehat{b1}$ ) (shape))
     f z x)))
  +
  (array () 0)
  ((array ()
    (λ ((q (A Int (shape  $\widehat{a1}$ )))
        (r (A Int (shape  $\widehat{a1}$   $\widehat{b1}$ ))))
      (* q r)))
    a b)))

```

### 8.2.19 Monomorphic and Polymorphic Functions Coexisting

[+ fst]	(frame (2) + fst)
[fst +]	(frame (2) fst +)
[[+ fst]	(frame (2)
[fst +]]	(frame (2) + fst)
	(frame (2) fst +))

Three closely related examples demonstrate a limitation of the bidirectional rules. Type synthesis for an array or a frame form synthesizes a type for the first atom or cell and uses that type to check the remaining ones. This can lead to trouble with identifying how polymorphic a function array is meant to be.

In the first case, synthesis checks `fst` against the monomorphic type of `+`. This succeeds, with `fst` coerced from polymorphic to monomorphic by type application:

```
(frame (2) + (t-app fst (A Int (shape))))
```

In the second case, synthesis tries to check `+` against the polymorphic type of `fst`. Since `+` is only usable on integers, it is not general enough to pass this check, and type synthesis fails. Developing more order-robust versions of `SYN:ARRAY` and `SYN:FRAME`, such as by devising

<sup>31</sup> *By both Remora and MLsub*

a distributive lattice structure around higher-rank polymorphic types as suggested by Dolan’s design of MLsub [19], is left to future work.<sup>31</sup>

We consider now an alternative possibility. Order dependence makes it as easy to fall out of synthesizability holes as it is to fall in them. If the unfortunately ordered `[fst +]` appears as a cell in a larger frame such as the third case, where another cell identifies the monomorphic atom type, synthesis succeeds. SYN:FRAME is able to synthesize a type for `[+ fst]` and then uses that to check the type of `[fst +]`. Knowing that we want a function on integer scalars prevents us from being led astray by starting with `fst`’s polymorphic type, and we generate the proper instantiation in elaborated code:

```
(frame (2) (frame (2) + (t-app fst (A Int (shape))))
      (frame (2) (t-app fst (A Int (shape))) +))
```

### 8.2.20 Length of Boxed Vector

```
((λ ((x (Σ ($d)
          [Int $d])))
  (unbox ($l v x)
    (length v)))
 (box 4 [1 2 3 4]))
((array ()
  (λ ((x (A (Σ ($d)
            (A Int
              (shape $d))))
    (unbox ($l v x)
      (length v))))
 (box 4 (array (4)
  1 2 3 4)))
```

A box’s type is polymorphic in that it abstracts over the shape of the contents. Unboxing relies on that polymorphism, so a hint about the intended abstraction is required. In this example, that hint is given as a cell-type specification. Having the  $\Sigma$  type annotated clarifies how the type of the contents—bound as `v`—relates to the abstracted dimension `$l`. There is no need to produce boxed output because the result type makes no mention of `$l`.

The argument is given as a `box` form with no annotation. Being passed to a function known to expect a particular  $\Sigma$  type gives a specific enough goal type that we check the argument’s type rather than synthesize it. The elaborated code shows the instantiation of `length` and the generated annotation on the `box`.

```

(array ()
  (λ ((x (A (Σ (($d Dim))
            (A Int (shape $d)))) (shape))))
  (unbox ($l v x)
    ((i-app (t-app length Int) $l (shape)) v))))
(array ()
  (box 4 (array (4) (1 2 3 4))
    (Σ (($d Dim)) (A Int (shape $d))))))

```

### 8.2.21 *Misuse of Unboxing*

```

(λ ((x (Σ ($d)
        [Int $d])))
  (unbox ($l v x)
    (reverse v)))
(box 4 [1 2 3 4]))

(array ()
  (λ ((x (A
        (Σ ($d)
          (A Int
            (shape $d))))
        (shape))))
    (unbox ($l v x)
      (reverse v))))
(box 4 (array (4)
  1 2 3 4)))

```

The only change from the previous example is replacing `length` with `reverse`. Now we have ill-typed code. The type synthesized for the function body includes `$l` as its length, and that variable goes out of scope as we exit the `unbox`.

### 8.2.22 *Factorial*

```

(λ ((x 0))
  (unbox ($l v (iota0 x))
    (reduce/0
      * 0 (+ 1 v))))

(λ ((x 0))
  (unbox ($l v (iota0 x))
    (reduce/0
      * (array () 0)
      (+ (array () 1)
        v))))

```

Instead of a manually constructed `box` form, we have a built-in function returning a boxed vector. Since the boxed array's type is determined from `iota0`'s output type, no annotation is needed. With the `box`'s hidden dimension variable bound as `$l`, frame analysis identifies the application of `+` as having `(shape $l)` as its principal frame. Then `reduce/0`'s instantiation is inferred like in previous examples.

```

(λ ((x (A Int (shape))))
  (unbox ($l v (iota0 x))
    ((i-app (t-app reduce/0 Int) $l (shape))
      * (array () (0)) (+ (array () (1)) v))))

```

### 8.2.23 *Review of Results*

This section’s corpus of Remora code samples covers several cases of rank-polymorphic lifting, with the principal frame provided by the function, an early argument, or a late argument. Instantiation of polymorphic functions is used heavily, but never with a user-provided annotation indicating the instantiation arguments or intended monomorphic type. Reranking is also used in many code samples, highlighting the fact that cell-polymorphic generalization is not necessary for common use of reranking. Since reranking is used as a local, purpose-specific alteration to some pre-existing function, type synthesis only requires choosing a suitable monomorphic type for a reranked function. Even if the underlying function is polymorphic, reranking elaborates to a monomorphic function with the appropriate instantiation of it.

The need for specifying cell *types* instead of ranks only arose when dealing with cell-type polymorphism. Generalizing to polymorphic input type requires that the programmer specify the intended type. Automatic generalization conflicts with Remora’s lack of principal types (due to both algebraic issues with dimension constraints and permitting higher-rank polymorphism). Constructing boxed data also requires a type annotation in order to specify what shape information is hidden and what is exposed, analogous to a module signature in ML. The other awkward point is packing monomorphic and polymorphic functions together in the same array. Uncertainty about how polymorphic to make the function array can lead type synthesis to paint itself into a corner, although if a goal type is specified to disambiguate, the checking mode in the bidirectional rules reaches the right conclusion.

Part III

TRANSLATION



## BACKGROUND

---

Silicon has limited flexibility to handle operands of varying size. While rank-polymorphic code is written without loops, there must eventually be a loop somewhere in order to run it. Past work on compiling APL, and array-oriented code more generally, has focused heavily on emitting appropriate loop code and been limited by the shape information available to the compiler. The goal of Remora is not to invent new uses for shape information but to make it more accessible. At this point, we have a core semantics describing how code which is polymorphic over frame shape ought to run, as well as a way of automatically instantiating code which is polymorphic over cell shape. What remains is to show that execution of Remora code does not rely on extensive run-time type information, as was used in the formal semantics.

There is a fork in the road here in terms of compilation strategy: We can start by paring down the type annotations or by making the iteration structure explicit. Before taking that fork, this chapter gives an overview of past work on translation techniques for array-oriented code and associated translation target.

### 9.1 COMPILATION TARGETS

Abrams defined an abstract machine for running APL programs [3]. The program representation used by this machine is quite close to APL itself, retaining implicit iteration of scalar operations over vector operands, though it is extended with control instructions. A key contribution of this machine is delaying execution of array-producing operations so that they can be fused with later array-consuming operations. Abrams notes the necessity of shape information for guiding the machine's execution. Elsmann and Dybdal designed Typed Array Intermediate Language [26] to keep track of rank information while compiling APL code, but their compiler does not keep full shape details. Follow-up work adds a translation from this TAIL to Futhark [37].

SISAL [67], a language with explicit iteration, has been used as a compilation target by the Apex compiler [6]. Apex uses data-flow analysis to discern the shapes of array values, but APL's hostility to static analysis leads to computing on arrays of uncertain shape. Such operations must have run-time shape checks.

Grelck and Scholz make a strong argument for SAC as a compilation target for rank-polymorphic code [34, 89]. SAC's with loops behave more like comprehensions than traditional for loops, producing arrays as

results. Although moving from APL’s implicit iteration to SAC’s explicit comprehension is a significant leap, little else in a typical APL program needs to be restructured in order to rewrite it as a SAC program. This particular research effort was concerned primarily with performance implications of writing SAC code in the APL style rather than the traditional C style, so no mechanical translation process is presented. However, the semantic leap from APL to SAC turns out to be quite small. When array dimensions are statically unknown due to writing cell-shape independent code, the proper iteration space can still be constructed in SAC by querying arrays’ shapes. SAC code written to follow the APL style far outperformed the original APL code (run through an off-the-shelf interpreter). The improvement comes the caveat that SAC code is rank monomorphic, so mechanical translation would require some analysis to determine all arrays’ ranks. This presents more of a problem for APL than it does for Remora.

Rink and Castrillon designed TeIL (“Tensor Intermediate Language”) as a formally specified intermediate representation for aggregate operations. While it does not use an explicit keyword to indicate iteration, index variables in a TeIL expression encode an iteration space whose boundaries are based on the extents of the dimensions those variables are used to index. TeIL is not meant to be attached to any particular compiler infrastructure, but was motivated partly by unexpected behavior observed in other array-program compilation systems. In TeIL, an array is typed by a tuple of natural number literals representing the array’s shape—TeIL code is not polymorphic in rank or individual dimensions. For Remora’s purposes, targeting TeIL would require dynamic compilation, static monomorphization, or substantial extension of TeIL itself.

Nova [13] was designed as a higher-level front-end to CUDA [70] with several common parallel operations built in. Unlike rank-polymorphic languages, Nova uses an explicit `map` operator which operates only over vectors. Calls to `map` must be nested in order to operate on higher-dimensional data. LambdaJIT [58] is a lower-level CUDA-targeted system originally meant for retargeting C++ STL algorithms for parallel execution on a GPU. Operating at run time allows LambdaJIT to specialize its generated code for the concrete data type and input sizes used in actual calls to STL algorithms. Since many operations provided by the STL—such as `transform`, `iota`, `reduce`—are the bread and butter of array-oriented programming, LambdaJIT has been explored as a compilation target for array programs, specifically with a prototype back-end for the Nova compiler.

The proliferation of custom operators not already provided by high-performance tensor computation libraries motivated the development of Tensor Comprehensions [96]. Such operators may involve memory access patterns or combinations of primitive arithmetic operations which are poorly supported by the underlying hardware. Graph-based systems



(discussed further below) have fared poorly on such use cases. TC's source language is meant to be a more flexible foundation than a collection of array-manipulation primitives such as is provided by a traditional rank-polymorphic language. Like many prior array languages, the program code represents a loop body, with the iteration structure derived by the compiler. However, in TC, that derivation is based partially on the occurrences of array-element index variables in the source program, using them as loop induction variables, as opposed to an APL-style notation with no induction variables at all. The loop bounds for those indices are inferred from the individual array dimensions they are used to implicitly traverse. Indices on the left-hand side of an assignment function somewhat like binding occurrences, and indices which appear free on the right-hand side describe dimensions along which to reduce. Although TC does not promise to always infer ranges correctly in especially tricky cases, particularly when there exist multiple valid choices of loop bounds, Remora need not rely on it to do so.

## 9.2 DATAFLOW GRAPHS

Array-based programming systems targeted at machine learning applications, such as TensorFlow [1] and PyTorch [77], often use a dataflow graph as the internal representation because a neural network is itself organized as a dataflow graph. Such representation facilitates both loop fusion and automatic differentiation by directly stating which inputs and intermediate values influence other computed values. The traditional method of executing a dataflow graph directly still carries significant interpretive overhead, which ought to be compiled away. XLA [92] and Glow [87] are compilers specifically designed towards that goal for TensorFlow and PyTorch respectively.

Glow's compilation pipeline is divided into high- and low-level sections with their own separate internal representations. Glow simplifies compilation to heterogeneous hardware by translating the very large arsenal of commonly used operations into a smaller set of primitives so that low-level stages, which are meant to be written or extended by hardware vendors, do not need to handle as wide a space of possible input programs. The low-level stages can instead focus on target-specific optimizations such as scheduling and memory reuse. Optimizations based on domain knowledge, such as algebraic properties of operations used in the source program, are performed instead on the high-level IR.

TVM [12] also performs substantial graph transformation but its primary contribution is to abstract away hardware-specific details such as constraints on operand dimensions and layout, scheduling and latency issues, and availability of specific instructions. TVM's compilation framework offers a way to declare new hardware-specific intrinsics, along with the code transformations which should generate their use. This method

of connecting higher-level array operations to the set of available hardware operations can create a large search space for target code, and the compiler cannot have extensive pre-formed knowledge of its structure. Because program optimization in this context demands highly general reasoning, *i.e.*, target-specific reasoning is infeasible, TVM uses machine learning to estimate the run-time cost of candidate output programs.

Glow and TVM's motivation echoes that of TC, but focusing on new target machine capabilities rather than new source language operations. In the other direction, TC addresses the issue of dataflow graph IRs not offering a rich enough set of primitives to keep up with a steady stream of newly invented network architectures.

Remora's higher-order features limit ease of statically constructing a dataflow graph, so that is beyond the scope of this work. In future work, dynamic compilation may give a sufficiently first-order view of a Remora program to make graph-based execution viable.

The dynamic semantics given in Chapter 4 relies on ubiquitous type annotations in order to determine how function application will proceed or how a frame of sub-arrays should collapse to a single array. While the possible case of constructing a frame with no actual result cells whose shape can be inspected can only be resolved by consulting a type annotation, the types themselves contain more information than is strictly needed. For example, it does not matter whether we are collapsing an empty frame of functions, an empty frame of integers, or an empty frame of boxes. The result shape is the same, regardless of the type of the atoms contained within the cells. All we truly need is the resulting shape (alternatively, the result cells' shape). Similarly, evaluating a function application requires knowing the expected cell shapes for the arguments, but it could, in principle, be done without knowing anything about their atoms. Function application is still tagged with a result shape, again to head off issues arising from mapping over an empty frame.

### 10.1 ERASED REMORA

In a type-erased version of Remora, we only need the term and index levels—the syntactic class of types is discarded. The syntax for erased Remora is given in Figure 10.1. Note that the grammar of type indices from Figure 4.1 is still in use here, although expressions, atoms, and their corresponding function and value-form subsets are now replaced with type-erased versions.

Evaluation in erased Remora proceeds similarly to explicit Remora. A function-application form has a principal frame chosen to be the largest of the function and argument frames, and a *lift* reduction replicates the function and argument arrays' atoms to bring all of the frames into agreement. The argument frames themselves are identified based on the individual argument positions' cell-shape annotations, rather than by inspecting a type annotation on the array in function position. A *map* reduction turns an application form where all pieces have the same frame into a *frame* form, where the end-result shape matches the result shape tag on the original application. Index application also maps over an array of index functions, producing a *frame* of substituted function bodies. Since the type level has been eliminated, there are no  $\top\lambda$  and  $\top$ -app forms and no need for a  $t\beta$  reduction rule.

$\widehat{e} \in \widehat{Expr} ::=$		<i>Type-erased expressions</i>
$x$		<i>Variable reference</i>
$  (\text{array } (n \dots) \widehat{a} \dots)$		<i>Array, containing atoms</i>
$  (\text{frame } \iota \widehat{e} \dots)$		<i>Frame, containing sub-arrays</i>
$  (\widehat{e}_f (\widehat{e}_a \iota_a) \dots \iota_r)$		<i>Term application</i>
$  (\text{i-app } \widehat{e}_f \iota_a \dots \iota_r)$		<i>Index application</i>
$  (\text{unbox } (x_i \dots x_e \widehat{e}_s) \widehat{e}_b \iota_b)$		<i>Let-binding box contents</i>
$\widehat{a} \in \widehat{Atom} ::=$		<i>Type-erased atoms</i>
$\mathfrak{b}$		<i>Base value</i>
$  \widehat{f}$		<i>Function</i>
$  (\text{I}\lambda (x \dots) \widehat{e})$		<i>Index abstraction</i>
$  (\text{box } \iota \dots \widehat{e})$		<i>Boxed array</i>
$\widehat{f} \in \widehat{Func} ::=$		<i>Type-erased functions</i>
$\mathfrak{o}$		<i>Primitive operator</i>
$  (\lambda (x \dots) \widehat{e})$		<i>Term abstraction</i>
$\widehat{v} \in \widehat{Val} ::=$		<i>Type-erased values</i>
$x$		
$  (\text{array } (n \dots) \widehat{v} \dots)$		
$\widehat{v} \in \widehat{Atval} ::=$		<i>Type-erased atomic values</i>
$\mathfrak{b}$		
$  \widehat{f}$		
$  (\text{I}\lambda (x \dots) \widehat{e})$		
$  (\text{box } \iota \dots \widehat{v})$		
$\widehat{V} \in \widehat{Ctxt} ::=$		<i>Type-erased evaluation contexts</i>
$\square$		
$  (\text{array } (n \dots) \widehat{v} \dots$		
$\quad (\text{box } \iota \dots \widehat{V})$		
$\quad \widehat{a} \dots)$		
$  (\text{frame } \iota \widehat{v} \dots \widehat{V} \widehat{e} \dots)$		
$  (\widehat{V} (\widehat{e}_a \iota_a) \dots \iota_r)$		
$  (\widehat{e}_f (\widehat{v}_a \iota_a) \dots (\widehat{V} \iota_a)$		
$\quad (\widehat{e}_a \iota_a) \dots \iota_r)$		
$  (\text{i-app } \widehat{V} \iota_a \dots \iota_r)$		
$  (\text{unbox } (x_i \dots x_e \widehat{V}) \widehat{e}_b \iota_b)$		
$  (\text{unbox } (x_i \dots x_e \widehat{v}_s) \widehat{V} \iota_b)$		
$\widehat{t} \in \widehat{Term} ::= \widehat{e} \mid \widehat{a}$		<i>Type-erased terms</i>

Figure 10.1: Abstract syntax for type-erased Remora

$$\begin{aligned}
& ((\text{array } (n_f \dots) \widehat{v}_f \dots) \\
& \quad ((\text{array } (n_a \dots n_i \dots) \widehat{v}_a \dots) (\text{shape } n_i \dots)) \dots \\
& \quad \iota_r) \\
& \mapsto_{\text{lift}} \\
& ((\text{array } (n_p \dots) \text{Concat} \left[ \left[ \text{Rep}_{n_{fe}} \left[ \left[ \text{Split}_1 \left[ \left[ \widehat{v}_f \dots \right] \right] \right] \right] \right] \right] \right) \\
& \quad ((\text{array } (n_p \dots \quad \quad \quad n_i \dots) \text{Concat} \left[ \left[ \text{Rep}_{n_{ae}} \left[ \left[ \text{Split}_{n_{ac}} \left[ \left[ \widehat{v}_a \dots \right] \right] \right] \right] \right] \right] \right) \\
& (\text{shape } n_i \dots)) \dots \\
& \quad \iota_r)
\end{aligned}$$

where

Not all of  $(n_f \dots), (n_a \dots) \dots$  are equal

$$\begin{aligned}
n_p \dots &= \bigsqcup \left[ \left[ (n_f \dots) (n_a \dots) \dots \right] \right] & n_{fe} &= \frac{\prod (n_p \dots)}{\prod (n_f \dots)} \\
n_{ae} \dots &= \frac{\prod (n_p \dots)}{\prod (n_a \dots)} \dots & n_{ac} \dots &= \left( \prod (n_i \dots) \right) \dots
\end{aligned}$$

$$\begin{aligned}
& ((\text{array } (n_f \dots) \widehat{v}_f \dots) \\
& \quad ((\text{array } (n_f \dots n_i \dots) \widehat{v}_a \dots) (\text{shape } n_i \dots)) \dots \\
& \quad \iota_r) \\
& \mapsto_{\text{map}} \\
& (\text{frame } (n_f \dots) \\
& \quad ((\text{array } () \widehat{v}_f) ((\text{array } (n_i \dots) \widehat{v}_c \dots) (\text{shape } n_i \dots)) \iota_c) \dots) \\
& \text{where}
\end{aligned}$$

$$\begin{aligned}
n_c \dots &= \left( \prod n_i \dots \right) \dots \\
((v_c \dots) \dots) \dots &= \text{Transpose} \left[ \left[ \text{Split}_{n_c} \left[ \left[ v_a \dots \right] \right] \right] \right] \\
\text{Length} \left[ \left[ n_f \dots \right] \right] &> 0 \\
\iota_c \dots &= (\iota_r \div (\text{shape } n_f \dots)) \dots
\end{aligned}$$

$$\begin{aligned}
& ((\text{array } () (\lambda (x \dots) \widehat{e})) ((\text{array } (n_i \dots) \widehat{v}) (\text{shape } n_i \dots)) \dots \\
& \quad \iota_r) \\
& \mapsto_{\beta} \widehat{e}[x \mapsto \widehat{v}, \dots]
\end{aligned}$$

$$\begin{aligned}
& (\text{i-app } (\text{array } (n_f \dots) (\lambda (x \dots) \widehat{e}) \dots) \iota_a \dots \iota_r) \\
& \mapsto_{i\beta} (\text{frame } \iota_r \widehat{e}[x \mapsto \iota_a, \dots] \dots)
\end{aligned}$$

$$\begin{aligned}
& (\text{frame } (\text{shape } n \dots) (\text{array } (n' \dots) v \dots) \dots) \\
& \mapsto_{\text{collapse}} (\text{array } (n \dots n' \dots) \text{Concat} \left[ \left[ (v \dots) \dots \right] \right])
\end{aligned}$$

$$\begin{aligned}
& (\text{unbox } (x_i \dots x_e (\text{array } (n_s \dots) (\text{box } \iota \dots \widehat{v}))) \widehat{e} \iota_b) \\
& \mapsto_{\text{unbox}} (\text{frame } (++) (\text{shape } n_s \dots) \iota_b) e[x_i \mapsto \iota, \dots, x_e \mapsto \widehat{v}])
\end{aligned}$$

Figure 10.2: Dynamic semantics for erased Remora

The translation from explicit Remora to erased Remora consists of three erasure functions:  $\mathcal{E}[\cdot] : Expr \rightarrow \widehat{Expr}$ ,  $\mathcal{A}[\cdot] : Atom \rightarrow \widehat{Atom}$ , and  $\mathcal{T}[\cdot] : Type \rightarrow Index$ . These functions are defined in Figure 10.3.

We also define  $\mathcal{C}[\cdot] : Ctxt \rightarrow \widehat{Ctxt}$ , given in Figure 10.4, which is not needed for defining the erased form of an explicit Remora program but is useful for demonstrating their equivalence.

Types in explicit Remora are turned into indices in erased Remora. These indices are the dynamic residue of types, in the same sense that term-level values are dynamic, though they are still subject to a static discipline which governs their values and their relation to the array values they describe. Array types become just the shapes used to construct them, whereas functions, universals, dependent sums and products, and base types become the “scalar” shape. Extracting the index components of all types means that type variables can be turned into index variables, which will stand for the index component of whatever type the variable originally stood for. This translation captures exactly the information that a frame form needs in the event that there are no cells. By extension, the term and index application forms also get the bookkeeping information needed by the frames they will eventually become.

For example, consider a function term whose type is  $(\rightarrow (s (Arr\ t\ (shape\ k))) (Arr\ t\ (shape\ k)))$ , where  $s$ ,  $t$ , and  $k$  are bound as `Array`, `Atom`, and `Dim` respectively. This function produces a vector of some statically uncertain length containing atoms of uncertain type. When we apply this function, the explicitly typed application form describes the resulting array’s type. If our arguments are a single  $s$  and a  $n \times 4$  matrix of numbers, with  $n$  also bound as a `Dim`, the principal frame shape is  $(shape\ n\ 4)$ . So we will have result type  $(Arr\ Num\ (shape\ n\ 4\ k))$ . Type-erasing the application form must still preserve enough information to produce an array of the correct shape, even if  $n$  turns out to be 0, leaving us with no result cells whose shape we can inspect. However, the dynamic semantics does not rely on knowing that the result array contains `Nums`. The binders for index variables  $n$  and  $k$ , which must be either `I $\lambda$`  or `unbox`, are still present in the type-erased program, since the indices they eventually bind to those variables will affect the program’s semantics. The `T $\lambda$ s` which bind  $s$  and  $t$  turn into `I $\lambda$ s`, though the variable  $t$  is never used in the type-erased program. If any type argument was bound to  $s$  in the original program, we replace it with its shape. All occurrences of  $s$  from the original program now stand for an array shape rather than a full array type.

## 10.2 CORRECTNESS OF TRANSLATION

We develop a bisimulation argument to show that the behavior of an explicitly typed term matches the behavior of its erased form. We define the space  $S$  of machine states to be the sum of the set of well-typed

$$\begin{aligned}
& \mathcal{E}[(\text{array } (n \dots) \mathbf{a} \dots)^{\tau_r}] \\
& \quad = (\text{array } (n \dots) \mathcal{A}[\mathbf{a}] \dots) \\
& \mathcal{E}[(\text{frame } (n \dots) e \dots)^{\tau_r}] \\
& \quad = (\text{frame } (\mathcal{T}[\tau_r]) \mathcal{E}[e] \dots) \\
& \mathcal{E}[(e_f^{(\lambda (-> (\tau_i \dots) \tau_o) \iota_f)} e_a \dots)^{\tau_r}] \\
& \quad = (\mathcal{E}[e_f] (\mathcal{E}[e_a] \mathcal{T}[\tau_i]) \dots \mathcal{T}[\tau_r]) \\
& \mathcal{E}[(\text{t-app } e_f \tau_a \dots)^{\tau_r}] \\
& \quad = (\text{i-app } \mathcal{E}[e_f] \mathcal{T}[\tau_a] \dots \mathcal{T}[\tau_r]) \\
& \mathcal{E}[(\text{i-app } e_f \iota_a \dots)^{\tau_r}] \\
& \quad = (\text{i-app } \mathcal{E}[e_f] \iota_a \dots \mathcal{T}[\tau_r]) \\
& \mathcal{E}[(\text{unbox } (x_i \dots x_e e_s) e_b^{\tau_b})] \\
& \quad = (\text{unbox } (x_i \dots x_e \mathcal{E}[e_s]) \mathcal{E}[e_b] \mathcal{T}[\tau_b])
\end{aligned}$$

$$\begin{aligned}
& \mathcal{A}[\mathbf{o}] = \mathbf{o} \\
& \mathcal{A}[\mathbf{b}] = \mathbf{b} \\
& \mathcal{A}[(\lambda ((x \tau) \dots) e)] = (\lambda (x \dots) \mathcal{E}[e]) \\
& \mathcal{A}[(\text{T}\lambda ((x k) \dots) v)] = (\text{I}\lambda (x \dots) \mathcal{E}[v]) \\
& \mathcal{A}[(\text{I}\lambda ((x \gamma) \dots) v)] = (\text{I}\lambda (x \dots) \mathcal{E}[v]) \\
& \mathcal{A}[(\text{box } \iota \dots e \tau)] = (\text{box } \iota \dots \mathcal{E}[e])
\end{aligned}$$

$$\begin{aligned}
& \mathcal{T}[x] = x \\
& \mathcal{T}[(\text{A } \tau \iota)] = \iota \\
& \mathcal{T}[\tau] = (\text{shape}) \quad \text{otherwise}
\end{aligned}$$

$$\mathcal{R}[\mathbf{a}] = \mathcal{A}[\mathbf{a}] \quad \mathcal{R}[e] = \mathcal{E}[e]$$

Figure 10.3: Type erasure for Remora

$$\begin{aligned}
\mathcal{C}[\square] &= \square \\
\mathcal{C}[(\text{array } (n \dots) \mathbf{v} \dots (\text{box } \iota \dots \mathbb{V} \tau) \mathbf{a} \dots)] & \\
&= (\text{array } (n \dots) \mathcal{A}[\mathbf{v}] \dots (\text{box } \iota \dots \mathcal{C}[\mathbb{V}] ) \mathcal{A}[\mathbf{a}] \dots ) \\
\mathcal{C}[(\text{frame } (n \dots) \mathbf{v} \dots \mathbb{V} e \dots)^{\tau_r}] & \\
&= (\text{frame } (\mathcal{T}[\tau_r]) \mathcal{E}[\mathbf{v}] \dots \mathcal{C}[\mathbb{V}] \mathcal{E}[e] \dots ) \\
\mathcal{C}[(\mathbb{V}^{\Lambda (-> (\tau_1 \dots \tau_o) \iota_f)} e_a \dots)^{\tau_r}] & \\
&= (\mathcal{C}[\mathbb{V}] (\mathcal{E}[e_a] \mathcal{T}[\tau_i]) \dots \mathcal{T}[\tau_r]) \\
\mathcal{C}[(e_f^{\Lambda (-> (\tau_1 \dots \tau_2 \tau_3 \dots) \tau_o) \iota_f}) \mathbf{v}_1 \dots \mathbb{V} e_3 \dots)^{\tau_r}] & \\
&= (\mathcal{E}[e_f] (\mathcal{E}[\mathbf{v}_1] \mathcal{T}[\tau_1]) \dots \\
&\quad (\mathcal{C}[\mathbb{V}] \mathcal{T}[\tau_2]) \\
&\quad (\mathcal{E}[e_3] \mathcal{T}[\tau_3]) \dots ) \\
\mathcal{C}[(\text{t-app } \mathbb{V} \tau_a \dots)^{\tau_r}] & \\
&= (\text{i-app } \mathcal{C}[\mathbb{V}] \mathcal{T}[\tau_a] \dots \mathcal{T}[\tau_r]) \\
\mathcal{C}[(\text{i-app } \mathbb{V} \iota_a \dots)^{\tau_r}] & \\
&= (\text{i-app } \mathcal{C}[\mathbb{V}] \iota_a \dots \mathcal{T}[\tau_r]) \\
\mathcal{C}[(\text{unbox } (x_i \dots x_e \mathbb{V}) e_b)] & \\
&= (\text{unbox } (x_i \dots x_e \mathcal{C}[\mathbb{V}]) \mathcal{E}[e_b])
\end{aligned}$$

Figure 10.4: Type-erasing Remora evaluation contexts



explicit Remora terms and the set of their type-erased forms. That is,  $S = \widehat{Expr^T} \uplus \widehat{Expr^T}$ , where  $Expr^T = \{e \in Expr \mid \cdot; \cdot; \cdot \vdash e : \tau\}$  and  $\widehat{Expr^T} = \{\mathcal{E}[e] \mid e \in Expr^T\}$ . Transitions in the machine match the explicit and erased languages' respective  $\mapsto$  relations. We also define the “erasure equivalence” relation  $\cong_{\mathcal{E}}$  on machine states as the equivalence closure of the relation imposed by  $\mathcal{E}[\cdot]$ . Before we show that  $\cong_{\mathcal{E}}$  is a bisimulation, several intermediate results are needed.

First, the bisimulation proof will in one case need to reach deep into an expression to find the next redex. A compositionality property of the erasure rule will make it possible to reason about the redex and its reduced form separately from the evaluation context in which it is embedded.

**Lemma 10.2.1** (Erasure in context). *Given an evaluation context  $\mathbb{V}$  and expression  $e$ , where  $\mathbb{V}[e]$  is well-typed,  $\mathcal{E}[\mathbb{V}[e]] = \mathcal{C}[\mathbb{V}][\mathcal{E}[e]]$ .*

*Proof sketch.* This follows from straightforward induction on  $\mathbb{V}$ .  $\square$

We will also rely on a series of lemmas showing that substitution commutes with erasure.

**Lemma 10.2.2** (Substituting terms into terms commutes with erasure).  $\mathcal{R}[t[x \mapsto \mathcal{E}[e_x]]] = \mathcal{R}[t][x \mapsto \mathcal{E}[e_x]]$

*Proof sketch.* This is straightforward induction on  $t$ .  $\square$

**Lemma 10.2.3** (Substituting types into types commutes with erasure).  $\mathcal{T}[\tau[x \mapsto \tau_x]] = \mathcal{T}[\tau][x \mapsto \mathcal{T}[\tau_x]]$

*Proof sketch.* This is straightforward induction on  $\tau$ .  $\square$

**Lemma 10.2.4** (Substituting types into terms commutes with erasure).  $\mathcal{R}[t[x \mapsto \tau_x]] = \mathcal{R}[t][x \mapsto \mathcal{T}[\tau_x]]$

*Proof sketch.* This is straightforward induction on  $t$ .  $\square$

**Lemma 10.2.5** (Substituting indices into types commutes with erasure).  $\mathcal{T}[\tau[x \mapsto \iota_x]] = \mathcal{T}[\tau][x \mapsto \iota_x]$

*Proof sketch.* This is straightforward induction on  $\tau$ .  $\square$

**Lemma 10.2.6** (Substituting indices into terms commutes with erasure).  $\mathcal{R}[t[x \mapsto \iota_x]] = \mathcal{R}[t][x \mapsto \iota_x]$

*Proof sketch.* This is straightforward induction on  $t$ .  $\square$

**Lemma 10.2.7** (Values erase to values). *For any well-typed term  $t$ ,*

- *If  $t$  has the form  $\mathbf{v}$ , then  $\mathcal{R}[t]$  has the form  $\widehat{\mathbf{v}}$*
- *If  $t$  has the form  $v$ , then  $\mathcal{R}[t]$  has the form  $\widehat{v}$*

*Proof sketch.* We use induction on  $t$ . The only nontrivial cases are `box` and array forms, which may be values or may contain incomplete computation. The contents of a `box` value must itself be a value, which the induction hypothesis implies will erase to a value. Similarly, an array value contains only atomic values, which erase to atomic values.  $\square$

**Lemma 10.2.8** (Lockstep). *For any well-typed  $e$ , one of the following holds:*

- $e$  has the form  $v$ , and  $\mathcal{E}[[e]]$  has the form  $\hat{v}$
- $e \mapsto e'$ , and  $\mathcal{E}[[e]] \mapsto \mathcal{E}[[e']]$
- $e \not\mapsto$ , and  $\mathcal{E}[[e]] \not\mapsto$

*Proof sketch.* We prove this by induction on  $e$ .

We rely on Lemma 10.2.1 (erasure in context) when  $e$  is a redex  $e_r$  within an evaluation context  $\mathbb{V}$  other than  $\square$ . If  $e_r \not\mapsto$  because we have a mis-applied primitive operator, then the same is true for  $\mathcal{E}[[e_r]]$ , so  $\mathcal{E}[[e]]$  is also an evaluation context around a mis-applied primitive operator. Otherwise,  $e_r \mapsto e'_r$ , and the induction hypothesis implies that  $\mathcal{E}[[e_r]] \mapsto \mathcal{E}[[e'_r]]$ . So  $\mathcal{E}[[e]] \mapsto \mathcal{E}[[e']]$ , the erased context filled with  $e'_r$ .

Values are handled by Lemma 10.2.7. For the remaining cases—redexes—straightforward symbol pushing shows that erased Remora's reduction rules follow those of Remora.  $\square$

Since we have a deterministic operational semantics for both explicitly typed Remora and type-erased Remora, the lockstep lemma also works in reverse. If an erased term takes an evaluation step, its preimage cannot be a value form or stuck state. The preimage must therefore step to some result expression, which itself erases to the same result. Similarly, a value form or stuck state in erased Remora cannot have a preimage which takes an evaluation step.

**Corollary 10.2.1** (Reverse lockstep). *If  $\mathcal{E}[[e]] \mapsto \mathcal{E}[[e']]$ , then for any  $e''$  such that  $e \mapsto e''$ , we have  $e' \cong_{\mathcal{E}} e''$ , and at least one such  $e''$  exists. If  $\mathcal{E}[[e]] \not\mapsto$ , then  $e \not\mapsto$ .*

Recall our relation  $\cong_{\mathcal{E}}$  on the set of machine states  $S = \text{Expr}^T \uplus \widehat{\text{Expr}^T}$ , where  $\text{Expr}^T = \{e \in \text{Expr} \mid \cdot; \cdot \vdash e : \tau\}$ , i.e. the set of well-typed explicitly-typed terms, and  $\widehat{\text{Expr}^T}$  is the image of  $\text{Expr}^T$  under type erasure.  $\cong_{\mathcal{E}}$  is the equivalence closure of the relation given by the erasure function  $\mathcal{E}[[\cdot]]$ . That is,  $\cong_{\mathcal{E}}$  is the least relation which relates two states  $s$  and  $w$  iff any of the following hold:

1.  $s \in \text{Expr}^T$  and  $\mathcal{E}[[s]] = w$  (erasure proper)
2.  $s =_{\alpha} w$  (reflexivity)
3.  $w \cong_{\mathcal{E}} s$  (symmetry)

4.  $s \cong_{\mathcal{E}} s'$  and  $s' \cong_{\mathcal{E}} w$  (transitivity)

Expanding the erasure relation based on  $\mathcal{E}[\cdot]$  to include both symmetry and transitivity relates any two explicitly typed expressions which produce  $\alpha$ -equivalent erased terms. A  $\cong_{\mathcal{E}}$  equivalence class consists of a single erased Remora expression and all of its preimages. There can be only one erased Remora expression because type erasure is a well-defined function (*i.e.*, no single explicitly typed expression can erase to multiple different results). Formally, every  $\cong_{\mathcal{E}}$  equivalence class must have the form

$$\{\widehat{e}\} \uplus \{e \in Expr \mid \mathcal{E}[e] = \widehat{e}\}$$

**Theorem 10.2.1.**  $\cong_{\mathcal{E}}$  is a bisimulation. That is, for any states  $s, w \in S$  if  $s \cong_{\mathcal{E}} w$ , either  $(s \mapsto u \wedge w \mapsto v \wedge u \cong_{\mathcal{E}} v)$  or  $(s \not\mapsto \wedge w \not\mapsto)$ .

*Proof.* There are four cases to consider, depending on which of  $Expr$  or  $\widehat{Expr}$  each related term is drawn from, but we can merge the two cases where  $s$  and  $w$  are drawn from different languages.

$s \in \widehat{Expr}$  AND  $w \in Expr$ , OR VICE VERSA: Then  $s$  is the sole type-erased expression in its equivalence class, and  $\mathcal{E}[w] = s$  (or vice versa). Our proof obligation is exactly the lockstep lemma (Lemma 10.2.8).

$s, w \in \widehat{Expr}$ : Since each equivalence class contains only one type-erased expression,  $s = w$ . They must therefore have the same reduction behavior.

$s, w \in Expr$ : If  $s \not\mapsto$ , then the lockstep lemma implies  $\mathcal{E}[s] = \mathcal{E}[w] \not\mapsto$ . Then by reverse lockstep,  $w \not\mapsto$  as well. On the other hand, if  $s \mapsto s'$ , then  $\mathcal{E}[s] = \mathcal{E}[w] \mapsto \mathcal{E}[s']$ . Lockstep implies  $\mathcal{E}[w] \mapsto \mathcal{E}[w']$ . Since Erased Remora has deterministic operational semantics,  $\mathcal{E}[s'] = \mathcal{E}[w']$  (they are both the result of taking an evaluation step from the same expression). Therefore, all of their preimages, including  $s'$  and  $w'$  are erasure-equivalent, *i.e.*,  $s' \cong_{\mathcal{E}} w'$ .  $\square$



## EXPLICIT ITERATION

Using the types as a guide, Remora’s implicit control structure can be translated into a form with explicit control. Given a fully type-annotated term, as required by the operational semantics in Chapter 4, all information required for a compiler to emit explicit loop code is available. The looping itself will be represented by adding new syntactic forms `map` and `rep`, representing lifting a function over several arguments with a uniform frame and replicating cells of an array to expand its frame. These forms replace Remora’s rank-polymorphic function application.

This can be combined with the type erasure pass from Chapter 10. Doing so requires a compiler to decorate abstract syntax tree nodes with frame information.

The transformations described in this chapter might be performed in different orders. While the presentation here has explicit iteration precede conversion of indices to expressions, the prototype compiler used for developing these transformations included merging the term and index levels as part of type erasure. The hard constraint on pass ordering is that type erasure should not precede frame and shape annotation—take note of that information while it is readily available.

## 11.1 MAPPING AND REPLICATION

The internal representation for explicit iteration introduces two new syntactic forms to describe term-level iteration, `map` and `rep`, described in the grammar in Figure 11.1. The `map` form replaces implicitly iterative function application, and it specifies the frame shape  $l_f$ . This one frame is used for the function position and all argument positions. A type annotation for the result cells is needed in case the frame itself is empty. Since Remora does not require equal frames, the `rep` form is needed to describe how individual arrays’ cells must be replicated in order to match. Function application itself is now restricted to scalar frames using the `s-app` form, which is equivalent to application in a conventional functional language.

The explicit-iteration grammar can be tuned to account for prior compilation passes. If types are erased before making iteration explicit, the type in a `map`, empty array, or empty frame is replaced with an erased type, *i.e.*, a shape. If a prior pass has merged the term and index levels,<sup>32</sup> the shapes in `map` and `rep` forms are vector terms rather than indices. Depending on how thoroughly types have been erased in prior passes, the internal representation may include analogues of `map` for type and

<sup>32</sup> How to do so will be discussed in the next section.

$e \in \text{Expr} ::=$	Expressions
$x$	Variable reference
$  \text{array } (n \dots) \mathfrak{a} \dots$	Array, containing atoms
$  \text{array } (n \dots) \tau$	Empty array and its atom type
$  \text{frame } (n \dots) e \dots$	Frame, containing array cells
$  \text{frame } (n \dots) \tau$	Empty frame and its cell type
$  \text{rep } I_f I_g \mathfrak{T} e_a$	Cell Replication
$  \text{map } I_f T_c e_f e_a \dots$	Function mapping
$  \text{s-app } \mathfrak{a}_f e_a \dots$	Scalar-frame application
$  \text{t-map } I_f T_c e \tau \dots$	Type mapping (optional)
$  \text{t-app } \mathfrak{a}_f \tau \dots$	Type application (optional)
$  \text{i-map } I_f T_c e \iota_a \dots$	Index mapping (optional)
$  \text{i-app } \mathfrak{a}_f \iota \dots$	Index application (optional)
$  \text{unbox } I_s T_c (x_i \dots x_e e_s) e_b$	Let-binding box contents

Figure 11.1: Abstract syntax for an internal representation with explicit iteration

index arguments. A frame shape is still required to specify the iteration space, but no argument replication is needed.

With new syntactic forms come new reduction rules, given in Figure 11.2. Where explicit Remora selects which reduction rule to use for a function application form based on the types of the function and argument arrays, the explicit-iteration representation removes this decision. A `map` can only take a *map*-like step, and a `rep` can only take the single-argument analogue of a *lift* step. However, since this form does not include pervasive type annotations, we must treat empty-frame cases differently. If the term and index level have merged, replacing shapes in the reduction rules with vectors, for example replacing  $(\text{shape } n_f \dots)$  with  $(\text{array } (n_f \dots) n_r)$ .

Explicit Remora also has implicit iteration in its `unbox` form, looping over each `box` in an array. So the explicit-iteration representation must include a frame shape identifying the iteration space.

If prior translation passes have not eliminated type- and index-level computation, the additional reduction rules in Figure 11.3 are needed. Mapping and applying type and index abstractions behaves like functions; as in explicit Remora, no *lift* analogue is needed.

Producing this internal representation from an explicit Remora program can be done by inspecting the type annotations on each function application and its function and argument arrays. The definitions of the erasure metafunctions— $\text{Expl}_{\mathcal{E}}[\![\cdot]\!]$  for expressions and  $\text{Expl}_{\mathcal{A}}[\![\cdot]\!]$  for atoms—are given in Figure 11.4.

$$\begin{aligned}
& (\text{rep } (\text{shp } n_f \dots) (\text{shp } n_g \dots) \mathfrak{T}) \\
& \quad (\text{array } (n_f \dots n_c \dots) \mathbf{v} \dots) \\
& \mapsto_{\text{rep}} \\
& (\text{array } (n_f \dots n_g \dots n_c \dots) \text{Rep}_{\prod(n_g \dots)} \llbracket \text{Split}_{\prod(n_c \dots)} \llbracket \mathbf{v} \dots \rrbracket \rrbracket) \\
& \text{where } 0 \notin n_g \dots
\end{aligned}$$

$$\begin{aligned}
& (\text{rep } (\text{shp } n_f \dots) (\text{shp } n_g \dots) \mathfrak{T}) \\
& \quad (\text{array } (n_f \dots n_c \dots) \mathbf{v} \dots) \\
& \mapsto_{\text{rep}0f} (\text{array } (n_f \dots n_g \dots n_c \dots) \mathfrak{T}) \\
& \text{where } 0 \in n_g \dots
\end{aligned}$$

$$\begin{aligned}
& (\text{rep } (\text{shp } n_f \dots) (\text{shp } n_g \dots) \mathfrak{T} (\text{array } (n_f \dots n_c \dots) \mathfrak{T})) \\
& \mapsto_{\text{rep}0c} (\text{array } (n_f \dots n_g \dots n_c \dots) \mathfrak{T})
\end{aligned}$$

$$\begin{aligned}
& (\text{map } (\text{shp } n_f \dots) T_c \\
& \quad (\text{array } (n_f \dots) \mathbf{v}_f \dots) (\text{array } (n_f \dots n_c \dots) \mathbf{v}_a \dots) \dots) \\
& \mapsto_{\text{map}} \\
& (\text{frame } (n_f \dots) (\text{s-app } \mathbf{v}_f (\text{array } (n_c \dots) \mathbf{v}_c \dots) \dots) \dots) \\
& \text{where } ((\mathbf{v}_c \dots) \dots) \dots = \text{Transpose} \llbracket \text{Split}_{n_c} \llbracket \mathbf{v}_a \dots \rrbracket \dots \rrbracket \\
& \quad 0 \notin n_f \dots
\end{aligned}$$

$$\begin{aligned}
& (\text{map } (\text{shp } n_f \dots) T_c \\
& \quad (\text{array } (n_f \dots) \mathbf{v}_f \dots) v \dots) \\
& \mapsto_{\text{map}0} (\text{frame } (n_f \dots) T_c) \\
& \text{where } 0 \in n_f \dots
\end{aligned}$$

$$\begin{aligned}
& (\text{s-app } (\lambda ((x \tau) \dots) e) v \dots) \\
& \mapsto_{\beta} \\
& e[x \mapsto v, \dots]
\end{aligned}$$

$$\begin{aligned}
& (\text{unbox } (\text{shp } n_f \dots) T_c \\
& \quad (x_i \dots x_e (\text{array } (n_f \dots) (\text{box } \iota_s \dots v_s))) e) \\
& \mapsto_{\text{unbox}} (\text{frame } (n_f \dots) e[x_i \mapsto \iota_s, \dots, x_e \mapsto v_s]) \\
& \text{where } 0 \notin n_f \dots
\end{aligned}$$

$$\begin{aligned}
& (\text{unbox } (\text{shp } n_f \dots) T_c \\
& \quad (x_i \dots x_e (\text{array } (n_f \dots) (\text{box } \iota_s \dots v_s))) e) \\
& \mapsto_{\text{unbox}0} (\text{frame } (n_f \dots) T_c) \\
& \text{where } 0 \in n_f \dots
\end{aligned}$$

Figure 11.2: Dynamic semantics for explicit iteration forms

$$\begin{aligned}
& (\text{t-map } (\text{shp } n_f \dots) T_c \\
& \quad (\text{array } (n_f \dots) \mathbf{v}_f \dots) \tau_a \dots) \\
& \mapsto_{tmap} (\text{frame } (n_f \dots) (\text{t-app } \mathbf{v}_f \tau_a \dots) \dots) \\
& \text{where } 0 \notin n_f \dots
\end{aligned}$$

$$\begin{aligned}
& (\text{t-map } (\text{shp } n_f \dots) T_c \\
& \quad (\text{array } (n_f \dots) \mathbf{v}_f \dots) \tau_a \dots) \\
& \mapsto_{tmap0} (\text{frame } (n_f \dots) T_c) \\
& \text{where } 0 \in n_f \dots
\end{aligned}$$

$$\begin{aligned}
& (\text{t-app } (\text{T}\lambda ((x k) \dots) e) \tau \dots) \\
& \mapsto_{t\beta} e[x \mapsto \tau, \dots]
\end{aligned}$$

$$\begin{aligned}
& (\text{i-map } (\text{shp } n_f \dots) T_c \\
& \quad (\text{array } (n_f \dots) \mathbf{v}_f \dots) \iota_a \dots) \\
& \mapsto_{imap} (\text{frame } (n_f \dots) (\text{i-app } \mathbf{v}_f \iota_a \dots) \dots) \\
& \text{where } 0 \notin n_f \dots
\end{aligned}$$

$$\begin{aligned}
& (\text{i-map } (\text{shp } n_f \dots) T_c \\
& \quad (\text{array } (n_f \dots) \mathbf{v}_f \dots) \tau_a \dots) \\
& \mapsto_{imap0} (\text{frame } (n_f \dots) T_c) \\
& \text{where } 0 \in n_f \dots
\end{aligned}$$

$$\begin{aligned}
& (\text{i-app } (\text{I}\lambda ((x \gamma) \dots) e) \iota \dots) \\
& \mapsto_{i\beta} e[x \mapsto \iota, \dots]
\end{aligned}$$

Figure 11.3: Dynamic semantics for optional forms



The translation rule for function application must identify the function and argument arrays’ frame expansions by subtracting their frame shapes,  $Shp_f$  and  $Shp_a \dots$  respectively, from the principal frame  $Shp_p$ . The “monus” (monoid minus) operator is used here as a meta-level operation, not part of the index language itself. Recall that in the universal fragment of the theory of shapes, the only way to ensure that one shape is a prefix of another is if listing their dimension and sub-shape components shows a prefix match. Then the remaining components of the longer shape are what must be added to the shorter shape to make them match. After the proper `rep` is performed, a `map` with the principal frame and the function’s output cell type will lead to the same result as executing the explicitly typed but implicitly iterative code. The `reps` and their resulting reduction steps collectively encode explicit Remora’s *lift* step, producing function and argument arrays with matching frames. Turning type and index application into explicit mapping requires less new code because the arguments are not arrays which must be replicated. We need only the function array’s frame (which is always the principal frame in a `t-app` or `i-app`) and the type or index abstraction’s result type. The `unbox` translation also needs the shape of  $e_s$ , the array of boxes, as its iteration space, but the result cell type is that of the body expression  $e_b$ .

The remaining translation rules are simply pass-through cases, translating all expression or atom subterms. There is no work to do on types or indices, though the pervasive type annotations which were needed to guide explicit Remora’s reduction semantics can finally be pared down.

The above presentation of translating from Remora’s implicit iteration to explicit iteration is designed for a compiler which retains static information. Shape information flows through the code as types and indices, rather than allowing arbitrary term-like computation in their positions. Any information carried in types remains available to guide static analysis.

A compiler which chooses instead to erase types eagerly, guided by Chapter 10, loses some information but can still produce explicitly iterative code, as described in Figure 11.5. However, it must make heavy use of primitive operators for querying and manipulating shapes. The meta-level  $\div$  is not generally usable on terms. Working from the assumption that the type-erased program was produced from well-typed code, we can implement a runtime shape subtraction function `%MONUS` as

```
(λ ((x 1) (y 1)) (drop (length x) y))
```

We also require a runtime function for selecting the longest of several shapes. This can be implemented in Remora as a reduction over a vector of boxed shape vectors, but there is no particular need to write the runtime system in Remora itself. Instead of building a `box` for each frame shape, we add a variable-arity `%longest` function to the runtime.<sup>33</sup> For the sake of brevity, the translation also uses a `let*` form with the conventional semantics. It can be desugared into  $\lambda$  and `s-app`. The long spine of

<sup>33</sup> Depending on a target machine’s calling convention and the number of arguments used in a `map`, this may turn out to behave much like assembling the array of boxed vectors.

$$\begin{aligned}
& \text{Expl}_{\mathcal{E}} \left[ \left[ e_f^{(A \rightarrow ((A \Upsilon_i I_i) \dots) (A \Upsilon_o I_o)) I_f)} e_a^{(A \Upsilon_i (++) \iota_a \iota_i)) \dots} \right] \right] \\
&= (\text{map } I_p \ (A \Upsilon_o I_o) \\
&\quad (\text{rep } I_f \ (I_p \dot{-} I_f) \ \text{Expl}_{\mathcal{E}} \llbracket e_f \rrbracket) \\
&\quad (\text{rep } I_a \ (I_p \dot{-} I_a) \ \text{Expl}_{\mathcal{E}} \llbracket e_a \rrbracket) \ \dots) \\
&\text{where } I_p = \bigsqcup \{ \iota_f, \iota_a \dots \}
\end{aligned}$$

$$\begin{aligned}
& \text{Expl}_{\mathcal{E}} \left[ \left[ (\text{t-app } e_f^{(A (\vee ((x k) \dots) T_f) I_f)} \ \tau_a \dots) \right] \right] \\
&= (\text{t-map } I_f \ T_f[x \mapsto \tau_a, \dots] \ \text{Expl}_{\mathcal{E}} \llbracket e_f \rrbracket \ \tau_a \ \dots)
\end{aligned}$$

$$\begin{aligned}
& \text{Expl}_{\mathcal{E}} \left[ \left[ (\text{i-app } e_f^{(A (\cap ((x \gamma) \dots) T_f) I_f)} \ \iota_a \dots) \right] \right] \\
&= (\text{i-map } I_f \ T_f[x \mapsto \iota_a, \dots] \ \text{Expl}_{\mathcal{E}} \llbracket e_f \rrbracket \ \iota_a \ \dots)
\end{aligned}$$

$$\begin{aligned}
& \text{Expl}_{\mathcal{E}} \left[ \left[ (\text{unbox } (x_i \dots x_e e_e^{(A (\Sigma ((x \gamma) \dots) \tau_s) I_f)} e_b^{T_b})) \right] \right] \\
&= (\text{unbox } I_f \ T_b \ (x_i \dots x_e \ \text{Expl}_{\mathcal{E}} \llbracket e_e \rrbracket) \ \text{Expl}_{\mathcal{E}} \llbracket e_b \rrbracket)
\end{aligned}$$

$$\begin{aligned}
& \text{Expl}_{\mathcal{E}} \llbracket x \rrbracket = x \\
& \text{Expl}_{\mathcal{E}} \llbracket (\text{array } (n \dots) \ a \dots) \rrbracket = (\text{array } (n \dots) \ \text{Expl}_{\mathcal{A}} \llbracket a \rrbracket \dots) \\
& \text{Expl}_{\mathcal{E}} \llbracket (\text{array } (n \dots) \ \Upsilon) \rrbracket = (\text{array } (n \dots) \ \Upsilon) \\
& \text{Expl}_{\mathcal{E}} \llbracket (\text{frame } (n \dots) \ e \dots) \rrbracket = (\text{frame } (n \dots) \ \text{Expl}_{\mathcal{E}} \llbracket e \rrbracket \dots) \\
& \text{Expl}_{\mathcal{E}} \llbracket (\text{frame } (n \dots) \ T) \rrbracket = (\text{frame } (n \dots) \ T)
\end{aligned}$$

$$\begin{aligned}
& \text{Expl}_{\mathcal{A}} \llbracket \mathbf{0} \rrbracket = \mathbf{0} \\
& \text{Expl}_{\mathcal{A}} \llbracket \mathbf{b} \rrbracket = \mathbf{b} \\
& \text{Expl}_{\mathcal{A}} \llbracket (\lambda ((x \ \tau) \dots) \ e) \rrbracket = (\lambda ((x \ \tau) \dots) \ \text{Expl}_{\mathcal{E}} \llbracket e \rrbracket) \\
& \text{Expl}_{\mathcal{A}} \llbracket (\text{T}\lambda ((x \ k) \dots) \ e) \rrbracket = (\text{T}\lambda ((x \ k) \dots) \ \text{Expl}_{\mathcal{E}} \llbracket e \rrbracket) \\
& \text{Expl}_{\mathcal{A}} \llbracket (\text{I}\lambda ((x \ \gamma) \dots) \ e) \rrbracket = (\text{I}\lambda ((x \ \gamma) \dots) \ \text{Expl}_{\mathcal{E}} \llbracket e \rrbracket) \\
& \text{Expl}_{\mathcal{A}} \llbracket (\text{box } \iota \dots \ e \ \tau) \rrbracket = (\text{box } \iota \dots \ \text{Expl}_{\mathcal{E}} \llbracket e \rrbracket \ \tau)
\end{aligned}$$

Figure 11.4: Converting explicit Remora's implicit iteration to explicit iteration

runtime calls in the function application case offers an estimate of the interpretive overhead that Remora’s type system allows to be compiled away. Less is needed for index application and unboxing because there is no frame matching involved in their computation. Index application still requires a `%monus` to determine the proper output cell shape, whereas the type-erased `unbox` form includes a result cell shape annotation to begin with. The remaining expression and atom forms are pass-through cases, requiring nothing more than translating subterms.

## 11.2 FURTHER STEPS TO A LOW-LEVEL LANGUAGE

With function application no longer carrying iterative control flow, there is no longer any distinction between the mechanics of term-level and index-level computation. Type indices describe data of the same variety that might arise from expressions: natural numbers and vectors containing them. We can therefore merge the term and index levels. Then  $I\lambda$  and `i-app` become ordinary  $\lambda$  and function application.

At this point, we have fulfilled the original promise of Remora’s type system: enabling static compilation of rank polymorphism’s implicit control flow. A compiler targeting an existing functional language could stop here, by implementing `map` and `rep` as functions provided by its runtime system. Standard functional compilation techniques, such as closure conversion and partial specialization of functions called on known arguments, behave as in conventional languages.

For code which does not use functions like `shape-of`, which allow the details of an array’s shape to leak into user-visible values, there is room for another trick in the representation of arrays. Although up to this point, the semantics of Remora has treated arrays as data which contains both a sequence of atoms and shape (a sequence of axis lengths), Remora’s type system makes shape information *static*. In order to break an array into cells, it is only truly necessary to know the number of atoms in each cell and the number of atoms in the entire array. Any further detail about the shape of the individual cells will already be present in the code which consumes it. The run-time representation of arrays can therefore be as simple as a flat buffer of atoms, with cell-polymorphic functions taking their shape arguments as products of dimensions. The exceptions are when shape information is meant to move from the index level to the term level and when data leaves the program.

An alternative array representation alleviates the cost of constructing some intermediate values by taking advantage of the fact that many primitive operations on arrays—such as `transpose` or `rotate` or even the `rep` special form—only rearrange the contents in some way without changing the individual elements themselves.

A low-level IR for dealing with arrays can represent an array value as one of

$$\begin{aligned}
& \text{Expl}_{\mathcal{E}} \llbracket (\widehat{e}_f (\widehat{e}_a \iota_a) \dots \iota_r) \rrbracket \\
&= (\text{let}^* ((x_f \text{Expl}_{\mathcal{E}} \llbracket \widehat{e}_f \rrbracket) \\
&\quad (x_a \text{Expl}_{\mathcal{E}} \llbracket \widehat{e}_a \rrbracket)) \dots \\
&\quad (x_{\text{ffrm}} (\text{s-app } \%shape\text{-of } x_f)) \\
&\quad (x_{\text{ashp}} (\text{s-app } \%shape\text{-of } x_a)) \dots \\
&\quad (x_{\text{afrm}} (\text{s-app } \%monus\ x_{\text{ashp}}\ \iota_a)) \dots \\
&\quad (x_{\text{pfrm}} (\text{s-app } \%longest\ x_{\text{ffrm}}\ x_{\text{afrm}} \dots))) \\
&\quad (\text{map } x_{\text{pfrm}} (\text{s-app } \%monus\ \iota_r\ x_{\text{pfrm}}) \\
&\quad (\text{rep } x_{\text{ffrm}}\ x_{\text{pfrm}}\ x_f) \\
&\quad (\text{rep } x_{\text{afrm}}\ x_{\text{pfrm}}\ x_a) \dots))
\end{aligned}$$

$$\begin{aligned}
& \text{Expl}_{\mathcal{E}} \llbracket (\text{i-app } \widehat{e} \iota_a \dots \iota_r) \rrbracket \\
&= (\text{let}^* ((x_e \text{Expl}_{\mathcal{E}} \llbracket \widehat{e} \rrbracket) \\
&\quad (x_f (\text{s-app } \%shape\text{-of } x_e))) \\
&\quad (\text{imap } x_f (\text{s-app } \%monus\ \iota_r\ x_f)\ x_e\ \iota_a \dots))
\end{aligned}$$

$$\begin{aligned}
& \text{Expl}_{\mathcal{E}} \llbracket (\text{unbox } (x_i \dots x_e \widehat{e}_e) \widehat{e}_b) \rrbracket \\
&= (\text{let } ((x_s \text{Expl}_{\mathcal{E}} \llbracket \widehat{e}_e \rrbracket)) \\
&\quad (\text{unbox } (\text{s-app } \%shape\text{-of } x_s)\ \iota_b \\
&\quad (x_i \dots x_e\ x_s) \\
&\quad \text{Expl}_{\mathcal{E}} \llbracket \widehat{e}_b \rrbracket))
\end{aligned}$$

$$\text{Expl}_{\mathcal{E}} \llbracket x \rrbracket = x$$

$$\text{Expl}_{\mathcal{E}} \llbracket (\text{array } (n \dots) \widehat{a} \dots) \rrbracket = (\text{array } (n \dots) \text{Expl}_{\mathcal{A}} \llbracket \widehat{a} \rrbracket \dots)$$

$$\text{Expl}_{\mathcal{E}} \llbracket (\text{frame } (n \dots) \widehat{e} \dots) \rrbracket = (\text{frame } (n \dots) \text{Expl}_{\mathcal{E}} \llbracket \widehat{e} \rrbracket \dots)$$

$$\text{Expl}_{\mathcal{A}} \llbracket \mathbf{o} \rrbracket = \mathbf{o}$$

$$\text{Expl}_{\mathcal{A}} \llbracket \mathbf{b} \rrbracket = \mathbf{b}$$

$$\text{Expl}_{\mathcal{A}} \llbracket (\lambda ((x \tau) \dots) \widehat{e}) \rrbracket = (\lambda ((x \tau) \dots) \text{Expl}_{\mathcal{E}} \llbracket \widehat{e} \rrbracket)$$

$$\text{Expl}_{\mathcal{A}} \llbracket (\text{I}\lambda ((x \gamma) \dots) \widehat{e}) \rrbracket = (\text{I}\lambda ((x \gamma) \dots) \text{Expl}_{\mathcal{E}} \llbracket \widehat{e} \rrbracket)$$

$$\text{Expl}_{\mathcal{A}} \llbracket (\text{box } \iota \dots \widehat{e}) \rrbracket = (\text{box } \iota \dots \text{Expl}_{\mathcal{E}} \llbracket \widehat{e} \rrbracket)$$

Figure 11.5: Converting erased Remora's implicit iteration to explicit iteration

- (manifest  $ptr_a$ )
- (view  $e_{idx} e_{array}$ )

A manifest array carries the actual atoms, so they can be directly accessed by pointer arithmetic. A view presents an underlying array (computed by  $e_{array}$ ) as another array by having the index tuple transformed by the index function (given by  $e_{idx}$ ). While composing a long chain of index-transformation functions can be expensive to do at run time, a compiler has the ability to compose functions statically via inlining.

Suppose we have the  $2 \times 3$  array  $[[1\ 2\ 3][4\ 5\ 6]]$ . The underlying data storage is a buffer at memory location  $p$  containing the numbers 1 through 6 in sequence. To extract an individual element from the matrix, we need to compute which offset into the buffer locates the desired element. Representing the element index as a tuple—row index and column index—the proper indexing function is  $(\lambda \langle x_0, x_1 \rangle. 3x_0 + x_1)$ . That is, we move 3 times the row number (each row is 3 elements long) plus the column number.

This means we can represent our  $2 \times 3$  matrix with:

$$e_1 = (\text{view } (\lambda \langle x_0, x_1 \rangle. 3x_0 + x_1) (\text{manifest } p))$$

Rotating it on its major axis effectively increments the row index and leaves the column index untouched, which can be written as:

$$e_2 = (\text{view } (\lambda \langle x_0, x_1 \rangle. \langle (x_0 + 1) \bmod 3, x_1 \rangle) e_1)$$

The rotated array could be reversed along its minor axis:

$$e_3 = (\text{view } (\lambda \langle x_0, x_1 \rangle. \langle x_0, 2 - x_1 \rangle) e_2)$$

Then we can replicate the 1-cells, to produce a  $2 \times 4 \times 3$  array, by dropping the index element which selects which row in a plane to use:

$$e_4 = (\text{view } (\lambda \langle x_0, x_1, x_2 \rangle. \langle x_0, x_2 \rangle) e_3)$$

This three-dimensional array can be transposed on its outer two axes:

$$e_5 = (\text{view } (\lambda \langle x_0, x_1, x_2 \rangle. \langle x_1, x_0, x_2 \rangle) e_4)$$

Nested views can be merged by composing their indexing functions. A formal theory for such transformations on APL's primitives was developed by Mullin [85]. Developing an optimizing compiler based on this theory is left to future work.



CONCLUSION

---

We began with the following claim:

The implicit, data-driven control structure of higher-order rank-polymorphic programs can be identified statically by a type system suitable for the programming style common in rank-polymorphic code.

In support of this thesis, this dissertation has presented the design of Remora, a higher-order rank-polymorphic programming language whose type system identifies the implicit iteration space associated with lifting computation pointwise over large aggregate input.

By isolating rank polymorphism from the incidental warts of prior languages such as APL and J, Remora can build on the better-developed foundation of  $\lambda$ -calculus. In doing so, Remora gains the ability to support higher-arity and first-class functions, which are held back (often by syntactic limitations) in past work. The core language has formally stated static and dynamic semantics. Typing rules identify how code written to operate on arrays of one rank will automatically lift to higher-rank input, and the dynamic semantics conforms to the predictions made by the typing rules. Iverson’s full “prefix agreement” rule is supported for arbitrary arity and extended to allow lifting higher-order functions by permitting arrays to appear in function position for application.

In light of the core language’s excessive annotation burden—types are often longer than the code they describe—this dissertation also gives a local type inference algorithm. Similar to early work on local type inference, the programmer is expected to identify intended types for values which are meant to be used polymorphically. The silver lining of this extra required work is a sort of enforced documentation. These annotations turn out to be rare in a corpus of sample code because most functions written in the middle of some larger function body are only used monomorphically. The common pattern of  $\eta$ -expanding a function with different input cell ranks in order to manipulate how it lifts over some arguments creates a one-off function which only needs to accept specific arguments.

Remora’s local type inference relies on Makanin modulo theories, a new constraint-solving method for integrating equations on strings with a nontrivial algebraic theory of string elements. In Remora, shapes of arrays (*i.e.*, iteration spaces) are represented as strings over terms from Presburger arithmetic: appendable sequences of addable natural numbers. Typical Remora code does not push the Makanin solver into expensive nondeterministic branching or long search paths. In exchange

for reasoning through string equalities over Presburger terms, a compiler for a rank-polymorphic language is relieved of the task of array-index bounds checking and the dependence analysis required to identify when a serial loop is legal to reorder or parallelize.

To finish fulfilling the promise of the type system and escape from the heavyweight run-time type information used by the core language’s formal semantics, this dissertation presents type-directed translation passes which eliminate the pervasive RTTI and convert from implicit to explicit iteration.

## 12.1 FUTURE DIRECTIONS

From the design for a programming language, there are many places to go. For general-purpose programming, the most immediately obvious direction is integrating Remora with a language suitable for the irregular control flow common to “suburb code,” the link between a core computation kernel and the user-facing application program which invokes it. The Racket-based embedding of Remora offers some experience in language interoperation, with code in `#lang remora/dynamic` able to not only invoke and lift arbitrary Racket procedures but also be used from general Racket programs via a library-like interface instead of Racket’s `#lang` facility.

Dataframes, column-labeled tables commonly used for exploratory data analysis in systems such as R [83] and Pandas [68], typically support a suite of operations similar to those offered by array-oriented languages. The key difference between arrays and dataframes is that dataframes are *heterogeneous* along the row axis. Extending Remora to include heterogeneous data in the form of field-named records allows dataframes to be built as vectors of records. If records are introduced and eliminated by functions, Remora’s own rank-polymorphic lifting automatically turns record construction and field projection into operations for constructing dataframes from columns of data and extracting columns by name. Conventional array operations such as `filter` and `scan` then cover the typical dataframe-manipulation tasks.

As a proof of concept, `#lang remora/dynamic` includes records, with constructor functions generated from field names. Elimination is performed via lenses [31, 53], with projection and update functions also generated from field names as needed. The key missing piece for integrating lens-based records into Remora is extending the type system. Row polymorphism [100] is needed in order to decouple record operations from the set of fields on which they *do not* operate. While integrating row polymorphism into Remora’s type inference strategy will require substantial new work, there has been some recent research interest in row typing beyond ML, including some with an eye towards eventual support for lenses [69].



The Makanin modulo theories solver uses a particularly old algorithm in a space which has received significant recent attention. Subsequent string-equation solving algorithms, such as searching for a sequence of equation rewrites [80, 81] or compression-based techniques [47, 48, 82] have tighter asymptotic cost bounds than Makanin. The major drawback for Remora’s purposes is that naïve use of them for handling reasoning about dimensions is having to nondeterministically choose the equivalence relation on dimensions up front—*i.e.*, before invoking string-equation logic. Future work may identify a way to delay this decision, as Makanin’s algorithm allows, making these newer, faster string decision procedures usable for Remora’s type inference.

There is room for developing a calculus with a more flexible reduction strategy than the formal semantics presented here. In particular, a more general  $\beta$ -reduction rule could justify transformations on Remora code that improve performance. While explicitly replicating argument cells works for specifying the behavior of rank polymorphism, efficient execution should avoid materializing large intermediate arrays or repeatedly gathering result cells from parallel execution units only to re-scatter them immediately afterward. The use of a general calculus for array indexing within a parallelizing compiler brings up new research questions because decisions about arrays’ memory layout create obligations for communication.



## BIBLIOGRAPHY

---

- [1] Martin Abadi et al. “TensorFlow: A System for Large-Scale Machine Learning.” In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283. URL: <https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf>.
- [2] Philip S. Abrams. “What’s Wrong with APL?” In: *Proceedings of Seventh International Conference on APL*. APL ’75. Pisa, Italy: ACM, 1975, pp. 1–8. DOI: [10.1145/800117.803777](https://doi.org/10.1145/800117.803777). URL: <http://doi.acm.org/10.1145/800117.803777>.
- [3] Philip Samuel Abrams. “An APL Machine.” AAI7022146. PhD thesis. Stanford, CA, USA, 1970.
- [4] John Backus. “Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs.” In: *Commun. ACM* 21.8 (Aug. 1978), pp. 613–641. ISSN: 0001-0782. DOI: [10.1145/359576.359579](https://doi.org/10.1145/359576.359579). URL: <http://doi.acm.org/10.1145/359576.359579>.
- [5] Henk Barendregt. “Introduction to Generalized Type Systems.” In: *Journal of Functional Programming* 1.2 (1991), pp. 125–154. DOI: [10.1017/S0956796800020025](https://doi.org/10.1017/S0956796800020025).
- [6] Robert Bernecky. “APEX, the APL parallel executor.” MA thesis. National Library of Canada= Bibliothèque nationale du Canada, 1999. URL: <http://hdl.handle.net/1807/11626>.
- [7] Guy Blelloch. *NESL: A Nested Data-Parallel Language (Version 3.1)*. Tech. rep. 1995.
- [8] Guy Blelloch and G Sabot. “Compiling Collection-Oriented Languages onto Massively Parallel Computers.” In: *Journal of Parallel and Distributed Computing* 8.2 (Aug. 1990), pp. 119–134.
- [9] Edwin Brady. “Idris, a General-Purpose Dependently Typed Programming Language: Design and Implementation.” In: *Journal of functional programming* 23.5 (2013), pp. 552–593.
- [10] Timothy Budd. *An APL compiler*. Springer-Verlag, 1988. ISBN: 0-387-96643-9.
- [11] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. “Accelerating Haskell Array Codes with Multicore GPUs.” In: *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*. DAMP

- '11. Austin, Texas, USA: ACM, 2011, pp. 3–14. ISBN: 978-1-4503-0486-3. DOI: [10.1145/1926354.1926358](https://doi.org/10.1145/1926354.1926358). URL: <http://doi.acm.org/10.1145/1926354.1926358>.
- [12] Tianqi Chen et al. “TVM: An Automated End-to-End Optimizing Compiler for Deep Learning.” In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594. ISBN: 978-1-939133-08-3. URL: <https://www.usenix.org/conference/osdi18/presentation/chen>.
- [13] Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. “NOVA: A Functional Language for Data Parallelism.” In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming. ARRAY'14*. Edinburgh, United Kingdom: ACM, 2014, 8:8–8:13. ISBN: 978-1-4503-2937-8. DOI: [10.1145/2627373.2627375](https://doi.org/10.1145/2627373.2627375). URL: <http://doi.acm.org/10.1145/2627373.2627375>.
- [14] Thierry Coquand and Gerard Huet. “The Calculus of Constructions.” In: *Inf. Comput.* 76.2-3 (Feb. 1988), pp. 95–120. ISSN: 0890-5401. DOI: [10.1016/0890-5401\(88\)90005-3](https://doi.org/10.1016/0890-5401(88)90005-3). URL: [http://dx.doi.org/10.1016/0890-5401\(88\)90005-3](http://dx.doi.org/10.1016/0890-5401(88)90005-3).
- [15] William Craig. “Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory.” In: *The Journal of Symbolic Logic* 22.3 (1957), pp. 269–285. ISSN: 00224812. URL: <http://www.jstor.org/stable/2963594>.
- [16] Martin Davis, George Logemann, and Donald Loveland. “A Machine Program for Theorem-Proving.” In: *Commun. ACM* 5.7 (July 1962), pp. 394–397. ISSN: 0001-0782. DOI: [10.1145/368273.368557](https://doi.org/10.1145/368273.368557). URL: <http://doi.acm.org/10.1145/368273.368557>.
- [17] Martin Davis and Hilary Putnam. “A Computing Procedure for Quantification Theory.” In: *J. ACM* 7.3 (July 1960), pp. 201–215. ISSN: 0004-5411. DOI: [10.1145/321033.321034](https://doi.org/10.1145/321033.321034). URL: <http://doi.acm.org/10.1145/321033.321034>.
- [18] Henri Auguste Delannoy. “Emploi de l'échiquier pour la résolution de divers problèmes de probabilité.” In: *Comptes rendus du Congrès annuel de l'Association française pour l'avancement des sciences*. Paris, France, Aug. 1889, pp. 43–52.
- [19] Stephen Dolan. “Algebraic Subtyping.” PhD thesis. University of Cambridge Computer Laboratory, Trinity College, UK, 2016.

- [20] J. Dunfield and Neel Krishnaswami. “Sound and Complete Bidirectional Typechecking for Higher-Rank Polymorphism and Indexed Types.” In: *Principles of Programming Languages (POPL)*. <http://www.cl.cam.ac.uk/~nk480/gadt.pdf>. Jan. 2019.
- [21] Joshua Dunfield and Neel Krishnaswami. *Bidirectional Typing*. 2019. arXiv: 1908.05839 [cs.PL].
- [22] Joshua Dunfield and Neelakantan R. Krishnaswami. “Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism.” In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. ICFP ’13*. Boston, Massachusetts, USA: ACM, 2013, pp. 429–442. ISBN: 978-1-4503-2326-0. DOI: [10.1145/2500365.2500582](https://doi.org/10.1145/2500365.2500582). URL: <http://doi.acm.org/10.1145/2500365.2500582>.
- [23] Joshua Dunfield and Frank Pfenning. “Tridirectional Typechecking.” In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL ’04*. Venice, Italy: ACM, 2004, pp. 281–292. ISBN: 1-58113-729-X. DOI: [10.1145/964001.964025](https://doi.org/10.1145/964001.964025). URL: <http://doi.acm.org/10.1145/964001.964025>.
- [24] Valery G Durnev. “Undecidability of the Positive  $\forall\exists^3$ -Theory of a Free Semigroup.” In: *Siberian Mathematical Journal* 36.5 (1995), pp. 917–929.
- [25] Valery Georgievich Durnev. “Positive theory of a free semigroup.” In: *Doklady Akademii Nauk*. Vol. 211. 4. Russian Academy of Sciences. 1973, pp. 772–774.
- [26] Martin Elsman and Martin Dybdal. “Compiling a Subset of APL Into a Typed Intermediate Language.” In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming. ARRAY’14*. Edinburgh, United Kingdom: ACM, 2014, 101:101–101:106. ISBN: 978-1-4503-2937-8. DOI: [10.1145/2627373.2627390](https://doi.org/10.1145/2627373.2627390). URL: <http://doi.acm.org/10.1145/2627373.2627390>.
- [27] Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin E. Oancea. “Static Interpretation of Higher-Order Modules in Futhark: Functional GPU Programming in the Large.” In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018), 97:1–97:30. ISSN: 2475-1421. DOI: [10.1145/3236792](https://doi.org/10.1145/3236792). URL: <http://doi.acm.org/10.1145/3236792>.
- [28] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. 1st. The MIT Press, 2009. ISBN: 0262062755, 9780262062756.

- [29] John T Feo, David C Cann, and Rodney R Oldehoeft. “A Report on the Sisal Language Project.” In: *Journal of Parallel and Distributed Computing* 10.4 (1990), pp. 349–366.
- [30] Matthew Flatt and PLT. *Reference: Racket*. Tech. rep. PLT-TR-2010-1. <https://racket-lang.org/tr1/>. PLT Design Inc., 2010.
- [31] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. “Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem.” In: *ACM Trans. Program. Lang. Syst.* 29.3 (May 2007). ISSN: 0164-0925. DOI: [10.1145/1232420.1232424](https://doi.org/10.1145/1232420.1232424). URL: <http://doi.acm.org/10.1145/1232420.1232424>.
- [32] Susan Lucille Gerhart. “Verification of APL Programs.” AAI7313936. PhD thesis. USA, 1972.
- [33] Jeremy Gibbons. “APLlicative Programming with Naperian Functors.” In: *Programming Languages and Systems*. Ed. by Hongseok Yang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 556–583. ISBN: 978-3-662-54434-1.
- [34] Clemens Greck and Sven-Bodo Scholz. “Accelerating APL Programs with SAC.” In: *SIGAPL APL Quote Quad* 29.2 (Dec. 1998), pp. 50–57. ISSN: 0163-6006. DOI: [10.1145/379277.312719](https://doi.org/10.1145/379277.312719). URL: <http://doi.acm.org/10.1145/379277.312719>.
- [35] Claudio Gutiérrez. “Satisfiability of Word Equations with Constants is in Exponential Space.” In: *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No. 98CB36280)*. IEEE. 1998, pp. 112–119.
- [36] Claudio Gutiérrez. “Solving Equations in Strings: On Makanin’s Algorithm.” In: *Latin American Symposium on Theoretical Informatics*. Springer. 1998, pp. 358–373.
- [37] Troels Henriksen, Martin Dybdal, Henrik Urms, Anna Sofie Kiehn, Daniel Gavin, Hjalte Abelskov, Martin Elsman, and Cosmin Oancea. “APL on GPUs: A TAIL from the Past, Scribbled in Futhark.” In: *Proceedings of the 5th International Workshop on Functional High-Performance Computing*. FHPC 2016. Nara, Japan: Association for Computing Machinery, 2016, pp. 38–43. ISBN: 9781450344333. DOI: [10.1145/2975991.2975997](https://doi.org/10.1145/2975991.2975997). URL: <https://doi.org/10.1145/2975991.2975997>.
- [38] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. “Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates.” In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI

2017. Barcelona, Spain: ACM, 2017, pp. 556–571. ISBN: 978-1-4503-4988-8. DOI: [10.1145/3062341.3062354](https://doi.org/10.1145/3062341.3062354). URL: <http://doi.acm.org/10.1145/3062341.3062354>.
- [39] R. Hindley. “The Principal Type-Scheme of an Object in Combinatory Logic.” In: *Transactions of the American Mathematical Society* 146 (1969), pp. 29–60. ISSN: 00029947. URL: <http://www.jstor.org/stable/1995158>.
- [40] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming.” In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: [10.1145/363235.363259](https://doi.org/10.1145/363235.363259). URL: <https://doi.org/10.1145/363235.363259>.
- [41] Eric Holk, William Byrd, Nilesh Mahajan, Jeremiah Willcock, Arun Chauhan, and Andrew Lumsdaine. “Declarative Parallel Programming for GPUs.” In: *PARCO*. 2011, pp. 297–304.
- [42] Aaron Hsu. “A Data Parallel Compiler Hosted on the GPU.” PhD thesis. USA, 2019.
- [43] Kenneth E. Iverson. *A Programming Language*. New York, NY, USA: John Wiley & Sons, Inc., 1962. ISBN: 0-471430-14-5.
- [44] Kenneth E. Iverson. “Notation As a Tool of Thought.” In: *Commun. ACM* 23.8 (Aug. 1980), pp. 444–465. ISSN: 0001-0782. DOI: [10.1145/358896.358899](http://doi.acm.org/10.1145/358896.358899). URL: <http://doi.acm.org/10.1145/358896.358899>.
- [45] Joxan Jaffar. “Minimal and Complete Word Unification.” In: *J. ACM* 37.1 (Jan. 1990), pp. 47–85. ISSN: 0004-5411. DOI: [10.1145/78935.78938](http://doi.acm.org/10.1145/78935.78938). URL: <http://doi.acm.org/10.1145/78935.78938>.
- [46] C. B. Jay. *The FISH Language Definition*. Tech. rep. University of Technology, Sydney, 1998.
- [47] Artur Jež. “Recompression: A Simple and Powerful Technique for Word Equations.” In: *J. ACM* 63.1 (Feb. 2016), 4:1–4:51. ISSN: 0004-5411. DOI: [10.1145/2743014](http://doi.acm.org/10.1145/2743014). URL: <http://doi.acm.org/10.1145/2743014>.
- [48] Artur Jež. “Word Equations in Nondeterministic Linear Space.” In: *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*. Ed. by Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl. Vol. 80. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 95:1–95:13. ISBN: 978-3-95977-041-5. DOI: [10.4230/LIPIcs.ICALP.2017.95](http://drops.dagstuhl.de/opus/volltexte/2017/7408). URL: <http://drops.dagstuhl.de/opus/volltexte/2017/7408>.

- [49] Ronald L. Johnston. “The Dynamic Incremental Compiler of APL\3000.” In: *Proceedings of the International Conference on APL: Part I. APL ’79*. New York, New York, USA: ACM, 1979, pp. 82–87. DOI: [10.1145/800136.804442](https://doi.org/10.1145/800136.804442). URL: <http://doi.acm.org/10.1145/800136.804442>.
- [50] Jsoftware, Inc. *Jsoftware: High-Performance Development Platform*. URL: <http://www.jsoftware.com/>.
- [51] Juhani Karhumäki, Filippo Mignosi, and Wojciech Plandowski. “The Expressibility of Languages and Relations by Word Equations.” In: *J. ACM* 47.3 (May 2000), pp. 483–505. ISSN: 0004-5411. DOI: [10.1145/337244.337255](https://doi.org/10.1145/337244.337255). URL: <http://doi.acm.org/10.1145/337244.337255>.
- [52] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. “Regular, Shape-Polymorphic, Parallel Arrays in Haskell.” In: *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming. ICFP ’10*. Baltimore, Maryland, USA: ACM, 2010, pp. 261–272.
- [53] Edward Kmett. *lens: Lenses, Folds and Traversals*. URL: <http://hackage.haskell.org/package/lens> (visited on 04/06/2019).
- [54] E. E. Kohlbecker and M. Wand. “Macro-by-Example: Deriving Syntactic Transformations from Their Specifications.” In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. POPL ’87*. Munich, West Germany: ACM, 1987, pp. 77–84. ISBN: 0-89791-215-2. DOI: [10.1145/41625.41632](https://doi.org/10.1145/41625.41632). URL: <http://doi.acm.org/10.1145/41625.41632>.
- [55] Antoni Kościelski and Leszek Pacholski. “Complexity of Makanin’s Algorithm.” In: *J. ACM* 43.4 (July 1996), pp. 670–684. ISSN: 0004-5411. DOI: [10.1145/234533.234543](https://doi.org/10.1145/234533.234543). URL: <http://doi.acm.org/10.1145/234533.234543>.
- [56] Kx Systems. *Kx kdb+ q Documentation*. URL: <https://code.kx.com/v2/>.
- [57] Ben Lippmeier, Manuel Chakravarty, Gabriele Keller, and Simon Peyton Jones. “Guiding Parallel Array Fusion with Indexed Types.” In: *Proceedings of the 2012 Haskell Symposium*. Haskell ’12. Copenhagen, Denmark: ACM, 2012, pp. 25–36. ISBN: 978-1-4503-1574-6. DOI: [10.1145/2364506.2364511](https://doi.org/10.1145/2364506.2364511). URL: <http://doi.acm.org/10.1145/2364506.2364511>.
- [58] Thibaut Lutz and Vinod Grover. “LambdaJIT: A Dynamic Compiler for Heterogeneous Optimizations of STL Algorithms.” In: *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional*



- High-performance Computing*. FHPC '14. Gothenburg, Sweden: ACM, 2014, pp. 99–108. ISBN: 978-1-4503-3040-4. DOI: [10.1145/2636228.2636233](https://doi.org/10.1145/2636228.2636233). URL: <http://doi.acm.org/10.1145/2636228.2636233>.
- [59] Gennady S Makanin. “The Problem of Solvability of Equations in a Free Semigroup.” In: *Sbornik: Mathematics* 32.2 (1977), pp. 129–198. DOI: [10.1070/SM1977v032n02ABEH002376](https://doi.org/10.1070/SM1977v032n02ABEH002376).
- [60] Panagiotis Manolios and Vasilis Papavasileiou. “ILP Modulo Theories.” In: *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044*. CAV 2013. Saint Petersburg, Russia: Springer-Verlag New York, Inc., 2013, pp. 662–677. ISBN: 978-3-642-39798-1. DOI: [10.1007/978-3-642-39799-8\\_44](https://doi.org/10.1007/978-3-642-39799-8_44). URL: [http://dx.doi.org/10.1007/978-3-642-39799-8\\_44](http://dx.doi.org/10.1007/978-3-642-39799-8_44).
- [61] SS Marchenkov. “Unsolvability of positive  $\forall\exists$ -theory of free semi-group.” In: *Sibirsky Matematicheskie Jurnal* 23.1 (1982), pp. 196–198.
- [62] Per Martin-Löf. *An intuitionistic theory of types*.
- [63] Inc. Mathworks. *Matlab, User's Guide*. Natick, MA, 1992.
- [64] Conor McBride. “Epigram: Practical Programming with Dependent Types.” In: *Proceedings of the 5th International Conference on Advanced Functional Programming*. AFP'04. Tartu, Estonia: Springer-Verlag, 2005, pp. 130–170. ISBN: 978-3-540-28540-3. DOI: [10.1007/11546382\\_3](https://doi.org/10.1007/11546382_3). URL: [http://dx.doi.org/10.1007/11546382\\_3](http://dx.doi.org/10.1007/11546382_3).
- [65] Conor McBride and Ross Paterson. “Applicative Programming with Effects.” In: *J. Funct. Program.* 18.1 (Jan. 2008), pp. 1–13. ISSN: 0956-7968. DOI: [10.1017/S0956796807006326](https://doi.org/10.1017/S0956796807006326). URL: <http://dx.doi.org/10.1017/S0956796807006326>.
- [66] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. “Optimising Purely Functional GPU Programs.” In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: ACM, 2013, pp. 49–60. ISBN: 978-1-4503-2326-0. DOI: [10.1145/2500365.2500595](https://doi.org/10.1145/2500365.2500595). URL: <http://doi.acm.org/10.1145/2500365.2500595>.
- [67] James McGraw, Stephen Skedzielewski, Stephen Allan, Dale Grit, Rob Oldehoeft, John Glauert, Ivan Dobes, and Paul Hohensee. *SISAL: Streams and Iteration in a Single-Assignment Language. Language Reference Manual, Version 1. 1*. Tech. rep. Lawrence Livermore National Lab., CA (USA), 1983.

- [68] Wes McKinney et al. “Data structures for statistical computing in python.” In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX. 2010, pp. 51–56.
- [69] J. Garrett Morris and James McKinna. “Abstracting Extensible Data Types: Or, Rows by Any Other Name.” In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: [10.1145/3290325](https://doi.org/10.1145/3290325). URL: <https://doi.org/10.1145/3290325>.
- [70] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. “Scalable Parallel Programming with CUDA.” In: *Queue* 6.2 (Mar. 2008), pp. 40–53. ISSN: 1542-7730. DOI: [10.1145/1365490.1365500](https://doi.org/10.1145/1365490.1365500). URL: <http://doi.acm.org/10.1145/1365490.1365500>.
- [71] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. “Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T).” In: *J. ACM* 53.6 (Nov. 2006), pp. 937–977. ISSN: 0004-5411. DOI: [10.1145/1217856.1217859](https://doi.org/10.1145/1217856.1217859). URL: <https://doi.org/10.1145/1217856.1217859>.
- [72] Ulf Norell. “Dependently Typed Programming in Agda.” In: *Proceedings of the 6th International Conference on Advanced Functional Programming*. AFP’08. Heijen, The Netherlands: Springer-Verlag, 2009, pp. 230–266. ISBN: 3-642-04651-7. URL: <http://dl.acm.org/citation.cfm?id=1813347.1813352>.
- [73] Travis E. Oliphant. *Guide to NumPy*. Trelgol Publishing USA, Dec. 2006.
- [74] Derek C. Oppen. “Elementary Bounds for Presburger Arithmetic.” In: *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*. STOC ’73. Austin, Texas, USA: ACM, 1973, pp. 34–37. DOI: [10.1145/800125.804033](https://doi.org/10.1145/800125.804033). URL: <http://doi.acm.org/10.1145/800125.804033>.
- [75] Richard J. Orgass. “Toward a Primitive Recursive Semantics for APL.” In: *Proceedings of the Eighth International Conference on APL*. APL ’76. Ottawa, Canada: Association for Computing Machinery, 1976, pp. 314–320. ISBN: 9781450374163. DOI: [10.1145/800114.803693](https://doi.org/10.1145/800114.803693). URL: <https://doi.org/10.1145/800114.803693>.
- [76] Emir Pasalic, Jeremy Siek, and Walid Taha. *Concoqtion: Mixing Dependent Types and Hindley-Milner Type Inference*. Tech. rep. 2006.
- [77] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library.” In: *Advances in Neural Information Processing Systems* 32. Ed. by H. Wallach, H.

- Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.
- [78] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [79] Benjamin C. Pierce and David N. Turner. “Local Type Inference.” In: *ACM Trans. Program. Lang. Syst.* 22.1 (Jan. 2000), pp. 1–44. ISSN: 0164-0925. DOI: [10.1145/345099.345100](https://doi.org/10.1145/345099.345100). URL: <http://doi.acm.org/10.1145/345099.345100>.
- [80] Wojciech Plandowski. “Satisfiability of Word Equations with Constants is in PSPACE.” In: vol. 51. 3. New York, NY, USA: ACM, May 2004, pp. 483–496. DOI: [10.1145/990308.990312](https://doi.org/10.1145/990308.990312). URL: <http://doi.acm.org/10.1145/990308.990312>.
- [81] Wojciech Plandowski. “An Efficient Algorithm for Solving Word Equations.” In: *Proceedings of the Thirty-eighth Annual ACM Symposium on Theory of Computing*. STOC '06. Seattle, WA, USA: ACM, 2006, pp. 467–476. ISBN: 1-59593-134-1. DOI: [10.1145/1132516.1132584](https://doi.org/10.1145/1132516.1132584). URL: <http://doi.acm.org/10.1145/1132516.1132584>.
- [82] Wojciech Plandowski and Wojciech Rytter. “Application of Lempel-Ziv Encodings to the Solution of Word Equations.” In: *Automata, Languages and Programming*. Ed. by Kim G. Larsen, Sven Skyum, and Glynn Winskel. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 731–742. ISBN: 978-3-540-68681-1.
- [83] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2013. URL: <http://www.R-project.org/>.
- [84] C. R. Reddy and D. W. Loveland. “Presburger Arithmetic with Bounded Quantifier Alternation.” In: *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*. STOC '78. San Diego, California, USA: ACM, 1978, pp. 320–325. DOI: [10.1145/800133.804361](https://doi.org/10.1145/800133.804361). URL: <http://doi.acm.org/10.1145/800133.804361>.
- [85] Lenore Marie Restifo Mullin. “A Mathematics of Arrays.” AAI8914581. PhD thesis. USA, 1988.
- [86] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. “Liquid Types.” In: *SIGPLAN Not.* 43.6 (June 2008), pp. 159–169. ISSN: 0362-1340. DOI: [10.1145/1379022.1375602](https://doi.org/10.1145/1379022.1375602). URL: <http://doi.acm.org/10.1145/1379022.1375602>.

- [87] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hege-  
man, Meghan Lele, Roman Levenstein, et al. “Glow: Graph  
lowering compiler techniques for neural networks.” In: *arXiv  
preprint arXiv:1805.00907* (2018).
- [88] Walter J. Savitch. “Relationships Between Nondeterministic and  
Deterministic Tape Complexities.” In: *Journal of Computer and  
System Sciences* 4.2 (1970), pp. 177–192. ISSN: 0022-0000. DOI:  
[https://doi.org/10.1016/S0022-0000\(70\)80006-X](https://doi.org/10.1016/S0022-0000(70)80006-X). URL:  
[http://www.sciencedirect.com/science/article/pii/  
S002200007080006X](http://www.sciencedirect.com/science/article/pii/S002200007080006X).
- [89] Sven-Bodo Scholz. “Single Assignment C: Efficient Support  
for High-Level Array Operations in a Functional Setting.” In: *J.  
Funct. Program.* 13.6 (Nov. 2003), pp. 1005–1059. ISSN: 0956-  
7968. URL: <https://doi.org/10.1017/S0956796802004458>.
- [90] Bo Joel Svensson, Michael Vollmer, Eric Holk, Trevor L. Mc-  
Donell, and Ryan R. Newton. “Converting Data-Parallelism  
to Task-Parallelism by Rewrites: Purely Functional Programs  
Across Multiple GPUs.” In: *Proceedings of the 4th ACM SIG-  
PLAN Workshop on Functional High-Performance Computing.  
FHPC 2015*. Vancouver, BC, Canada: ACM, 2015, pp. 12–22.  
ISBN: 978-1-4503-3807-3. DOI: [10.1145/2808091.2808093](https://doi.org/10.1145/2808091.2808093).  
URL: <http://doi.acm.org/10.1145/2808091.2808093>.
- [91] The Coq Development Team. *The Coq Proof Assistant, version  
8.9.0*. Jan. 2019. DOI: [10.5281/zenodo.2554024](https://doi.org/10.5281/zenodo.2554024). URL: <https://doi.org/10.5281/zenodo.2554024>.
- [92] The XLA Team. *XLA - TensorFlow, compiled*. URL: [https://developers.googleblog.com/2017/03/xla-tensorflow-  
compiled.html](https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html).
- [93] Kai Trojahnner and Clemens Grelck. “Dependently typed array  
programs don’t go wrong.” In: *Journal of Logic and Algebraic  
Programming* 78.7 (2009), pp. 643–664.
- [94] Hai-Chen Tu. “FAC: Functional Array Calculator and its Appli-  
cation to APL and Functional Programming.” PhD thesis. New  
Haven, CT, USA, 1986.
- [95] Hiroshi Unno and Naoki Kobayashi. “Dependent Type Inference  
with Interpolants.” In: *Proceedings of the 11th ACM SIGPLAN  
Conference on Principles and Practice of Declarative Program-  
ming*. PPDP ’09. Coimbra, Portugal: ACM, 2009, pp. 277–288.  
ISBN: 978-1-60558-568-0. DOI: [10.1145/1599410.1599445](https://doi.org/10.1145/1599410.1599445).  
URL: <http://doi.acm.org/10.1145/1599410.1599445>.

- [96] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. “Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions.” In: *arXiv preprint arXiv:1802.04730* (2018).
- [97] Dimitrios Vytiniotis and Stephanie Weirich. “Dependent types: Easy as PIE.” In: *8th Symp. on Trends in Functional Programming*. 2007.
- [98] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. “Boxy Types: Inference for Higher-Rank Types and Impredicativity.” In: *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*. ICFP ’06. Portland, Oregon, USA: ACM, 2006, pp. 251–262. ISBN: 1-59593-309-3. DOI: [10.1145/1159803.1159838](https://doi.org/10.1145/1159803.1159838). URL: <http://doi.acm.org/10.1145/1159803.1159838>.
- [99] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. “FPH: First-Class Polymorphism for Haskell.” In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’08. Victoria, BC, Canada: ACM, 2008, pp. 295–306. ISBN: 978-1-59593-919-7. DOI: [10.1145/1411204.1411246](https://doi.org/10.1145/1411204.1411246). URL: <http://doi.acm.org/10.1145/1411204.1411246>.
- [100] Mitchell Wand. “Type inference for record concatenation and multiple inheritance.” In: *Information and Computation* 93.1 (1991), pp. 1–15.
- [101] Zvi Weiss and Harry J. Saal. “Compile Time Syntax Analysis of APL Programs.” In: *Proceedings of the International Conference on APL*. APL ’81. San Francisco, California, USA: ACM, 1981, pp. 313–320. ISBN: 0-89791-035-4. DOI: [10.1145/800142.805380](https://doi.org/10.1145/800142.805380). URL: <http://doi.acm.org/10.1145/800142.805380>.
- [102] Hongwei Xi. “Dependent Types in Practical Programming.” AAI9918624. PhD thesis. Pittsburgh, PA, USA, 1998.
- [103] Hongwei Xi and Frank Pfenning. “Eliminating Array Bound Checking Through Dependent Types.” In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. PLDI ’98. Montreal, Quebec, Canada: ACM, 1998, pp. 249–257. ISBN: 0-89791-987-4. DOI: [10.1145/277650.277732](https://doi.org/10.1145/277650.277732). URL: <http://doi.acm.org/10.1145/277650.277732>.
- [104] Peter Zhang, Eric Holk, John Matty, Samantha Misurda, Marcin Zalewski, Jonathan Chu, Scott McMillan, and Andrew Lumsdaine. “Dynamic Parallelism for Simple and Efficient GPU Graph Algorithms.” In: *Proceedings of the 5th Workshop on*

*Irregular Applications: Architectures and Algorithms*. IA3 '15. Austin, Texas: ACM, 2015, 11:1–11:4. ISBN: 978-1-4503-4001-4. DOI: [10.1145/2833179.2833189](https://doi.org/10.1145/2833179.2833189). URL: <http://doi.acm.org/10.1145/2833179.2833189>.

Part IV

APPENDIX





## PROOFS (4.2: STATIC SEMANTICS)

---

**Lemma 4.2.2** (Preservation of sorts under index substitution). *If  $\Theta, x :: \gamma_x \vdash \iota :: \gamma$  and  $\Theta \vdash \iota_x :: \gamma_x$  then  $\Theta \vdash \iota[x \mapsto \iota_x] :: \gamma$ .*

*Proof.* We use induction on the sort derivation  $\Theta, x :: \gamma_x \vdash \iota :: \gamma$ .

**Case NAT**

$$\frac{n \in \mathbb{N}}{\Theta, x :: \gamma_1 \vdash n :: \text{Dim}} \text{ S:NAT}$$

Since  $\iota_0 = n$ ,  $\iota[x \mapsto \iota_x] = \iota_0$ , so the original derivation is still valid.

**Case VAR**

$$\frac{(x' :: \gamma) \in \Theta}{\Theta, x :: \gamma_x \vdash x' :: \gamma} \text{ S:VAR}$$

Then  $\iota$  has the form  $x'$ . If  $x' = x$ , then  $\iota[x \mapsto \iota_x] = \iota_1$ , and (by Lemma 4.2.1, uniqueness of sorting)  $\gamma = \gamma_x$ . By assumption,  $\Theta \vdash \iota_x :: \gamma_x$ , so the equality of  $\gamma$  and  $\gamma_x$  implies  $\Theta \vdash \iota_x :: \gamma$ . If  $x' \neq x$ , then  $\iota[x \mapsto \iota_x] = x$ , and we use the same S:VAR derivation we started with.

**Case SHAPE**

$$\frac{\Theta, x :: \gamma_x \vdash \iota_d :: \text{Dim} \dots}{\Theta, x :: \gamma_x \vdash (\text{shape } \iota_d \dots) :: \text{Shape}} \text{ S:SHAPE}$$

Then the induction hypothesis implies that  $\Theta \vdash \iota_d[x \mapsto \iota_x] :: \text{Dim}$  for each of the  $\iota_d \dots$ . So applying S:SHAPE derives

$$\frac{\Theta \vdash \iota_d[x \mapsto \iota_x] :: \text{Dim} \dots}{\Theta \vdash (\text{shape } \iota_d[x \mapsto \iota_x] \dots) :: \text{Shape}} \text{ S:SHAPE}$$

**Case PLUS**

$$\frac{\Theta, x :: \gamma_x \vdash \iota_d :: \text{Dim} \dots}{\Theta, x :: \gamma_x \vdash (+ \iota_d \dots) :: \text{Dim}} \text{ S:PLUS}$$

By the induction hypothesis,  $\Theta \vdash \iota_d[x \mapsto \iota_x] :: \text{Dim}$  for each of  $\iota_d \dots$ . Then we use their derivations to construct

$$\frac{\Theta \vdash \iota_d[x \mapsto \iota_x] :: \text{Dim} \dots}{\Theta \vdash (+ \iota_d[x \mapsto \iota_x] \dots) :: \text{Dim}} \text{ S:PLUS}$$

**Case APPEND**

$$\frac{\Theta, x :: \gamma_x \vdash \iota_s :: \text{Shape} \dots}{\Theta, x :: \gamma_x \vdash (++ \iota_s \dots) :: \text{Shape}} \text{ S:APPEND}$$

The induction hypothesis gives  $\Theta \vdash \iota_s[x \mapsto \iota_x] :: \text{Shape}$  for each of  $\iota_s \dots$ . Then we derive

$$\frac{\Theta \vdash \iota_s[x \mapsto \iota_x] :: \text{Shape} \dots}{\Theta \vdash ( ++ \iota_s[x \mapsto \iota_x] \dots ) :: \text{Shape}} \text{S:APPEND}$$

□

**Lemma 4.2.4** (Preservation of kinds under index substitution). *If  $\Theta, x :: \gamma; \Delta \vdash \tau :: k$  and  $\Theta \vdash \iota_x :: \gamma$  then  $\Theta; \Delta \vdash \tau[x \mapsto \iota_x] :: k$ .*

*Proof.* We use induction on the kind derivation  $\Theta, x :: \gamma; \Delta \vdash \tau :: k$ .

**Case VAR**

$$\frac{x' :: k \in \Delta}{\Theta, x :: \gamma; \Delta \vdash x' :: k} \text{K:VAR}$$

Since  $x$  cannot appear free in  $\tau$ ,  $x'[x \mapsto \iota_x] = x'$ , so the same kind is derivable:

$$\frac{x' :: k \in \Delta}{\Theta; \Delta \vdash x' :: k} \text{K:VAR}$$

**Case BASE**

$$\frac{}{\Theta, x :: \gamma; \Delta \vdash B :: k} \text{K:BASE}$$

Again,  $x$  cannot appear free in  $\tau$ , so we derive

$$\frac{}{\Theta; \Delta \vdash B :: k} \text{K:BASE}$$

**Case FN**

$$\frac{\Theta, x :: \gamma; \Delta \vdash \tau_i :: \text{Array} \dots \quad \Theta, x :: \gamma; \Delta \vdash \tau_o :: \text{Array}}{\Theta, x :: \gamma; \Delta \vdash (-> (\tau_i \dots) \tau_o) :: \text{Array}} \text{K:FN}$$

By the induction hypothesis, we have  $\Theta; \Delta \vdash \tau_i[x \mapsto \iota_x] :: \text{Array}$  for each of  $\tau_i \dots$ , and  $\Theta; \Delta \vdash \tau_o[x \mapsto \iota_x] :: \text{Array}$ . Then we can derive

$$\frac{\Theta; \Delta \vdash \tau_i[x \mapsto \iota_x] :: \text{Array} \dots \quad \Theta; \Delta \vdash \tau_o[x \mapsto \iota_x] :: \text{Array}}{\Theta; \Delta \vdash (-> (\tau_i[x \mapsto \iota_x] \dots) \tau_o[x \mapsto \iota_x]) :: \text{Array}} \text{K:FN}$$

**Case UNIV**

$$\frac{\Theta, x :: \gamma; \Delta, x_u :: k_u \dots \vdash \tau_u :: \text{Array}}{\Theta, x :: \gamma; \Delta \vdash (\forall ((x_u k_u) \dots) \tau_u) :: \text{Atom}} \text{K:UNIV}$$

The induction hypothesis implies

$$\Theta; \Delta, x_u :: k_u \dots \vdash \tau_u[x \mapsto \iota_x] :: \text{Array}$$

So we derive

$$\frac{\Theta; \Delta, x_u :: k_u \dots \vdash \tau_u[x \mapsto \iota_x] :: \text{Array}}{\Theta, x :: \gamma; \Delta \vdash (\forall ((x_u k_u) \dots) \tau_u[x \mapsto \iota_x]) :: \text{Atom}} \text{K:UNIV}$$

**Case PI**

$$\frac{\Theta, x :: \gamma, x_p :: \gamma_p \dots; \Delta \vdash \tau_p :: \text{Array}}{\Theta, x :: \gamma; \Delta \vdash (\prod ((x_p \gamma_p) \dots) \tau_p) :: \text{Atom}} \text{K:PI}$$

By the induction hypothesis,  $\Theta, x_p :: \gamma_p \dots; \Delta \vdash \tau_p[x \mapsto \iota_x] :: \text{Array}$ . Note that, following Barendregt's convention,  $x_p \dots$  are all unique and distinct from  $x$ . Then we construct the derivation

$$\frac{\Theta, x_p :: \gamma_p \dots; \Delta \vdash \tau_p[x \mapsto \iota_x] :: \text{Array}}{\Theta; \Delta \vdash (\prod ((x_p \gamma_p) \dots) \tau_p[x \mapsto \iota_x]) :: \text{Atom}} \text{K:PI}$$

**Case SIGMA**

$$\frac{\Theta, x :: \gamma, x_s :: \gamma_s \dots; \Delta \vdash \tau_s :: \text{Array}}{\Theta, x :: \gamma; \Delta \vdash (\Sigma ((x_s \gamma_s) \dots) \tau_s) :: \text{Atom}} \text{K:SIGMA}$$

As in the K:PI case, the induction hypothesis gives  $\Theta, x_s :: \gamma_s \dots; \Delta \vdash \tau_s[x \mapsto \iota_x] :: \text{Array}$ . Then we can derive

$$\frac{\Theta, x_s :: \gamma_s \dots; \Delta \vdash \tau_s[x \mapsto \iota_x] :: \text{Array}}{\Theta, x :: \gamma; \Delta \vdash (\Sigma ((x_s \gamma_s) \dots) \tau_s[x \mapsto \iota_x]) :: \text{Atom}} \text{K:SIGMA}$$

**Case ARRAY**

$$\frac{\Theta, x :: \gamma; \Delta \vdash \tau_a :: \text{Atom} \quad \Theta, x :: \gamma \vdash \iota_a :: \text{Shape}}{\Theta, x :: \gamma; \Delta \vdash (A \tau_a \iota_a) :: \text{Array}} \text{K:ARRAY}$$

The induction hypothesis implies that  $\Theta; \Delta \vdash \tau_a[x \mapsto \iota_x] :: \text{Atom}$ , and Lemma 4.2.2 (substitution in an index) implies  $\Theta \vdash \iota_a[x \mapsto \iota_x] :: \text{Shape}$ . So we then have

$$\frac{\Theta; \Delta \vdash \tau_a[x \mapsto \iota_x] :: \text{Atom} \quad \Theta \vdash \iota_a[x \mapsto \iota_x] :: \text{Shape}}{\Theta, x :: \gamma; \Delta \vdash (A \tau_a[x \mapsto \iota_x] \iota_a[x \mapsto \iota_x]) :: \text{Array}} \text{K:ARRAY}$$

□

**Lemma 4.2.5** (Preservation of kinds under type substitution). *Given  $\Theta; \Delta, x :: k_x \vdash \tau :: k$  and  $\Theta; \Delta \vdash \tau_x :: k_x$  then  $\Theta; \Delta \vdash \tau[x \mapsto \tau_x] :: k$ .*

*Proof.* We use induction on the kind derivation  $\Theta; \Delta, x :: k_x \vdash \tau :: k$ .

**Case VAR,  $\tau = x$**

$$\frac{(x :: k) \in \Delta, x :: k_x}{\Theta; \Delta, x :: k_x \vdash x :: k} \text{K:VAR}$$

Then  $\tau[x \mapsto \tau_x] = \tau_x$ , which has kind  $k_x$  by assumption.

**Case VAR,  $\tau = x' \neq x$**

$$\frac{(x' :: k) \in \Delta, x :: k_x}{\Theta; \Delta, x :: k_x \vdash x' :: k} \text{K:VAR}$$

Then  $\tau[x \mapsto \tau_x] = x'$ , and  $x' \neq x$  implies  $x' :: k \in \Delta$ . So we have

$$\frac{(x' :: k) \in \Delta}{\Theta; \Delta \vdash x' :: k} \text{K:VAR}$$

**Case BASE**

$$\frac{}{\Theta; \Delta, x :: k_x \vdash B :: \text{Atom}} \text{K:BASE}$$

Since  $x$  cannot appear free in  $\tau$ ,  $\tau[x \mapsto \tau_x] = \tau = B$ . The kinding rule K:BASE applies in any environment, so we have

$$\frac{}{\Theta; \Delta \vdash B :: \text{Atom}} \text{K:BASE}$$

**Case FN**

$$\frac{\Theta; \Delta, x :: k_x \vdash \tau_i :: \text{Array} \dots \quad \Theta; \Delta, x :: k_x \vdash \tau_o :: \text{Array}}{\Theta; \Delta, x :: k_x \vdash (-> (\tau_i \dots) \tau_o) :: \text{Atom}} \text{K:FN}$$

By the induction hypothesis,  $\Theta; \Delta \vdash \tau_i[x \mapsto \tau_x] :: \text{Array}$  for each of  $\tau_i \dots$ , and  $\Theta; \Delta \vdash \tau_o[x \mapsto \tau_x] :: \text{Array}$ . Then we derive

$$\frac{\Theta; \Delta \vdash \tau_i[x \mapsto \tau_x] :: \text{Array} \dots \quad \Theta; \Delta \vdash \tau_o[x \mapsto \tau_x] :: \text{Array}}{\Theta; \Delta \vdash (-> (\tau_i[x \mapsto \tau_x] \dots) \tau_o[x \mapsto \tau_x]) :: \text{Atom}} \text{K:FN}$$

**Case UNIV**

$$\frac{\Theta; \Delta, x :: k_x, x_u :: k_u \dots \vdash \tau_u :: \text{Array}}{\Theta; \Delta, x :: k_x \vdash (\forall ((x_u k_u) \dots) \tau_u) :: \text{Atom}} \text{K:UNIV}$$

The induction hypothesis implies

$$\Theta; \Delta, x_u :: k_u \dots \vdash \tau_u[x \mapsto \tau_x] :: \text{Array}$$

(again, Barendregt's convention promises that  $x \notin x_u \dots$ ). This leads to the derivation

$$\frac{\Theta; \Delta, x_u :: k_u \dots \vdash \tau_u[x \mapsto \tau_x] :: \text{Array}}{\Theta; \Delta \vdash (\forall ((x_u k_u) \dots) \tau_u[x \mapsto \tau_x]) :: \text{Atom}} \text{K:UNIV}$$

**Case PI**

$$\frac{\Theta, x_p :: \gamma_p \dots; \Delta, x :: k_x \vdash \tau_p :: \text{Array}}{\Theta; \Delta, x :: k_x \vdash (\Pi ((x_p \gamma_p) \dots) \tau_p) :: \text{Atom}} \text{K:PI}$$

By the induction hypothesis,  $\Theta, x_p :: \gamma_p \dots; \Delta \vdash \tau_p[x \mapsto \tau_x] :: \text{Array}$ . Then applying K:PI derives

$$\frac{\Theta, x_p :: \gamma_p \dots; \Delta \vdash \tau_p[x \mapsto \tau_x] :: \text{Array}}{\Theta; \Delta \vdash (\Pi ((x_p \gamma_p) \dots) \tau_p[x \mapsto \tau_x]) :: \text{Atom}} \text{K:PI}$$

**Case SIGMA**

$$\frac{\Theta, x_p :: \gamma_p \dots; \Delta, x :: k_x \vdash \tau_s :: \text{Array}}{\Theta; \Delta, x :: k_x \vdash (\Sigma ((x_s \gamma_s) \dots) \tau_s) :: \text{Atom}} \text{K:SIGMA}$$

The induction hypothesis gives  $\Theta, x_s :: \gamma_s \dots; \Delta \vdash \tau_s[x \mapsto \tau_x] :: \text{Array}$ . We then derive

$$\frac{\Theta, x_s :: \gamma_s \dots; \Delta \vdash \tau_s[x \mapsto \tau_x] :: \text{Array}}{\Theta; \Delta \vdash (\Sigma ((x_s \gamma_s) \dots) \tau_s[x \mapsto \tau_x]) :: \text{Atom}} \text{K:SIGMA}$$

**Case ARRAY**

$$\frac{\Theta; \Delta, x :: k_x \vdash \tau_a :: \text{Atom} \quad \Theta \vdash \iota_a :: \text{Shape}}{\Theta; \Delta, x :: k_x \vdash (A \tau_a \iota_a) :: \text{Array}} \text{K:ARRAY}$$

By the induction hypothesis,  $\Theta; \Delta \vdash \tau_a[x \mapsto \tau_x] :: \text{Atom}$ . Recycling the sort derivation for  $\iota_a$  as is, we derive

$$\frac{\Theta; \Delta \vdash \tau_a[x \mapsto \tau_x] :: \text{Atom} \quad \Theta \vdash \iota_a :: \text{Shape}}{\Theta; \Delta \vdash (A \tau_a[x \mapsto \tau_x] \iota_a) :: \text{Array}} \text{K:ARRAY}$$

□

**Lemma 4.2.6** (Canonical forms for atomic values). *Let  $\mathbf{v}$  be a well-typed atomic value, that is,  $\cdot; \cdot \vdash \mathbf{v} : \tau$ .*

1. *If  $\tau$  is of the form  $(\rightarrow (\tau_i \dots) \tau_o)$ , then  $\mathbf{v}$  is of the form  $\mathbf{v}$  or  $(\lambda ((x \tau_i) \dots) e)$ .*
2. *If  $\tau$  is of the form  $(\forall ((x k) \dots) \tau_u)$ , then  $\mathbf{v}$  is of the form  $(\top \lambda ((x_u k) \dots) e)$ .*

3. If  $\tau$  is of the form  $(\Pi ((x \gamma) \dots) \tau_p)$ ,  
then  $v$  is of the form  
 $(\text{I}\lambda ((x_p \gamma) \dots) e)$ .
4. If  $\tau$  is of the form  $(\Sigma ((x \gamma) \dots) \tau_b)$ ,  
then  $v$  is of the form  
 $(\text{box } \iota \dots v_b (\Sigma ((x_b \gamma) \dots) \tau'_b))$ ,  
with  $\tau \cong (\Sigma ((x_b \gamma) \dots) \tau'_b)$ .
5. If  $\tau$  is of the form  $B$ ,  
then  $v$  is of the form  
 $b$ .

*Proof.* A type derivation may end with a chain of uses of the T:EQV rule, but this chain must then be preceded by a use of some other rule. We consider the last non-T:EQV rule used in the derivation; the type ascribed by this rule must then be equivalent to the type ascribed by the full derivation. Types of two different forms (e.g., a function and a dependent sum) cannot be equivalent because no type equivalence rule can relate them.

1. The only rules capable of ascribing a function type to an atom are T:OP and T:LAM, which apply to atomic values of the form  $v$  and  $(\lambda ((x \tau_i) \dots) e)$  respectively.
2. Ascribing  $\tau$  to an atom requires T:TLAM, which applies only to an atomic value of the form  $(\text{T}\lambda ((x_u k) \dots) e)$ .
3. Ascribing  $\tau$  to an atom requires T:ILAM, which applies only to an atomic value of the form  $(\text{I}\lambda ((x_p \gamma) \dots) e)$ .
4. Ascribing  $\tau$  to an atom requires T:BOX, which applies only to an atomic value of the form  $(\text{box } \iota \dots v_b (\Sigma ((x_b \gamma) \dots) \tau'_b))$ . T:BOX ascribes the annotated type, but the full derivation may produce any equivalent type.
5. Only T:BASE can ascribe base type to an atom, and it applies only to literals of the appropriate type of base value.

□

**Lemma 4.2.7** (Canonical forms for arrays). *Let  $v$  be a well-typed value, that is,  $;\cdot; \vdash v : \tau$ ,*

1. If  $\tau$  is of the form  $(A (-> (\tau_i \dots) \tau_o) \iota)$ ,  
then  $v$  is of the form  
 $(\text{array } (n \dots) \bar{e} \dots)$ .
2. If  $\tau$  is of the form  $(A (\forall ((x k) \dots) \tau_u) \iota)$ ,  
then  $v$  is of the form  
 $(\text{array } (n \dots) (\text{T}\lambda ((x_u k) \dots) e) \dots)$ .

3. If  $\tau$  is of the form  $(A (\Pi ((x \gamma) \dots) \tau_p) \iota)$ ,  
then  $v$  is of the form  
 $(\text{array } (n \dots) (\text{I}\lambda ((x_p \gamma) \dots) e) \dots)$ .
4. If  $\tau$  is of the form  $(A (\Sigma ((x \gamma) \dots) \tau_b) \iota)$ ,  
then  $v$  is of the form  
 $(\text{array } (n \dots) (\text{box } \iota \dots v_b (\Sigma ((x_b \gamma) \dots) \tau_b)) \dots)$ ,  
with  $\tau \cong (\Sigma ((x_b \gamma) \dots) \tau'_b)$ .
5. If  $\tau$  is of the form  $(A B \iota)$ ,  
then  $v$  is of the form  
 $(\text{array } (n \dots) \mathfrak{b} \dots)$ ,  
with  $\cdot; \cdot; \cdot \vdash \mathfrak{b} : B$  for each of  $\mathfrak{b} \dots$ .

*Proof.* As for Lemma 4.2.6, the type derivation for  $v$  must at some point use a rule other than T:EQV to ascribe a type to  $v$ , and any subsequent use of T:EQV must preserve the form of that type. All of the types considered here have the form  $(A \tau_a \iota_a)$ ; array types must be ascribed by T:ARRAY. So  $v$  must be an array literal— $(\text{array } (n \dots) \mathfrak{v} \dots)$ —and we can shift to considering the forms of the atoms  $\mathfrak{v} \dots$ . Each case in this lemma corresponds to a particular case in Lemma 4.2.6.  $\square$

**Lemma 4.2.8** (Symmetry of  $\cong$ ). *If  $\tau \cong \tau'$ , then  $\tau' \cong \tau$ .*

*Proof.* We use a straightforward inductive argument on the derivation of  $\tau \cong \tau'$ .

**Case REFL:**

$$\frac{}{\tau \cong \tau} \text{TEQV:REFL}$$

Since  $\tau$  and  $\tau'$  are equal, the obligation is to show  $\tau \cong \tau$ , which we already have.

**Case ARRAY:**

$$\frac{\tau_a \cong \tau'_a \quad \models \iota_a \equiv \iota'_a}{(A \tau_a \iota_a) \cong (A \tau'_a \iota'_a)} \text{TEQV:ARRAY}$$

Index equality is symmetric, so  $\models \iota'_a \equiv \iota_a$ . The induction hypothesis gives  $\tau'_a \cong \tau_a$ . We then apply TEQV:ARRAY to construct

$$\frac{\tau'_a \cong \tau_a \quad \models \iota'_a \equiv \iota_a}{(A \tau'_a \iota'_a) \cong (A \tau_a \iota_a)} \text{TEQV:ARRAY}$$

**Case FN:**

$$\frac{\tau_i \cong \tau'_i \dots \quad \tau_o \cong \tau'_o}{(-> (\tau_i \dots) \tau_o) \cong (-> (\tau'_i \dots) \tau'_o)} \text{TEQV:FN}$$

The induction hypothesis gives  $\tau_i \cong \tau'_i, \dots, \tau_o \cong \tau'_o$ , so TEQV:FN produces

$$\frac{\tau'_i \cong \tau_i \dots \quad \tau'_o \cong \tau_o}{(-> (\tau'_i \dots) \tau'_o) \cong (-> (\tau_i \dots) \tau_o)} \text{TEQV:FN}$$

**Case UNIV:**

$$\frac{\tau_s[x \mapsto x_f, \dots] \cong \tau'_s[x' \mapsto x_f, \dots] \quad \text{with fresh } x_f \dots}{(\forall ((x \gamma) \dots) \tau_u) \cong (\forall ((x' \gamma) \dots) \tau'_u)} \text{TEQV:UNIV}$$

The induction hypothesis gives  $\tau'_u[x' \mapsto x_f, \dots] \cong \tau_u[x \mapsto x_f, \dots]$ , so we construct

$$\frac{\tau'_s[x' \mapsto x_f, \dots] \cong \tau_s[x \mapsto x_f, \dots] \quad \text{with fresh } x_f \dots}{(\forall ((x' \gamma) \dots) \tau'_u) \cong (\forall ((x \gamma) \dots) \tau_u)} \text{TEQV:UNIV}$$

**Case PI:**

$$\frac{\tau_s[x \mapsto x_f, \dots] \cong \tau'_s[x' \mapsto x_f, \dots] \quad \text{with fresh } x_f \dots}{(\prod ((x \gamma) \dots) \tau_p) \cong (\prod ((x' \gamma) \dots) \tau'_p)} \text{TEQV:PI}$$

The induction hypothesis gives  $\tau'_s[x' \mapsto x_f, \dots] \cong \tau_s[x \mapsto x_f, \dots]$ . We then derive

$$\frac{\tau'_p[x' \mapsto x_f, \dots] \cong \tau_p[x \mapsto x_f, \dots] \quad \text{with fresh } x_f \dots}{(\prod ((x' \gamma) \dots) \tau'_p) \cong (\prod ((x \gamma) \dots) \tau_p)} \text{TEQV:PI}$$

**Case SIGMA:**

$$\frac{\tau_s[x \mapsto x_f, \dots] \cong \tau'_s[x' \mapsto x_f, \dots] \quad \text{with fresh } x_f \dots}{(\Sigma ((x \gamma) \dots) \tau_s) \cong (\Sigma ((x' \gamma) \dots) \tau'_s)} \text{TEQV:SIGMA}$$

The induction hypothesis gives  $\tau'_s[x' \mapsto x_f, \dots] \cong \tau_s[x \mapsto x_f, \dots]$ , so we construct

$$\frac{\tau'_s[x' \mapsto x_f, \dots] \cong \tau_s[x \mapsto x_f, \dots] \quad \text{with fresh } x_f \dots}{(\Sigma ((x' \gamma) \dots) \tau'_s) \cong (\Sigma ((x \gamma) \dots) \tau_s)} \text{TEQV:SIGMA}$$

□

**Lemma 4.2.9** (Transitivity of  $\cong$ ). *If  $\tau_0 \cong \tau_1$  and  $\tau_1 \cong \tau_2$ , then  $\tau_0 \cong \tau_2$ .*

*Proof.* We use induction on the derivations of  $\tau_0 \cong \tau_1$  and  $\tau_1 \cong \tau_2$ . Since both derivations involve  $\tau_1$ , the structure of the equivalence rules prohibits the derivations from ending with different rules, unless one is TEQV:REFL. All non-reflexivity rules place incompatible restrictions on the types they find equivalent. If either derivation is by TEQV:REFL, then  $\tau_1$  is either  $\tau_0$  or  $\tau_2$ , so the conclusion  $\tau_0 \cong \tau_2$  is the same as one



of the assumptions. In the remaining cases, both derivations end with the same rule.

**Case ARRAY:**

$$\frac{\tau'_0 \cong \tau'_1 \quad \models \iota_0 \equiv \iota_1}{(A \tau'_0 \iota_0) \cong (A \tau'_1 \iota_1)} \text{TEQV:ARRAY}$$

and

$$\frac{\tau'_1 \cong \tau'_2 \quad \models \iota_1 \equiv \iota_2}{(A \tau'_1 \iota_1) \cong (A \tau'_2 \iota_2)} \text{TEQV:ARRAY}$$

Since  $\tau'_0 \cong \tau'_1$  and  $\tau'_1 \cong \tau'_2$ , the induction hypothesis implies  $\tau'_0 \cong \tau'_2$ . Transitivity of equality (on indices) implies  $\models \iota_0 \equiv \iota_2$ . So we can derive

$$\frac{\tau'_0 \cong \tau'_2 \quad \models \iota_0 \equiv \iota_2}{(A \tau'_0 \iota_0) \cong (A \tau'_2 \iota_2)} \text{TEQV:ARRAY}$$

**Case FN:**

$$\frac{\tau_{i_0} \cong \tau_{i_1} \dots \quad \tau_{o_0} \cong \tau_{o_1}}{(-> (\tau_{i_0} \dots) \tau_{o_0}) \cong (-> (\tau_{i_1} \dots) \tau_{o_1})} \text{TEQV:FN}$$

and

$$\frac{\tau_{i_1} \cong \tau_{i_2} \dots \quad \tau_{o_1} \cong \tau_{o_2}}{(-> (\tau_{i_1} \dots) \tau_{o_1}) \cong (-> (\tau_{i_2} \dots) \tau_{o_2})} \text{TEQV:FN}$$

We have equivalent argument types, via the induction hypothesis—each  $\tau_{i_0} \cong \tau_{i_1} \dots$  and  $\tau_{i_1} \cong \tau_{i_2} \dots$  implies  $\tau_{i_0} \cong \tau_{i_2} \dots$ . We also have equivalent result types,  $\tau_{o_0} \cong \tau_{o_2} \dots$ . Then TEQV:FN gives

$$\frac{\tau_{i_0} \cong \tau_{i_2} \dots \quad \tau_{o_0} \cong \tau_{o_2}}{(-> (\tau_{i_0} \dots) \tau_{o_0}) \cong (-> (\tau_{i_2} \dots) \tau_{o_2})} \text{TEQV:FN}$$

**Case UNIV:**

$$\frac{\tau_{u_0}[x_0 \mapsto x_f, \dots] \cong \tau_{u_1}[x_1 \mapsto x_f, \dots]}{(\forall ((x_0 k) \dots) \tau_{u_0}) \cong (\forall ((x_1 k) \dots) \tau_{u_1})} \text{TEQV:UNIV}$$

and

$$\frac{\tau_{u_1}[x_1 \mapsto x_f, \dots] \cong \tau_{u_2}[x_2 \mapsto x_f, \dots]}{(\forall ((x_1 k) \dots) \tau_{u_1}) \cong (\forall ((x_2 k) \dots) \tau_{u_2})} \text{TEQV:UNIV}$$

The induction hypothesis relates  $\tau_{u_0}[x_0 \mapsto x_f, \dots]$  to  $\tau_{u_2}[x_2 \mapsto x_f, \dots]$ , so we can construct the derivation

$$\frac{\tau_{u_0}[x_0 \mapsto x_f, \dots] \cong \tau_{u_2}[x_2 \mapsto x_f, \dots]}{(\forall ((x_0 k) \dots) \tau_{u_0}) \cong (\forall ((x_2 k) \dots) \tau_{u_2})} \text{TEQV:UNIV}$$

**Case PI:**

$$\frac{\tau_{p_0}[x_0 \mapsto x_f, \dots] \cong \tau_{p_1}[x_1 \mapsto x_f, \dots]}{(\prod ((x_0 \gamma) \dots) \tau_{p_0}) \cong (\prod ((x_1 \gamma) \dots) \tau_{p_1})} \text{TEQV:PI}$$

and

$$\frac{\tau_{p_1}[x_1 \mapsto x_f, \dots] \cong \tau_{p_2}[x_2 \mapsto x_f, \dots]}{(\prod ((x_1 k) \dots) \tau_{u_1}) \cong (\prod ((x_2 k) \dots) \tau_{u_2})} \text{TEQV:PI}$$

The induction hypothesis implies the equivalence of  $\tau_{u_0}[x_0 \mapsto x_f, \dots]$  and  $\tau_{u_2}[x_2 \mapsto x_f, \dots]$ , so we can apply TEQV:PI:

$$\frac{\tau_{u_0}[x_0 \mapsto x_f, \dots] \cong \tau_{u_2}[x_2 \mapsto x_f, \dots]}{(\prod ((x_0 k) \dots) \tau_{u_0}) \cong (\prod ((x_2 k) \dots) \tau_{u_2})} \text{TEQV:PI}$$

**Case SIGMA:**

$$\frac{\tau_{s_0}[x_0 \mapsto x_f, \dots] \cong \tau_{s_1}[x_1 \mapsto x_f, \dots]}{(\sum ((x_0 \gamma) \dots) \tau_{s_0}) \cong (\sum ((x_1 \gamma) \dots) \tau_{s_1})} \text{TEQV:SIGMA}$$

and

$$\frac{\tau_{s_1}[x_1 \mapsto x_f, \dots] \cong \tau_{s_2}[x_2 \mapsto x_f, \dots]}{(\sum ((x_1 \gamma) \dots) \tau_{s_1}) \cong (\sum ((x_2 \gamma) \dots) \tau_{s_2})} \text{TEQV:SIGMA}$$

By the induction hypothesis,  $\tau_{s_0}[x_0 \mapsto x_f, \dots] \cong \tau_{s_2}[x_2 \mapsto x_f, \dots]$ , allowing us to derive

$$\frac{\tau_{s_0}[x_0 \mapsto x_f, \dots] \cong \tau_{s_2}[x_2 \mapsto x_f, \dots]}{(\sum ((x_0 \gamma) \dots) \tau_{s_0}) \cong (\sum ((x_2 \gamma) \dots) \tau_{s_2})} \text{TEQV:SIGMA}$$

□

**Lemma 4.2.10.** *If  $\Theta; \Delta \vdash \tau :: k$  and  $\tau \cong \tau'$ , then  $\Theta; \Delta \vdash \tau' :: k$ .*

*Proof.* We use induction on the derivation of  $\tau \cong \tau'$ .

**Case REFL:**

$$\frac{\text{---}}{\tau \cong \tau} \text{TEQV:REFL}$$

Then  $\tau = \tau'$ , so the kind derivation for  $\tau$  applies to  $\tau'$  as well.

**Case ARRAY:**

$$\frac{\tau_a \cong \tau'_a \quad \models \iota_a \equiv \iota'_a}{(A \tau_a \iota_a) \cong (A \tau'_a \iota'_a)} \text{TEQV:ARRAY}$$

The kind derivation for  $\tau$  must have the form

$$\frac{\Theta; \Delta \vdash \tau_a :: \text{Atom} \quad \Theta \vdash \iota_a :: \text{Shape}}{\Theta; \Delta \vdash (A \tau_a \iota_a) :: \text{Array}} \text{K:ARRAY}$$

Similarly, equivalent indices must be of the same sort (dimensions and shapes are distinct objects in our universe of type indices). Applying the induction hypothesis to our kinding result for  $\tau'_a$  gives  $\Theta; \Delta \vdash \tau'_a :: \text{Atom}$ . Then we can derive for  $\tau'$ :

$$\frac{\Theta; \Delta \vdash \tau'_a :: \text{Atom} \quad \Theta \vdash l'_a :: \text{Shape}}{\Theta; \Delta \vdash (\text{A } \tau'_a l'_a) :: \text{Array}} \text{K:ARRAY}$$

**Case FN:**

$$\frac{\tau_i \cong \tau'_i \dots \quad \tau_o \cong \tau'_o}{(-> (\tau_i \dots) \tau_o) \cong (-> (\tau'_i \dots) \tau'_o)} \text{TEQV:FN}$$

The kinding derivation for  $\tau$  has the form

$$\frac{\Theta; \Delta \vdash \tau_i :: \text{Array} \dots \quad \Theta; \Delta \vdash \tau_o :: \text{Array}}{\Theta; \Delta \vdash (-> (\tau_i \dots) \tau_o) :: \text{Atom}} \text{K:FN}$$

By the induction hypothesis, we have kind derivations ascribing Array to each of the  $\tau_i \dots$  and  $\tau_o$ . So we can then derive

$$\frac{\Theta; \Delta \vdash \tau'_i :: \text{Array} \dots \quad \Theta; \Delta \vdash \tau'_o :: \text{Array}}{\Theta; \Delta \vdash (-> (\tau'_i \dots) \tau'_o) :: \text{Atom}} \text{K:FN}$$

**Case UNIV:**

$$\frac{\tau_u[x \mapsto x_f, \dots] \cong \tau'_u[x' \mapsto x_f, \dots] \quad \text{with fresh } x_f \dots}{(\forall ((x k) \dots) \tau_u) \cong (\forall ((x' k) \dots) \tau'_u)} \text{TEQV:UNIV}$$

The kind derivation for  $\tau$  must have the form

$$\frac{\Theta; \Delta, x :: k \dots \vdash \tau_u :: \text{Array}}{\Theta; \Delta \vdash (\forall ((x k) \dots) \tau_u) :: \text{Atom}} \text{K:UNIV}$$

The premise implies in turn via  $\alpha$ -conversion that

$$\Theta; \Delta, x_f :: k \dots \vdash \tau_u[x \mapsto x_f, \dots] :: \text{Array}$$

By the induction hypothesis, we have

$$\Theta; \Delta, x_f :: k \dots \vdash \tau'_u[x' \mapsto x_f, \dots] :: \text{Array}$$

We use  $\alpha$ -conversion again, to convert the derived type  $\tau'_u[x' \mapsto x_f, \dots]$  into  $(\tau'_u[x' \mapsto x_f, \dots])[x_f \mapsto x', \dots]$ , which is  $\tau'_u$ . This means that

$$\Theta; \Delta, x' :: k \dots \vdash \tau'_u :: \text{Array}$$

Then we can derive

$$\frac{\Theta; \Delta, x' :: k \dots \vdash \tau'_u :: \text{Array}}{\Theta; \Delta \vdash (\forall ((x' k) \dots) \tau'_u) :: \text{Atom}} \text{K:UNIV}$$

**Case PI:**

$$\frac{\tau_p[x \mapsto x_f, \dots] \cong \tau'_p[x' \mapsto x_f, \dots] \quad \text{with fresh } x_f \dots}{(\Pi((x \ k) \dots) \tau_p) \cong (\Pi((x' \ k) \dots) \tau'_p)} \text{TEQV:PI}$$

Deriving a kind for  $\tau$  must end with

$$\frac{\Theta, x :: \gamma \dots; \Delta \vdash \tau_p :: \text{Array}}{\Theta; \Delta \vdash (\Pi((x \ k) \dots) \tau_p) :: \text{Atom}} \text{K:PI}$$

By  $\alpha$ -converting  $x \dots$  to  $x_f \dots$ , we get the judgment

$$\Theta, x_f :: \gamma \dots; \Delta \vdash \tau_p[x \mapsto x_f, \dots] :: \text{Array}$$

This implies via the induction hypothesis that

$$\Theta, x_f :: \gamma \dots; \Delta \vdash \tau'_p[x' \mapsto x_f, \dots] :: \text{Array}$$

Further  $\alpha$ -conversion gives

$$\Theta, x' :: \gamma \dots; \Delta \vdash \tau'_p :: \text{Array}$$

So we derive

$$\frac{\Theta, x' :: \gamma \dots; \Delta \vdash \tau'_p :: \text{Array}}{\Theta; \Delta \vdash (\Pi((x' \ k) \dots) \tau'_p) :: \text{Atom}} \text{K:PI}$$

**Case SIGMA:**

$$\frac{\tau_s[x \mapsto x_f, \dots] \cong \tau'_s[x' \mapsto x_f, \dots] \quad \text{with fresh } x_f \dots}{(\Sigma((x \ k) \dots) \tau_s) \cong (\Sigma((x' \ k) \dots) \tau'_s)} \text{TEQV:SIGMA}$$

We also have a kind derivation for  $\tau$

$$\frac{\Theta, x :: \gamma \dots; \Delta \vdash \tau_s :: \text{Array}}{\Theta; \Delta \vdash (\Sigma((x \ k) \dots) \tau_s) :: \text{Atom}} \text{K:SIGMA}$$

As in the previous two cases, applying the induction hypothesis under  $\alpha$ -conversion gives  $\Theta, x' :: \gamma \dots; \Delta \vdash \tau'_s :: \text{Array}$ , leading to the derivation

$$\frac{\Theta, x' :: \gamma \dots; \Delta \vdash \tau'_s :: \text{Array}}{\Theta; \Delta \vdash (\Sigma((x' \ k) \dots) \tau'_s) :: \text{Atom}} \text{K:SIGMA}$$

□

**Lemma 4.2.11.** *If  $\models \iota \equiv \iota'$ , then for any index variable  $x$ ,  $\tau[x \mapsto \iota] \cong \tau[x \mapsto \iota']$ .*

*Proof.* This is provable using induction on the structure of  $\tau$ . Only the case for arrays makes direct use of  $\iota$  and  $\iota'$ ; the other cases simply use the induction hypothesis to prove the premises of the derivation of  $\tau[x \mapsto \iota] \cong \tau[x \mapsto \iota']$ .

**Case ARRAY:**

$\tau = (\text{A } \tau_a \iota_a)$ , so the induction hypothesis implies that  $\tau_a[x \mapsto \iota] \cong \tau_a[x \mapsto \iota']$ . The index equality  $\models \iota_a[x \mapsto \iota] \equiv \iota_a[x \mapsto \iota']$  follows from the substitution lemma of first-order logic. Then we can construct the type equivalence derivation

$$\frac{\tau_a[x \mapsto \iota] \cong \tau_a[x \mapsto \iota'] \quad \models \iota_a[x \mapsto \iota] \equiv \iota_a[x \mapsto \iota']}{(\text{A } \tau_a[x \mapsto \iota] \iota_a[x \mapsto \iota]) \cong (\text{A } \tau_a[x \mapsto \iota'] \iota_a[x \mapsto \iota'])} \text{TEQV:ARRAY}$$

□

**Theorem 4.2.2.** *If  $\tau \cong \tau'$  and  $\tau_x \cong \tau'_x$ , then for any type variable  $x$ ,  $\tau[x \mapsto \tau_x] \cong \tau'[x \mapsto \tau'_x]$ .*

*Proof.* We use induction on the derivation of  $\tau \cong \tau'$ .

**Case REFL:**

$\tau = \tau'$ . By Lemma 4.2.12,  $\tau[x \mapsto \tau_x] \cong \tau[x \mapsto \tau'_x]$ .

**Case ARRAY:**

$$\frac{\tau_a \cong \tau'_a \quad \models \iota_a \equiv \iota'_a}{(\text{A } \tau_a \iota_a) \cong (\text{A } \tau'_a \iota'_a)} \text{TEQV:ARRAY}$$

The induction hypothesis gives  $\tau_a[x \mapsto \tau_x] \cong \tau'_a[x \mapsto \tau'_x]$ , so we derive

$$\frac{\tau_a[x \mapsto \tau_x] \cong \tau'_a[x \mapsto \tau'_x] \quad \models \iota_a \equiv \iota'_a}{(\text{A } \tau_a[x \mapsto \tau_x] \iota_a) \cong (\text{A } \tau'_a[x \mapsto \tau'_x] \iota'_a)} \text{TEQV:ARRAY}$$

**Case FN:**

$$\frac{\tau_i \cong \tau'_i \dots \quad \tau_o \cong \tau'_o}{(\rightarrow (\tau_i \dots) \tau_o) \cong (\rightarrow (\tau'_i \dots) \tau'_o)} \text{TEQV:FN}$$

The induction hypothesis gives derivations for equivalence of corresponding input types after substitution,  $\tau_i[x \mapsto \tau_x] \cong \tau'_i[x \mapsto \tau'_x] \dots$ , as well as equivalence for output types,  $\tau_o[x \mapsto \tau_x] \cong \tau'_o[x \mapsto \tau'_x]$ . Then applying TEQV:FN gives

$$\frac{\tau_i[x \mapsto \tau_x] \cong \tau'_i[x \mapsto \tau'_x] \dots \quad \tau_o[x \mapsto \tau_x] \cong \tau'_o[x \mapsto \tau'_x]}{(\rightarrow (\tau_i[x \mapsto \tau_x] \dots) \tau_o[x \mapsto \tau_x]) \cong (\rightarrow (\tau'_i[x \mapsto \tau'_x] \dots) \tau'_o[x \mapsto \tau'_x])} \text{TEQV:FN}$$

**Case UNIV:**

$$\frac{\tau_u[x_u \mapsto x_f, \dots] \cong \tau'_u[x'_u \mapsto x_f, \dots]}{(\forall ((x_u k) \dots) \tau_u) \cong (\forall ((x'_u k) \dots) \tau'_u)} \text{TEQV:UNIV}$$

If  $x$  is shadowed by  $x_u \dots$  but not by  $x'_u \dots$  (or vice versa), then the universal types  $\tau$  and  $\tau'$  will not be equivalent, as a type variable is only equivalent to itself (via TEQV:REFL). If  $x$  does not appear at all in both  $\tau_u$  and  $\tau'_u$ , then substitution leaves them unchanged, and our required conclusion is an initial assumption. So we now assume that  $x$  is free in both  $\tau_u$  and  $\tau'_u$ . When we substitute in  $\tau_x$  and  $\tau'_x$ , the induction hypothesis implies

$$\tau_u[x_u \mapsto x_f, \dots][x \mapsto \tau_x] \cong \tau'_u[x'_u \mapsto x_f, \dots][x \mapsto \tau'_x]$$

Lack of shadowing means the substitutions commute, *i.e.*, these types are equal to  $\tau_u[x \mapsto \tau_x][x_u \mapsto x_f, \dots]$  and  $\tau'_u[x \mapsto \tau'_x][x'_u \mapsto x_f, \dots]$  respectively. So we then use TEQV:UNIV

$$\frac{\tau_u[x \mapsto \tau_x][x_u \mapsto x_f, \dots] \cong \tau'_u[x \mapsto \tau'_x][x'_u \mapsto x_f, \dots]}{(\forall ((x_u k) \dots) \tau_u[x \mapsto \tau_x]) \cong (\forall ((x'_u k) \dots) \tau'_u[x \mapsto \tau'_x])} \text{TEQV:UNIV}$$

**Case PI:**

$$\frac{\tau_p[x_p \mapsto x_f, \dots] \cong \tau'_p[x'_p \mapsto x_f, \dots]}{(\prod ((x_p \gamma)) \tau_p) \cong (\prod ((x'_p \gamma)) \tau'_p)} \text{TEQV:PI}$$

Shadowing  $x$  is no longer a concern because it is a type variable—the dependent product only binds index variables. Substituting  $\tau_x$  and  $\tau'_x$  into the  $\alpha$ -converted body types produces  $\tau_p[x_p \mapsto x_f, \dots][x \mapsto \tau_x]$  and  $\tau'_p[x'_p \mapsto x_f, \dots][x \mapsto \tau'_x]$ , which the induction hypothesis implies are equivalent. They are in turn equal to  $\tau_p[x \mapsto \tau_x][x_p \mapsto x_f, \dots]$  and  $\tau'_p[x \mapsto \tau'_x][x'_p \mapsto x_f, \dots]$  respectively. We then construct the derivation

$$\frac{\tau_p[x \mapsto \tau_x][x_p \mapsto x_f, \dots] \cong \tau'_p[x \mapsto \tau'_x][x'_p \mapsto x_f, \dots]}{(\prod ((x_p \gamma)) \tau_p[x \mapsto \tau_x]) \cong (\prod ((x_p \gamma)) \tau'_p[x \mapsto \tau'_x])} \text{TEQV:PI}$$

**Case SIGMA:**

This case proceeds as the PI case.  $\square$

**Theorem 4.2.3** (Uniqueness of typing, up to equivalence). *If  $\Theta; \Delta; \Gamma \vdash t : \tau$  and  $\Theta; \Delta; \Gamma \vdash t : \tau'$ , then  $\tau \cong \tau'$ .*

*Proof.* Any derivation of  $\Theta; \Delta; \Gamma \vdash t : \tau$  can be followed with zero or more additional uses of T:EQV or have any terminal uses of T:EQV stripped off to derive another type. Since type equivalence is transitive, any alternative type  $\tau'$  derived in this way is equivalent to  $\tau$ . It remains to show, by induction on  $t$ , that all derivations for  $\Theta; \Delta; \Gamma \vdash t : \tau'$  ending in a non-EQV rule ascribe a  $\tau'$  equivalent to  $\tau$ .

VARIABLE:  $t = x$ , so the only applicable rule is T:VAR, which ascribes  $\Gamma(x)$ .

ARRAY:  $t = (\text{array } (n \dots) \mathfrak{a} \dots)$ . Only T:ARRAY can derive a type, so the derivations for  $\tau$  and  $\tau'$  are

$$\frac{\Theta; \Delta; \Gamma \vdash \mathfrak{a} : \tau_a \dots \quad \Theta; \Delta \vdash \tau_a :: \text{Atom} \quad \text{Length} \llbracket \mathfrak{a} \dots \rrbracket = \prod n \dots}{\Theta; \Delta; \Gamma \vdash (\text{array } (n \dots) \mathfrak{a} \dots) : (A \tau_a (\text{shape } n \dots))} \text{T:ARRAY}$$

and

$$\frac{\Theta; \Delta; \Gamma \vdash \mathfrak{a} : \tau'_a \dots \quad \Theta; \Delta \vdash \tau'_a :: \text{Atom} \quad \text{Length} \llbracket \mathfrak{a} \dots \rrbracket = \prod n \dots}{\Theta; \Delta; \Gamma \vdash (\text{array } (n \dots) \mathfrak{a} \dots) : (A \tau'_a (\text{shape } n \dots))} \text{T:ARRAY}$$

By the induction hypothesis, the derivations for each  $\mathfrak{a}$  must produce equivalent types, *i.e.*,  $\tau_a \cong \tau'_a$ . Thus we can derive

$$\frac{\tau_a \cong \tau'_a \quad \models (\text{shape } n \dots) \equiv (\text{shape } n \dots)}{(A \tau_a (\text{shape } n \dots)) \cong (A \tau'_a (\text{shape } n \dots))} \text{TEQV:ARRAY}$$

FRAME:  $t = (\text{frame } (n \dots) e \dots)$ . Only T:FRAME is applicable, so the derivations are

$$\frac{\Theta; \Delta; \Gamma \vdash e : (A \tau_a \iota_c) \dots \quad \Theta; \Delta \vdash (A \tau_a \iota_c) :: \text{Array} \quad \text{Length} \llbracket \mathfrak{a} \dots \rrbracket = \prod n \dots}{\Theta; \Delta; \Gamma \vdash (\text{frame } (n \dots) e \dots) : (A \tau_a (++) (\text{shape } n \dots) \iota_c)} \text{T:FRAME}$$

and

$$\frac{\Theta; \Delta; \Gamma \vdash e : (A \tau'_a \iota'_c) \dots \quad \Theta; \Delta \vdash (A \tau'_a \iota'_c) :: \text{Array} \quad \text{Length} \llbracket \mathfrak{a} \dots \rrbracket = \prod n \dots}{\Theta; \Delta; \Gamma \vdash (\text{frame } (n \dots) e \dots) : (A \tau'_a (++) (\text{shape } n \dots) \iota'_c)} \text{T:FRAME}$$

By the induction hypothesis,  $(A \tau_a \iota_c) \cong (A \tau'_a \iota'_c)$ . The derivation for this must either end with TEQV:ARRAY or be simply TEQV:REFL. In either case, we have  $\tau_a \cong \tau'_a$  and  $\models \iota_c \equiv \iota'_c$ . The latter implies  $\models (++) (\text{shape } n \dots) \iota_c \equiv (++) (\text{shape } n \dots) \iota'_c$ , which leads to the derivation

$$\frac{\tau_a \cong \tau'_a \quad \models (\text{shape } n \dots) \iota_c \equiv (\text{shape } n \dots) \iota'_c}{(A \tau_a (++) (\text{shape } n \dots) \iota_c) \cong (A \tau'_a (++) (\text{shape } n \dots) \iota'_c)} \text{TEQV:ARRAY}$$

EMPTY ARRAY:  $t = (\text{array } (n \dots) \tau_a)$ . The only possible non-EQV rule for ending a type derivation is

$$\frac{\Theta; \Delta \vdash \tau_a :: \text{Atom} \quad 0 \in n \dots}{\Theta; \Delta; \Gamma \vdash (\text{array } (n \dots) \tau_a) : (A \tau_a (\text{shape } n \dots))} \text{T:EMPTYA}$$

Note that there are no atoms for derivations to conclude have distinct (but equivalent) types. Instead, the exact atom type is specified in the term itself. Therefore all types derived for  $t$  by TEQV:EMPTYA are equal, thus equivalent.

EMPTY FRAME:  $t = (\text{frame } (n \dots) (A \tau_a \iota))$ . Similar to the empty array case, only one non-EQV end to the derivation is possible:

$$\frac{\Theta; \Delta \vdash \tau_a :: \text{Atom} \quad \Theta \vdash \iota :: \text{Shape} \quad 0 \in n \dots}{\Theta; \Delta; \Gamma \vdash (\text{frame } (n \dots) (A \tau_a \iota)) : (A \tau_a (\text{shape } n \dots))} \text{T:EMPTYF}$$

Again, only one unique type is derivable.

APPLICATION:  $t = (e_f e_a \dots)$  The of a typing derivations for  $t$  end with

$$\frac{\begin{array}{c} \Theta; \Delta; \Gamma \vdash e_f : (A (-> ((A \tau_i \iota_i) \dots) (A \tau_o \iota_o)) \iota_f) \\ \Theta; \Delta; \Gamma \vdash e_a : (A \tau_i (++) \iota_a \iota_i) \dots \\ \iota_p = \text{Max} \llbracket \iota_f \iota_a \dots \rrbracket \end{array}}{\Theta; \Delta; \Gamma \vdash (e_f e_a \dots) : (A \tau_o (++) \iota_p \iota_o)}$$

and

$$\frac{\begin{array}{c} \Theta; \Delta; \Gamma \vdash e_f : (A (-> ((A \tau'_i \iota'_i) \dots) (A \tau'_o \iota'_o)) \iota'_f) \\ \Theta; \Delta; \Gamma \vdash e_a : (A \tau'_i (++) \iota'_a \iota'_i) \dots \\ \iota'_p = \text{Max} \llbracket \iota'_f \iota'_a \dots \rrbracket \end{array}}{\Theta; \Delta; \Gamma \vdash (e_f e_a \dots) : (A \tau'_o (++) \iota'_p \iota'_o)}$$

By the induction hypothesis, the types ascribed to  $e_f$  are equivalent. Since their equivalence can only be concluded using either TEQV:FN or TEQV:REFL, we know that the corresponding input types— $(A \tau_i \iota_i)$  and  $(A \tau'_i \iota'_i)$ —are equivalent. This in turn implies the equivalence of corresponding atom types  $\tau_i$  and  $\tau'_i$  and of corresponding argument cell shapes  $\iota_i$  and  $\iota'_i$  (the array types could only be equivalent because of TEQV:ARRAY or TEQV:REFL). For the same reason, we have equality of the function frames and of corresponding argument shapes. What we then need is equality of corresponding *argument frames*.

In the free monoid,  $a ++ b \equiv a ++ c$  implies  $b \equiv c$ , and  $b ++ a \equiv c ++ a$  implies  $b \equiv c$  (*n.b.*, this does not hold in all monoids). In effect, if we know that prefixes match on two equal shapes, we can conclude that their suffixes match as well. Applied to our situation, this means that



the equality of  $(++ \iota_a \iota_i)$  and  $(++ \iota'_a \iota'_i)$  with equal  $\iota_i$  and  $\iota'_i$  entails the equality of  $\iota_a$  and  $\iota'_a$ . So any two frames derived for the same argument *are* equal. Since all corresponding frames are equal, their respective maxima  $\iota_p$  and  $\iota'_p$  must also be equal.

Returning to the derived function types, their equivalence also implies the equivalence of their output types. Since  $teqv(A \tau_o \iota_o)(A \tau'_o \iota'_o)$ , it must be the case that  $\tau_o \cong \tau'_o$  and that  $\models \iota_o \equiv \iota'_o$ .

The equivalences  $\tau_o \cong \tau'_o$ ,  $\models \iota_o \equiv \iota'_o$ , and  $\models \iota_p \equiv \iota'_p$  lead to

$$\frac{\tau_o \cong \tau'_o \quad \models (++) \iota_p \iota_o \equiv (++) \iota'_p \iota'_o}{(A \tau_o (++) \iota_p \iota_o) \cong (A \tau'_o (++) \iota'_p \iota'_o)}$$

TYPE APPLICATION:  $t = (\text{t-app } e_f \tau_a \dots)$ . The derivations must have the form

$$\frac{\Theta; \Delta; \Gamma \vdash e_f : (A (\forall ((x k) \dots) (A \tau_u \iota_u))) \iota_f \quad \Theta; \Delta \vdash \tau_a :: k \dots}{\Theta; \Delta; \Gamma \vdash (\text{t-app } e_f \tau_a \dots) : (A \tau_u [x \mapsto \tau_a, \dots] (++) \iota_f \iota_u))} \text{T:TAPP}$$

and

$$\frac{\Theta; \Delta; \Gamma \vdash e_f : (A (\forall ((x' k) \dots) (A \tau'_u \iota'_u))) \iota'_f \quad \Theta; \Delta \vdash \tau_a :: k \dots}{\Theta; \Delta; \Gamma \vdash (\text{t-app } e_f \tau_a \dots) : (A \tau'_u [x' \mapsto \tau_a, \dots] (++) \iota'_f \iota'_u))} \text{T:TAPP}$$

From the induction hypothesis, we have

$$\begin{aligned} & (A (\forall ((x k) \dots) (A \tau_u \iota_u))) \iota_f \\ & \cong (A (\forall ((x' k) \dots) (A \tau'_u \iota'_u))) \iota'_f \end{aligned}$$

which then implies that  $\tau_u[x \mapsto x_f, \dots] \cong \tau'_u[x' \mapsto x_f, \dots]$ . Because substituting equal types into equivalent types produces equivalent types (by Lemma 4.2.12), we have

$$(\tau_u[x \mapsto x_f, \dots])[x_f \mapsto \tau_a, \dots] \cong (\tau'_u[x' \mapsto x_f, \dots])[x_f \mapsto \tau_a, \dots]$$

Applying both substitutions in order, we relate types  $\tau_u[x \mapsto \tau_a, \dots]$  and  $\tau'_u[x' \mapsto \tau_a, \dots]$ . Equivalence of types derived for  $e_f$  also implies equivalence of the result cell shapes  $\iota_u$  and  $\iota'_u$  and of the frame shapes  $\iota_f$  and  $\iota'_f$ . This means  $\models (++) \iota_f \iota_u \equiv (++) \iota'_f \iota'_u$ . So we can derive

$$\frac{\tau_u[x \mapsto \tau_a, \dots] \cong \tau'_u[x' \mapsto \tau_a, \dots] \quad \models (++) \iota_f \iota_u \equiv (++) \iota'_f \iota'_u}{(A \tau_u [x \mapsto \tau_a, \dots] (++) \iota_f \iota_u) \cong (A \tau'_u [x' \mapsto \tau_a, \dots] (++) \iota'_f \iota'_u)} \text{TEQV:ARRAY}$$

INDEX APPLICATION:  $t = (\text{i-app } e_f \iota_a \dots)$ . Non-EQV endings of the type derivations must have the form

$$\frac{\Theta; \Delta; \Gamma \vdash e : (A (\Pi ((x \gamma) \dots) (A \tau_p \iota_p))) \iota_f \quad \Theta \vdash \iota_a :: \gamma \dots}{\Theta; \Delta; \Gamma \vdash (\text{i-app } e_f \iota_a \dots) : (A \tau_p [x \mapsto \iota, \dots] (++) \iota_f \iota_p [x \mapsto \iota, \dots]))} \text{T:IAPP}$$

Suppose that in two typing derivations,  $e_f$  is ascribed types

$$(A (\Pi ((x \gamma) \dots) (A \tau_p \iota_p))) \iota_f$$

and

$$(A (\Pi ((x' \gamma') \dots) (A \tau'_p \iota'_p))) \iota'_f$$

The induction hypothesis implies that these two types are equivalent. Deriving this equivalence must use TEQV:ARRAY, relating  $\iota_f$  with  $\iota'_f$  and  $(A \tau_p \iota_p)$  with  $(A \tau'_p \iota'_p)$ . By similar substitution-tracing arguments as in the type application case, we have

$$\tau_p [x \mapsto \iota_a, \dots] \cong \tau'_p [x' \mapsto \iota_a, \dots]$$

and

$$\text{VALID}[\iota_p [x \mapsto \iota, \dots] \equiv \iota'_p [x \mapsto \iota, \dots]]$$

leading to

$$\frac{\tau_p [x \mapsto \iota_a, \dots] \cong \tau'_p [x' \mapsto \iota_a, \dots] \quad \models (++) \iota_f \iota_p [x \mapsto \iota, \dots] \equiv (++) \iota'_f \iota'_p [x \mapsto \iota, \dots]}{(A \tau_p [x \mapsto \iota, \dots] (++) \iota_f \iota_p [x \mapsto \iota, \dots]) \cong (A \tau'_p [x' \mapsto \iota, \dots] (++) \iota'_f \iota'_p [x \mapsto \iota, \dots])} \text{TEQV:PI}$$

UNBOXING:  $t = (\text{unbox } (x_i \dots x_e e_s) e_b)$ . The only non-T:EQV rule which can type  $t$  is T:UNBOX, so type derivations must end with

$$\frac{\Theta; \Delta; \Gamma \vdash e_s : (A (\Sigma ((x'_i \gamma') \dots) \tau_s) \iota_s) \quad \Theta, x_i :: \gamma \dots; \Delta; \Gamma, x_e : \tau_s [x'_i \mapsto x_i, \dots] \vdash e_b : (A \tau_b \iota_b) \quad \Theta; \Delta \vdash (A \tau_b \iota_b) :: \text{Array}}{\Theta; \Delta; \Gamma \vdash (\text{unbox } (x_i \dots x_e e_s) e_b) : (A \tau_b (++) \iota_s \iota_b)}$$

According to the induction hypothesis, any two types  $(A \tau_b (++) \iota_s \iota_b)$  and  $(A \tau'_b (++) \iota'_s \iota'_b)$  ascribed to  $e_b$  using the extended environment by the second premise must be equivalent—*i.e.*,  $(A \tau_b (++) \iota_s \iota_b) \cong (A \tau'_b (++) \iota'_s \iota'_b)$ . This is exactly the proof obligation.

ABSTRACTION:  $t = (\lambda ((x \tau_i) \dots) e)$ . Type derivations must end with T:LAM:

$$\frac{\Theta; \Delta; \Gamma, x : \tau_i \dots \vdash e : \tau_o \quad \Theta; \Delta \vdash \tau_i :: \text{Array} \dots}{\Theta; \Delta; \Gamma \vdash (\lambda ((x \tau_i) \dots) e) : (-> (\tau_i \dots) \tau_o)} \text{ T:LAM}$$

For any two such derivations, the induction hypothesis implies that types  $\tau_o$  and  $\tau'_o$  are equivalent, since they are both ascribed to  $e$ . Since the input types  $\tau_i \dots$  are given explicitly in the term, we can derive

$$\frac{\frac{\dots}{\tau_i \cong \tau_i} \text{ TEQV:REFL} \quad \dots \quad \tau_o \cong \tau'_o}{(-> (\tau_i \dots) \tau_o) \cong (-> (\tau_i \dots) \tau'_o)}$$

TYPE ABSTRACTION:  $t = (\top \lambda ((x k) \dots) e)$ . Non-EQV derivations must end with

$$\frac{\Theta; \Delta, x :: k \dots; \Gamma \vdash e : \tau_u}{\Theta; \Delta; \Gamma \vdash (\top \lambda ((x k) \dots) e) : (\forall ((x k) \dots) \tau_u)} \text{ T:TLAM}$$

Any  $\tau_u$  and  $\tau'_u$  ascribed to  $e$ , the body of the abstraction, must be equivalent, and substituting in the same fresh variables will preserve that equivalence, by Lemma 4.2.12. So we can construct the derivation

$$\frac{\tau_u[x \mapsto x_f, \dots] \cong \tau'_u[x \mapsto x_f, \dots]}{(\forall ((x k) \dots) \tau_u) \cong (\forall ((x k) \dots) \tau'_u)} \text{ TEQV:UNIV}$$

INDEX ABSTRACTION:  $t = (\text{I} \lambda ((x \gamma) \dots) e)$ .

$$\frac{\Theta, x :: \gamma \dots; \Delta; \Gamma \vdash e : \tau_p}{\Theta; \Delta; \Gamma \vdash (\text{I} \lambda ((x k) \dots) e) : (\forall ((x k) \dots) \tau_p)} \text{ T:ILAM}$$

Similar to the type abstraction case, any types ascribed to the abstraction body must be equivalent, and  $\alpha$ -conversion will preserve that equivalence, leading to

$$\frac{\tau_p[x \mapsto x_f, \dots] \cong \tau'_p[x \mapsto x_f, \dots]}{(\forall ((x k) \dots) \tau_p) \cong (\forall ((x k) \dots) \tau'_p)} \text{ TEQV:PI}$$

BOX CONSTRUCTION:  $t = (\text{box } \iota \dots e \tau_s)$ . Only  $\tau_s$  can be ascribed to  $t$  without ending the derivation with T:EQV.

BASE VALUE OR PRIMITIVE OPERATOR:  $t = \mathbf{b}$  or  $t = \mathbf{o}$ . Each base value and primitive operator has a single defined type.  $\square$

**Lemma 4.2.13** (Preservation of types under index substitution). *Given  $\Theta, x :: \gamma; \Delta; \Gamma \vdash t : \tau$  and  $\Theta \vdash \iota_x :: \gamma$  then  $\Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash t[x \mapsto \iota_x] : \tau[x \mapsto \iota_x]$ .*

*Proof.* We use induction on the derivation of  $\Theta, x :: \gamma; \Delta; \Gamma \vdash t : \tau$ .

**Case EQV**

$$\frac{\Theta, x :: \gamma; \Delta; \Gamma \vdash t : \tau' \quad \tau \cong \tau'}{\Theta, x :: \gamma; \Delta; \Gamma \vdash t : \tau} \text{ T:EQV}$$

By the induction hypothesis,  $\Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash t[x \mapsto \iota_x] : \tau'[x \mapsto \iota_x]$ , and preservation of equivalence (Lemma 4.2.11) implies  $\tau[x \mapsto \iota_x] \cong \tau'[x \mapsto \iota_x]$ . Thus we can build the derivation

$$\frac{\Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash t[x \mapsto \iota_x] : \tau'[x \mapsto \iota_x] \quad \tau[x \mapsto \iota_x] \cong \tau'[x \mapsto \iota_x]}{\Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash t[x \mapsto \iota_x] : \tau[x \mapsto \iota_x]} \text{ T:EQV}$$

**Case OP, BASE**

These cases are trivial, since primitive operators and base values are only ascribed closed types.

**Case VAR**

$$\frac{(x' : \tau) \in \Gamma}{\Theta, x :: \gamma; \Delta; \Gamma \vdash x' : \tau} \text{ T:VAR}$$

In this case,  $x$  cannot appear in  $t$ , so  $x'[x \mapsto \iota_x] = x'$ . Since  $x'$  is bound in  $\Gamma$ , we can construct a similar derivation with  $\Gamma(x')$  updated by substitution:

$$\frac{(x' : \tau[x \mapsto \iota_x]) \in \Gamma[x \mapsto \iota_x]}{\Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash x' : \tau[x \mapsto \iota_x]} \text{ T:VAR}$$

**Case ARRAY**

$$\frac{\Theta, x :: \gamma; \Delta; \Gamma \vdash \mathfrak{a} : \tau_a \dots \quad \Theta, x :: \gamma; \Delta \vdash \tau_a :: \text{Atom} \quad \text{Length}[\llbracket \mathfrak{a} \dots \rrbracket] = \prod n \dots}{\Theta, x :: \gamma; \Delta; \Gamma \vdash (\text{array } (n \dots) \mathfrak{a} \dots) : (\text{A } \tau_a \text{ (shape } n \dots \text{)})} \text{ T:ARRAY}$$

By the induction hypothesis,  $\Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash \mathfrak{a}[x \mapsto \iota_x] : \tau_a[x \mapsto \iota_x]$  for each of  $\mathfrak{a} \dots$ . Preservation of kinds under substitution (Lemma 4.2.4) implies  $\Theta; \Delta \vdash \tau_a[x \mapsto \iota_x] :: \text{Atom}$ . We use these results to construct the derivation

$$\frac{\Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash \mathfrak{a}[x \mapsto \iota_x] : \tau_a[x \mapsto \iota_x] \dots \quad \Theta; \Delta \vdash \tau_a[x \mapsto \iota_x] :: \text{Atom} \quad \text{Length}[\llbracket \mathfrak{a} \dots \rrbracket] = \prod n \dots}{\Theta; \Delta; \Gamma \vdash (\text{array } (n \dots) \mathfrak{a}[x \mapsto \iota_x] \dots) : (\text{A } \tau_a[x \mapsto \iota_x] \text{ (shape } n \dots \text{)})} \text{ T:ARRAY}$$

**Case FRAME**

$$\begin{array}{c}
 \Theta, x :: \gamma; \Delta; \Gamma \vdash e_a : (A \tau_a \iota_a) \dots \\
 \Theta, x :: \gamma; \Delta \vdash (A \tau_a \iota_a) :: \text{Array} \\
 \text{Length} \llbracket e_a \dots \rrbracket = \prod n \dots \\
 \hline
 \Theta, x :: \gamma; \Delta; \Gamma \vdash \\
 (\text{frame } (n \dots) e \dots) \\
 : (A \tau_a (++) (\text{shape } n \dots) \iota_a)
 \end{array} \quad \text{T:FRAME}$$

The induction hypothesis implies for each of the  $e \dots$  that

$$\Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash e[x \mapsto \iota_x] : (A \tau_a[x \mapsto \iota_x] \iota_a[x \mapsto \iota_x])$$

As in the T:ARRAY case, we use preservation of kinds to determine that  $\Theta; \Delta \vdash (A \tau_a[x \mapsto \iota_x] \iota_a[x \mapsto \iota_x]) :: \text{Array}$ . Then we derive

$$\begin{array}{c}
 \Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash e_a[x \mapsto \iota_x] \\
 : (A \tau_a[x \mapsto \iota_x] \iota_a[x \mapsto \iota_x]) \dots \\
 \Theta; \Delta \vdash (A \tau_a[x \mapsto \iota_x] \iota_a[x \mapsto \iota_x]) :: \text{Array} \\
 \text{Length} \llbracket e_a \dots \rrbracket = \prod n \dots \\
 \hline
 \Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash (\text{frame } (n \dots) e[x \mapsto \iota_x] \dots) \\
 : (A \tau_a[x \mapsto \iota_x] (++) (\text{shape } n \dots) \iota_a[x \mapsto \iota_x])
 \end{array} \quad \text{T:FRAME}$$

**Case EMPTYA**

$$\begin{array}{c}
 \Theta, x :: \gamma; \Delta \vdash \tau_a :: \text{Atom} \quad 0 \in n \dots \\
 \hline
 \Theta, x :: \gamma; \Delta; \Gamma \vdash \\
 (\text{array } (n \dots) \tau_a) \\
 : (A \tau_a (\text{shape } n \dots))
 \end{array} \quad \text{T:EMPTYA}$$

By preservation of kinds,  $\Theta; \Delta \vdash \tau_a[x \mapsto \iota_x] :: \text{Atom}$ . This leads to the derivation

$$\begin{array}{c}
 \Theta; \Delta \vdash \tau_a[x \mapsto \iota_x] :: \text{Atom} \quad 0 \in n \dots \\
 \hline
 \Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash \\
 (\text{array } (n \dots) \tau_a[x \mapsto \iota_x]) \\
 : (A \tau_a[x \mapsto \iota_x] (\text{shape } n \dots))
 \end{array} \quad \text{T:EMPTYA}$$

**Case EMPTYF**

$$\begin{array}{c}
 \Theta, x :: \gamma; \Delta \vdash \tau_a :: \text{Atom} \\
 \Theta, x :: \gamma \vdash \iota_a :: \text{Shape} \quad 0 \in n \dots \\
 \hline
 \Theta, x :: \gamma; \Delta; \Gamma \vdash \\
 (\text{frame } (n \dots) (A \tau_a \iota_a)) \\
 : (A \tau_a (++) (\text{shape } n \dots) \iota_a)
 \end{array} \quad \text{T:EMPTYF}$$

As before, preservation of kinds implies  $\Theta; \Delta \vdash \tau_a[x \mapsto \iota_x] :: \text{Atom}$ . We also have, by preservation of sorts under substitution (Lemma 4.2.2)  $\Theta \vdash \iota_a[x \mapsto \iota_x] :: \text{Shape}$ . So we can derive

$$\frac{\Theta; \Delta \vdash \tau_a[x \mapsto \iota_x] :: \text{Atom} \quad \Theta \vdash \iota_a[x \mapsto \iota_x] :: \text{Shape} \quad 0 \in n \dots}{\Theta; \Delta; \Gamma \vdash \text{(frame } (n \dots) \text{ (A } \tau_a[x \mapsto \iota_x] \iota_a[x \mapsto \iota_x] \text{)) : (A } \tau_a[x \mapsto \iota_x] \text{ (} ++ \text{ (shape } n \dots \text{) } \iota_a[x \mapsto \iota_x] \text{))}} \text{T:EMPTYF}$$

### Case LAM

$$\frac{\Theta, x :: \gamma; \Delta; \Gamma, x_i : \tau_i \dots \vdash e : \tau_o \quad \Theta, x :: \gamma; \Delta \vdash \tau_i :: \text{Array}}{\Theta, x :: \gamma; \Delta; \Gamma \vdash (\lambda ((x_i \tau_i) \dots) e) : (\rightarrow (\tau_i \dots) \tau_o)} \text{T:LAM}$$

The induction hypothesis gives us  $\Theta; \Delta; \Gamma[x \mapsto \iota_x], x_i : \tau_i[x \mapsto \iota_x] \dots \vdash e[x \mapsto \iota_x] : \tau_o[x \mapsto \iota_x]$ . By preservation of kinds,  $\Theta; \Delta \vdash \Gamma[x \mapsto \iota_x] :: \text{Array}$  for each of  $\tau_i \dots$ , which leads to the derivation

$$\frac{\Theta; \Delta; \Gamma[x \mapsto \iota_x], x_i : \tau_i[x \mapsto \iota_x] \dots \vdash e[x \mapsto \iota_x] : \tau_o[x \mapsto \iota_x] \quad \Theta; \Delta \vdash \tau_i[x \mapsto \iota_x] :: \text{Array} \dots}{\Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash (\lambda (x_i \dots \tau_i[x \mapsto \iota_x]) e[x \mapsto \iota_x]) : (\rightarrow (\tau_i[x \mapsto \iota_x] \dots) \tau_o[x \mapsto \iota_x])} \text{T:LAM}$$

### Case TLAM

$$\frac{\Theta, x :: \gamma; \Delta, x_u :: k \dots; \Gamma \vdash e : \tau_u}{\Theta, x :: \gamma; \Delta; \Gamma \vdash (\text{T}\lambda ((x_u k) \dots) e) : (\forall ((x_u k) \dots) \tau_u)} \text{T:TLAM}$$

By the induction hypothesis,  $\Theta; \Delta, x_u :: k \dots; \Gamma[x \mapsto \iota_x] \vdash e[x \mapsto \iota_x] : \tau_u[x \mapsto \iota_x]$ , so we derive

$$\frac{\Theta; \Delta, x_u :: k \dots; \Gamma[x \mapsto \iota_x] \vdash e[x \mapsto \iota_x] : \tau_u[x \mapsto \iota_x]}{\Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash (\text{T}\lambda ((x_u k) \dots) e[x \mapsto \iota_x]) : (\forall ((x_u k) \dots) \tau_u[x \mapsto \iota_x])} \text{T:TLAM}$$

### Case ILAM

$$\frac{\Theta, x :: \gamma, x_u :: \gamma_p \dots; \Delta; \Gamma \vdash e : \tau_p}{\Theta, x :: \gamma; \Delta; \Gamma \vdash (\text{I}\lambda ((x_p \gamma_p) \dots) e) : (\prod ((x_p \gamma_p) \dots) \tau_p)} \text{T:ILAM}$$

Following Barendregt's convention, we assume that  $x$  does not appear in  $\iota_x$ . The induction hypothesis gives  $\Theta, x_p :: \gamma_p \dots; \Delta; \Gamma[x \mapsto \iota_x] \vdash e[x \mapsto \iota_x] : \tau_p[x \mapsto \iota_x]$ , leading to

$$\frac{\Theta, x_u :: \gamma_u \dots; \Delta; \Gamma[x \mapsto \iota_x] \vdash e[x \mapsto \iota_x] : \tau_u[x \mapsto \iota_x]}{\Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash (\text{T}\lambda ((x_u k) \dots) e[x \mapsto \iota_x]) : (\forall ((x_u k) \dots) \tau_u[x \mapsto \iota_x])} \text{T:ILAM}$$

**Case BOX**

$$\frac{\begin{array}{c} \Theta, x :: \gamma \vdash \iota_s :: \gamma_s \dots \\ \Theta, x :: \gamma; \Delta \vdash (\Sigma ((x_s \gamma_s) \dots) \tau_s) :: \text{Atom} \\ \Theta, x :: \gamma; \Delta; \Gamma \vdash e : \tau_s[x_s \mapsto \iota_s, \dots] \end{array}}{\Theta, x :: \gamma; \Delta; \Gamma \vdash (\text{box } \iota_s \dots e (\Sigma ((x_s \gamma_s) \dots) \tau_s)) : (\Sigma ((x_s \gamma_s) \dots) \tau_s)} \text{T:BOX}$$

By the induction hypothesis,

$$\Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash e[x \mapsto \iota_x] : \tau_s[x_s \mapsto \iota_s, \dots][x \mapsto \iota_x]$$

This ascribed type is equal to  $\tau_s[x \mapsto \iota_x][x_s \mapsto \iota_s, \dots]$ . Preservation of sorts implies  $\Theta \vdash \iota_s[x \mapsto \iota_x] :: \gamma_s$  for each of  $\iota_s \dots$ , while preservation of kinds implies  $\Theta; \Delta \vdash (\Sigma ((x_s \gamma_s) \dots) \tau_s[x \mapsto \iota_x]) :: \text{Atom}$ . Applying T:BOX derives

$$\frac{\begin{array}{c} \Theta \vdash \iota_s[x \mapsto \iota_x] :: \gamma_s \dots \\ \Theta; \Delta \vdash (\Sigma ((x_s \gamma_s) \dots) \tau_s[x \mapsto \iota_x]) :: \text{Atom} \\ \Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash e[x \mapsto \iota_x] : \tau_s[x \mapsto \iota_x][x_s \mapsto \iota_s, \dots] \end{array}}{\Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash (\text{box } \iota_s[x \mapsto \iota_x] \dots e[x \mapsto \iota_x] (\Sigma ((x_s \gamma_s) \dots) \tau_s[x \mapsto \iota_x])) : (\Sigma ((x_s \gamma_s) \dots) \tau_s[x \mapsto \iota_x])} \text{T:BOX}$$

**Case TAPP**

$$\frac{\begin{array}{c} \Theta, x :: \gamma; \Delta; \Gamma \vdash e : (A (\forall ((x_u k) \dots) (A \tau_u \iota_u)) \iota_f) \\ \Theta, x :: \gamma; \Delta \vdash \tau_a :: k \dots \end{array}}{\Theta, x :: \gamma; \Delta; \Gamma \vdash (\text{t-app } e \tau_a \dots) : (A \tau_u[x_u \mapsto \tau_a, \dots] (\iota_f \iota_u))} \text{T:TAPP}$$

The induction hypothesis gives

$$\begin{array}{c} \Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash e[x \mapsto \iota_x] : \\ (A (\forall ((x_u k) \dots) \\ (A \tau_u[x \mapsto \iota_x] \iota_u[x \mapsto \iota_x])) (\iota_f[x \mapsto \iota_x])) \end{array}$$

Preservation of kinds under index substitution (Lemma 4.2.4) implies  $\Theta; \Delta \vdash \tau_a[x \mapsto \iota_x] :: k$  for each of  $\tau_a \dots$ . So we derive

$$\frac{\begin{array}{c} \Theta; \Delta \vdash \tau_a[x \mapsto \iota_x] :: k \dots \\ \Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash e[x \mapsto \iota_x] : \\ (A (\forall ((x_u k) \dots) \\ (A \tau_u[x \mapsto \iota_x] \iota_u[x \mapsto \iota_x]) \\ \iota_f[x \mapsto \iota_x])) \end{array}}{\Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash (\text{t-app } e \tau_a[x \mapsto \iota_x] \dots) : (A \tau_u[x \mapsto \iota_x][x_u \mapsto \tau_a, \dots] (\iota_f[x \mapsto \iota_x] \iota_u[x \mapsto \iota_x]))} \text{T:TAPP}$$

**Case IAPP**

$$\frac{\Theta, x :: \gamma; \Delta; \Gamma \vdash e : (A (\Pi ((x_p \gamma_p) \dots)) (A \tau_p \iota_p)) \iota_f \quad \Theta, x :: \gamma \vdash \iota_a :: \gamma_p \dots}{\Theta, x :: \gamma; \Delta; \Gamma \vdash (\text{i-app } e \iota_a \dots)} \text{T:IAPP}$$

The induction hypothesis implies

$$\begin{aligned} & \Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash e[x \mapsto \iota_x] : \\ & (A (\Pi ((x_p \gamma_p) \dots)) \\ & \quad (A \tau_p[x \mapsto \iota_x] \iota_p[x \mapsto \iota_x])) \\ & \quad \iota_f[x \mapsto \iota_x]) \end{aligned}$$

Preservation of sorts under index substitution (Lemma 4.2.2) implies  $\Theta \vdash \iota_a[x \mapsto \iota_x] :: \gamma_a$  for each corresponding pair  $(\iota_a, \gamma_a) \dots$ . So we construct the derivation

$$\frac{\begin{aligned} & \Theta \vdash \iota_a[x \mapsto \iota_x] :: \gamma_p \dots \\ & \Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash e[x \mapsto \iota_x] \\ & : (A (\forall ((x_u k) \dots)) \\ & \quad (A \tau_p[x \mapsto \iota_x] \iota_p[x \mapsto \iota_x])) \\ & \quad (\iota_f[x \mapsto \iota_x])) \end{aligned}}{\Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash (\text{i-app } e \iota_a[x \mapsto \iota_x] \dots) : (A \tau_p[x \mapsto \iota_x][x_p \mapsto \iota_a, \dots] (++) \iota_f[x \mapsto \iota_x] \iota_p[x \mapsto \iota_x][x_p \mapsto \iota_a, \dots]))} \text{T:IAPP}$$

**Case APP**

$$\frac{\begin{aligned} & \Theta, x :: \gamma; \Delta; \Gamma \vdash e_f : (A (-> ((A \tau_i \iota_i) \dots) (A \tau_o \iota_o)) \iota_f) \\ & \quad \Theta, x :: \gamma; \Delta; \Gamma \vdash e_a : (A \tau_i (++) \iota_a \iota_i) \dots \\ & \quad \iota_p = \text{Max} [\iota_f \iota_a \dots] \end{aligned}}{\Theta, x :: \gamma; \Delta; \Gamma \vdash (e_f e_a \dots) : (A \tau_o (++) \iota_p \iota_o)} \text{T:APP}$$

By the induction hypothesis,

$$\begin{aligned} & \Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash e_f[x \mapsto \iota_x] : \\ & (A (-> ((A \tau_i[x \mapsto \iota_x] \iota_i[x \mapsto \iota_x]) \dots) \\ & \quad (A \tau_o[x \mapsto \iota_x] \iota_o[x \mapsto \iota_x]))) \\ & \quad \iota_f[x \mapsto \iota_x]) \end{aligned}$$

and for each of  $e_a \dots$ , we have

$$\begin{aligned} & \Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash e_a[x \mapsto \iota_x] : \\ & (A \tau_i[x \mapsto \iota_x] (++) \iota_a[x \mapsto \iota_x] \iota_i[x \mapsto \iota_x])) \end{aligned}$$



Since substituting into a shape is monotonic in the subbed-in index, *i.e.*, prefix ordering commutes with substitution,

$$\begin{aligned} & \text{Max} \llbracket \iota_f[x \mapsto \iota_x] \iota_a[x \mapsto \iota_x] \dots \rrbracket \\ &= \text{Max} \llbracket \iota_f \iota_a \dots \rrbracket [x \mapsto \iota_x] \\ &= \iota_p[x \mapsto \iota_x] \end{aligned}$$

Then we build the derivation

$$\begin{array}{c} \Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash e_f[x \mapsto \iota_x] \\ : (A \ (-> \ ((A \ \tau_i[x \mapsto \iota_x] \ \iota_i[x \mapsto \iota_x]) \\ \dots) \\ (A \ \tau_o[x \mapsto \iota_x] \ \iota_o[x \mapsto \iota_x]))) \\ \iota_f) \\ \\ \Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash e_a[x \mapsto \iota_x] \\ : (A \ \tau_i[x \mapsto \iota_x] \ (++) \ \iota_a[x \mapsto \iota_x] \ \iota_i[x \mapsto \iota_x])) \quad \dots \\ \\ \frac{\iota_p[x \mapsto \iota_x] = \text{Max} \llbracket \iota_f[x \mapsto \iota_x] \ \iota_a[x \mapsto \iota_x] \dots \rrbracket}{\Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash (e_f[x \mapsto \iota_x] \ e_a[x \mapsto \iota_x] \dots)} \quad \text{T:APP} \\ : (A \ \tau_o[x \mapsto \iota_x] \ (++) \ \iota_p[x \mapsto \iota_x] \ \iota_o[x \mapsto \iota_x])) \end{array}$$

### Case UNBOX

$$\begin{array}{c} \Theta, x :: \gamma; \Delta; \Gamma \vdash e_s : (A \ (\Sigma \ (x'_i :: \gamma_i \dots) \ \tau_s) \ \iota_s) \\ \Theta, x :: \gamma, x_i :: \gamma_i \dots \Delta \Gamma, x_e : \tau_s[x'_i \mapsto x_i, \dots] \\ \vdash e_b : (A \ \tau_b \ \iota_b) \\ \\ \Theta, x :: \gamma; \Delta \vdash (A \ \tau_b \ \iota_b) :: \text{Array} \\ \hline \Theta, x :: \gamma; \Delta; \Gamma \vdash (\text{unbox} \ (x_i \dots \ x_e \ e_s) \ e_b) \\ : (A \ \tau_b \ (++) \ \iota_s \ \iota_b)) \quad \text{T:UNBOX} \end{array}$$

Per Barendregt's convention, we stipulate that  $x \notin x'_i \dots$ . Then the induction hypothesis gives both

$$\begin{aligned} & \Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash e_s[x \mapsto \iota_x] : \\ & (A \ (\Sigma \ (x'_i \dots \ \gamma_i) \ \tau_s[x \mapsto \iota_x]) \ \iota_s[x \mapsto \iota_x]) \end{aligned}$$

and

$$\begin{aligned} & \Theta, x_i :: \gamma_i \dots \Delta \Gamma[x \mapsto \iota_x], x_e : (\tau_s[x'_i \mapsto x_i, \dots])[x \mapsto \iota_x] \\ & \vdash e_b[x \mapsto \iota_x] : \tau_b[x \mapsto \iota_x] \end{aligned}$$

By preservation of kinds,  $\Theta; \Delta \vdash (A \tau_b \iota_b)[x \mapsto \iota_x] :: \text{Array}$ . So we can build the derivation

$$\begin{array}{c}
\Theta, \Delta, \Gamma[x \mapsto \iota_x] \vdash e_s[x \mapsto \iota_x] \\
: (A (\Sigma (x'_i :: \gamma_i \dots) \tau_s[x \mapsto \iota_x]) \iota_s) \\
\\
\Theta, x_i :: \gamma_i \dots; \Delta; \\
\Gamma[x \mapsto \iota_x], x_e : (\tau_s[x'_i \mapsto x_i, \dots])[x \mapsto \iota_x] \\
\vdash e_b[x \mapsto \iota_x] : (A \tau_b \iota_b)[x \mapsto \iota_x] \\
\\
\frac{\Theta; \Delta \vdash (A \tau_b \iota_b)[x \mapsto \iota_x] :: \text{Array}}{\Theta; \Delta; \Gamma[x \mapsto \iota_x] \vdash} \text{T:UNBOX} \\
(\text{unbox } (x_i \dots x_e e_s[x \mapsto \iota_x]) e_b[x \mapsto \iota_x]) \\
: (A \tau_b \iota_b)[x \mapsto \iota_x]
\end{array}$$

□

**Lemma 4.2.14** (Preservation of types under type substitution).

Given  $\Theta; \Delta, x :: k; \Gamma \vdash t : \tau$  and  $\Theta; \Delta \vdash \tau_x :: k$ ,  
then  $\Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash t[x \mapsto \tau_x] : \tau[x \mapsto \tau_x]$ .

*Proof.* We use induction on the derivation of  $\Theta; \Delta, x :: k; \Gamma \vdash t : \tau$ .

**Case EQV**

$$\frac{\Theta; \Delta, x :: k; \Gamma \vdash t : \tau' \quad \tau \cong \tau'}{\Theta; \Delta, x :: k; \Gamma \vdash t : \tau} \text{T:EQV}$$

The induction hypothesis implies

$$\Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash t[x \mapsto \tau_x] : \tau'[x \mapsto \tau_x]$$

Preservation of equivalence (Lemma 4.2.12) implies

$$\tau[x \mapsto \tau_x] \cong \tau'[x \mapsto \tau_x]$$

So we derive

$$\frac{\Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash t[x \mapsto \tau_x] : \tau'[x \mapsto \tau_x] \quad \tau[x \mapsto \tau_x] \cong \tau'[x \mapsto \tau_x]}{\Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash t[x \mapsto \tau_x] : \tau[x \mapsto \tau_x]} \text{T:EQV}$$

**Case OP, BASE**

Since  $x$  cannot appear free in  $t$  or  $\tau$ , the goal, after substitution, is to ascribe  $\tau$  to  $t$  in the diminished environment, which can be done because T:OP and T:BASE do not depend on the contents of the environment.

**Case VAR**

$$\frac{(x' : \tau) \in \Gamma}{\Theta; \Delta, x :: k; \Gamma \vdash x' : \tau} \text{T:VAR}$$

The type variable  $x$  cannot appear free in  $x'$  (which is a term variable), so  $x'[x \mapsto \tau_x] = x'$ . Applying substitution to  $\Gamma$  maps  $\Gamma(x')$  into  $\Gamma(x')[x \mapsto \tau_x]$ , so  $(x' : \tau[x \mapsto \tau_x]) \in \Gamma[x \mapsto \tau_x]$ . We then derive

$$\frac{(x' : \tau[x \mapsto \tau_x]) \in \Gamma[x \mapsto \tau_x]}{\Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash x' : \tau[x \mapsto \tau_x]} \text{ T:VAR}$$

### Case ARRAY

$$\frac{\begin{array}{l} \Theta; \Delta, x :: k; \Gamma \vdash a : \tau_a \dots \\ \Theta; \Delta, x :: k \vdash \tau_a :: \text{Atom} \quad \text{Length} \llbracket a \dots \rrbracket = \prod n \dots \end{array}}{\Theta; \Delta, x :: k; \Gamma \vdash (\text{array } (n \dots) a \dots) : (A \tau_a (\text{shape } n \dots))} \text{ T:ARRAY}$$

The induction hypothesis implies  $\Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash a : \tau_a[x \mapsto \tau_x]$  for each of the  $a \dots$ . By preservation of kinds (Lemma 4.2.5),  $\Theta; \Delta \vdash \tau_a[x \mapsto \tau_x] :: \text{Atom}$ . This leads to the derivation

$$\frac{\begin{array}{l} \Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash a[x \mapsto \tau_x] : \tau_a[x \mapsto \tau_x] \dots \\ \Theta; \Delta \vdash \tau_a[x \mapsto \tau_x] :: \text{Atom} \quad \text{Length} \llbracket a \dots \rrbracket = \prod n \dots \end{array}}{\begin{array}{l} \Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash \\ (\text{array } (n \dots) a[x \mapsto \tau_x] \dots) \\ : (A \tau_a[x \mapsto \tau_x] (\text{shape } n \dots)) \end{array}} \text{ T:ARRAY}$$

### Case FRAME

$$\frac{\begin{array}{l} \Theta; \Delta, x :: k; \Gamma \vdash e : (A \tau_a \iota_a) \dots \\ \Theta; \Delta, x :: k \vdash (A \tau_a \iota_a) :: \text{Array} \\ \text{Length} \llbracket a \dots \rrbracket = \prod n \dots \end{array}}{\Theta; \Delta, x :: k; \Gamma \vdash (\text{frame } (n \dots) e \dots) : (A \tau_a (++) (\text{shape } n \dots) \iota_a)} \text{ T:FRAME}$$

By the induction hypothesis, for each of the  $e \dots$  we have

$$\Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash e[x \mapsto \tau_x] : (A \tau_a[x \mapsto \tau_x] \iota_a)$$

Preservation of kinds gives

$$\Theta; \Delta \vdash (A \tau_a[x \mapsto \tau_x] \iota_a) :: \text{Array}$$

So we derive

$$\frac{\begin{array}{l} \Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash e[x \mapsto \tau_x] : (A \tau_a[x \mapsto \tau_x] \iota_a) \dots \\ \Theta; \Delta \vdash (A \tau_a[x \mapsto \tau_x] \iota_a) :: \text{Array} \\ \text{Length} \llbracket a \dots \rrbracket = \prod n \dots \end{array}}{\begin{array}{l} \Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash \\ (\text{frame } (n \dots) e[x \mapsto \tau_x] \dots) \\ : (A \tau_a[x \mapsto \tau_x] (++) (\text{shape } n \dots) \iota_a) \end{array}} \text{ T:FRAME}$$

**Case EMPTYA**

$$\frac{\Theta; \Delta, x :: k \vdash \tau_a :: \text{Atom} \quad 0 \in n \dots}{\Theta; \Delta, x :: k; \Gamma \vdash (\text{array } (n \dots) \tau_a) : (A \tau_a (\text{shape } n \dots))} \text{T:EMPTYA}$$

Preservation of kinds implies  $\Theta; \Delta \vdash \tau_a[x \mapsto \tau_x] :: \text{Atom}$ , so we can derive

$$\frac{\Theta; \Delta \vdash \tau_a[x \mapsto \tau_x] :: \text{Atom} \quad 0 \in n \dots}{\Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash (\text{array } (n \dots) \tau_a[x \mapsto \tau_x]) : (A \tau_a[x \mapsto \tau_x] (\text{shape } n \dots))} \text{T:EMPTYA}$$

**Case EMPTYF**

$$\frac{\Theta; \Delta, x :: k \vdash (A \tau_a \iota_a) :: \text{Array} \quad 0 \in n \dots}{\Theta; \Delta, x :: k; \Gamma \vdash (\text{frame } (n \dots) (A \tau_a \iota_a)) : (A \tau_a (++) \iota_a (\text{shape } n \dots))} \text{T:EMPTYF}$$

Preservation of kinds implies  $\Theta; \Delta \vdash (A \tau_a[x \mapsto \tau_x] \iota_a) :: \text{Array}$ . so we can derive

$$\frac{\Theta; \Delta \vdash (A \tau_a[x \mapsto \tau_x] \iota_a) :: \text{Array} \quad 0 \in n \dots}{\Theta; \Delta, x :: k; \Gamma \vdash (\text{frame } (n \dots) (A \tau_a[x \mapsto \tau_x] \iota_a)) : (A \tau_a[x \mapsto \tau_x] (++) \iota_a (\text{shape } n \dots))} \text{T:EMPTYF}$$

**Case LAM**

$$\frac{\Theta; \Delta, x :: k; \Gamma, x_i : \tau_i \dots \vdash e : \tau_o}{\Theta; \Delta, x :: k; \Gamma \vdash (\lambda ((x_i \tau_i) \dots) e) : (-> (\tau_i \dots) \tau_o)} \text{T:LAM}$$

The induction hypothesis implies  $\Theta; \Delta; \Gamma[x \mapsto \tau_x], x_i : \tau_i[x \mapsto \tau_x] \dots \vdash e[x \mapsto \tau_x] : \tau_o[x \mapsto \tau_x]$  (*n.b.*,  $(\Gamma, x_i : \tau_i \dots)[x \mapsto \tau_x] = \Gamma[x \mapsto \tau_x], x_i : \tau_i \dots$ ). Then applying T:LAM derives

$$\frac{\Theta; \Delta; \Gamma[x \mapsto \tau_x], x_i : \tau_i[x \mapsto \tau_x] \dots \vdash e[x \mapsto \tau_x] : \tau_o[x \mapsto \tau_x]}{\Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash (\lambda ((x_i \tau_i[x \mapsto \tau_x]) \dots) e[x \mapsto \tau_x]) : (-> (\tau_i[x \mapsto \tau_x] \dots) \tau_o[x \mapsto \tau_x])} \text{T:LAM}$$

**Case TLAM**

$$\frac{\Theta; \Delta, x :: k, x_u :: k_u \dots; \Gamma \vdash e : \tau_u}{\Theta; \Delta, x :: k; \Gamma \vdash (\text{T}\lambda ((x_u k_u) \dots) e) : (\forall ((x_u k_u) \dots) \tau_u)} \text{T:TLAM}$$

By the induction hypothesis,  $\Theta; \Delta, x_u :: k_u \dots; \Gamma[x \mapsto \tau_x] \vdash e[x \mapsto \tau_x] : \tau_u[x \mapsto \tau_x]$  (per Barendregt's convention,  $x \notin x_u \dots$ ). We then derive

$$\frac{\Theta; \Delta, x_u :: k_u \dots; \Gamma[x \mapsto \tau_x] \vdash e[x \mapsto \tau_x] : \tau_u[x \mapsto \tau_x]}{\Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash (\text{T}\lambda ((x_u k_u) \dots) e[x \mapsto \tau_x]) : (\forall ((x_u k_u) \dots) \tau_u[x \mapsto \tau_x])} \text{T:TLAM}$$

**Case ILAM**

$$\frac{\Theta, x_p :: \gamma_p \dots ; \Delta, x :: k; \Gamma \vdash e : \tau_p}{\Theta; \Delta, x :: k; \Gamma \vdash (\text{I}\lambda ((x_p \gamma_p) \dots) e) : (\text{P} ((x_p \gamma_p) \dots) \tau_p)} \text{T:ILAM}$$

The induction hypothesis implies

$$\Theta, x_p :: \gamma_p \dots ; \Delta; \Gamma[x \mapsto \tau_x] \vdash e[x \mapsto \tau_x] : \tau_p[x \mapsto \tau_x]$$

Then we can derive

$$\frac{\Theta, x_p :: \gamma_p \dots ; \Delta; \Gamma[x \mapsto \tau_x] \vdash e[x \mapsto \tau_x] : \tau_p[x \mapsto \tau_x]}{\Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash (\text{I}\lambda ((x_p \gamma_p) \dots) e[x \mapsto \tau_x]) : (\text{P} ((x_p \gamma_p) \dots) \tau_p[x \mapsto \tau_x])} \text{T:ILAM}$$

**Case BOX**

$$\frac{\Theta \vdash \iota_s :: \gamma_s \dots \quad \Theta; \Delta, x :: k \vdash (\Sigma ((x_s \gamma_s) \dots) \tau_s) :: \text{Atom} \quad \Theta; \Delta, x :: k; \Gamma \vdash e : \tau_s[x_s \mapsto \iota_s, \dots]}{\Theta; \Delta, x :: k; \Gamma \vdash (\text{box } \iota_s \dots e (\Sigma ((x_s \gamma_s) \dots) \tau_s)) : (\Sigma ((x_s \gamma_s) \dots) \tau_s)} \text{T:BOX}$$

By the induction hypothesis,

$$\Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash e : \tau_s[x_s \mapsto \iota_s, \dots][x \mapsto \tau_x]$$

Preservation of kinds under type substitution gives a derivation for

$$\Theta; \Delta \vdash (\Sigma ((x_s \gamma_s) \dots) \tau_s[x \mapsto \tau_x]) :: \text{Atom}$$

By merging substitutions, the ascribed type  $\tau_s[x_s \mapsto \iota_s, \dots][x \mapsto \tau_x]$  is equal to  $\tau_s[x_s \mapsto \iota_s, \dots, x \mapsto \tau_x]$ . We then derive

$$\frac{\Theta \vdash \iota_s :: \gamma_s \dots \quad \Theta; \Delta \vdash (\Sigma ((x_s \gamma_s) \dots) \tau_s[x \mapsto \tau_x]) :: \text{Atom} \quad \Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash e : \tau_s[x_s \mapsto \iota_s \dots, x \mapsto \tau_x]}{\Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash (\text{box } \iota_s \dots e[x \mapsto \tau_x] (\Sigma ((x_s \gamma_s) \dots) \tau_s[x \mapsto \tau_x])) : (\Sigma ((x_s \gamma_s) \dots) \tau_s[x \mapsto \tau_x])} \text{T:BOX}$$

**Case TAPP**

$$\frac{\Theta; \Delta, x :: k; \Gamma \vdash e : (A (\forall ((x_u k_u) \dots) (A \tau_u \iota_u)) \iota_f) \quad \Theta; \Delta, x :: k \vdash \tau_a :: k_u \dots}{\Theta; \Delta, x :: k; \Gamma \vdash (\text{t-app } e \tau_a \dots) : (A \tau_u[x_u \mapsto \tau_a, \dots] (\text{++ } \iota_f \iota_u))} \text{T:TAPP}$$

The induction hypothesis gives  $\Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash e[x \mapsto \tau_x] : (A (\forall ((x_u k_u) \dots) (A \tau_u[x \mapsto \tau_x] \iota_u)) \iota_f)$ . Lemma 4.2.5 (preservation of

kinds) implies for each of  $\tau_a \dots$  that  $\Theta; \Delta \vdash \tau_a[x \mapsto \tau_x] :: k_u$ . Then we can derive

$$\begin{array}{c} \Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash e[x \mapsto \tau_x] \\ : (A (\forall ((x_u k_u) \dots) (A \tau_u[x \mapsto \tau_x] l_u))) l_f \\ \\ \frac{\Theta; \Delta \vdash \tau_a[x \mapsto \tau_x] :: k_u \dots}{\Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash} \quad \text{T:TAPP} \\ (t\text{-app } e[x \mapsto \tau_x] \tau_a[x \mapsto \tau_x] \dots) \\ : (A \tau_u[x \mapsto \tau_x][x_u \mapsto \tau_a, \dots] (++) l_f l_u)) \end{array}$$

### Case IAPP

$$\begin{array}{c} \Theta; \Delta, x :: k; \Gamma \vdash e : (A (\Pi ((x_p \gamma_p) \dots) (A \tau_p l_p))) l_f \\ \Theta \vdash l_a :: \gamma_p \dots \\ \\ \frac{}{\Theta; \Delta, x :: k; \Gamma \vdash (i\text{-app } e l_a \dots)} \quad \text{T:IAPP} \\ : (A \tau_p[x_p \mapsto l_a, \dots] (++) l_f l_p[x_p \mapsto l_a, \dots])) \end{array}$$

By the induction hypothesis,  $\Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash e[x \mapsto \tau_x] : (A (\Pi ((x_p \gamma_p) \dots) (A \tau_p[x \mapsto \tau_x] l_p))) l_f$ . Applying T:IAPP produces the derivation

$$\begin{array}{c} \Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash e[x \mapsto \tau_x] \\ : (A (\Pi ((x_p \gamma_p) \dots) (A \tau_p[x \mapsto \tau_x] l_p))) l_f \\ \\ \frac{\Theta \vdash l_a :: \gamma_p \dots}{\Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash (i\text{-app } e[x \mapsto \tau_x] l_a \dots)} \quad \text{T:IAPP} \\ : (A \tau_p[x_p \mapsto l_a, \dots] (++) l_f l_p[x_p \mapsto l_a, \dots])) \end{array}$$

### Case APP

$$\begin{array}{c} \Theta; \Delta, x :: k; \Gamma \vdash e_f : (A (-> ((A \tau_i l_i) \dots) (A \tau_o l_o))) l_f \\ \Theta; \Delta, x :: k; \Gamma \vdash e_a : (A \tau_i (++) l_a l_i)) \dots \\ l_p = \text{Max} \llbracket l_f l_a \dots \rrbracket \\ \\ \frac{}{\Theta; \Delta, x :: k; \Gamma \vdash (e_f e_a \dots) : (A \tau_o (++) l_p l_o))} \quad \text{T:APP} \end{array}$$

The induction hypothesis gives derivations for

$$\begin{array}{c} \Theta; \Delta, x :: k; \Gamma[x \mapsto \tau_x] \vdash e_f[x \mapsto \tau_x] : \\ (A (-> ((A \tau_i[x \mapsto \tau_x] l_i) \dots) \\ (A \tau_o[x \mapsto \tau_x] l_o))) l_f \end{array}$$

It also gives for each of the  $e_a \dots$  a derivation for

$$\Theta; \Delta, x :: k; \Gamma[x \mapsto \tau_x] \vdash e_a[x \mapsto \tau_x] : (A \tau_i[x \mapsto \tau_x] (++) l_a l_i))$$

Note that the frames of the function and argument arrays are unchanged, so the principal frame  $\iota_p$  is also unchanged. We then derive

$$\begin{array}{c}
 \Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash e_f[x \mapsto \tau_x] : \\
 (A \ (-> \ ((A \ \tau_i[x \mapsto \tau_x] \ \iota_i) \ \dots) \\
 \quad (A \ \tau_o[x \mapsto \tau_x] \ \iota_o))) \\
 \\
 \Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash e_a[x \mapsto \tau_x] : (A \ \tau_i[x \mapsto \tau_x] \ (++) \ \iota_a \ \iota_i) \ \dots \\
 \\
 \frac{\iota_p = \text{Max} \llbracket \iota_f \ \iota_a \ \dots \rrbracket}{\Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash} \quad \text{T:APP} \\
 (e_f[x \mapsto \tau_x] \ e_a[x \mapsto \tau_x] \ \dots) \\
 : (A \ \tau_o[x \mapsto \tau_x] \ (++) \ \iota_p \ \iota_o)
 \end{array}$$

### Case UNBOX

$$\begin{array}{c}
 \Theta; \Delta, x :: k; \Gamma \vdash e_s : (A \ (\Sigma \ ((x'_i \ \gamma_i) \ \dots) \ \tau_s) \ \iota_s) \\
 \Theta, x_i :: \gamma_i \ \dots; \Delta, x :: k; \Gamma, x_e : \tau_s[x'_i \mapsto x_i, \dots] \vdash \\
 \quad e_b : (A \ \tau_b \ \iota_b) \\
 \Theta; \Delta, x :: k \vdash (A \ \tau_b \ \iota_b) :: \text{Array} \\
 \hline
 \Theta; \Delta, x :: k; \Gamma \vdash (\text{unbox } (x_i \ \dots \ x_e \ e_s) \ e_b) \\
 : (A \ \tau_b \ (++) \ \iota_s \ \iota_b)
 \end{array} \quad \text{T:UNBOX}$$

By the induction hypothesis,

$$\begin{array}{c}
 \Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash e_s[x \mapsto \tau_x] : \\
 (A \ (\Sigma \ ((x'_i \ \gamma_i) \ \dots) \ \tau_s[x \mapsto \tau_x]) \ \iota_s)
 \end{array}$$

and

$$\begin{array}{c}
 \Theta, x_i :: \gamma_i \ \dots; \Delta; \Gamma[x \mapsto \tau_x], x_e : (\tau_s[x'_i \mapsto x_i, \dots])[x \mapsto \tau_x] \\
 \vdash e_b[x \mapsto \tau_x] : (A \ \tau_b \ \iota_b)[x \mapsto \tau_x]
 \end{array}$$

Preservation of kinds implies  $\Theta; \Delta \vdash (A \ \tau_b \ \iota_b)[x \mapsto \tau_x] :: \text{Array}$ . Then we can derive

$$\begin{array}{c}
 \Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash e_s[x \mapsto \tau_x] \\
 : (A \ (\Sigma \ ((x'_i \ \gamma_i) \ \dots) \ \tau_s[x \mapsto \tau_x]) \ \iota_s) \\
 \\
 \Theta, x_i :: \gamma_i \ \dots; \Delta; \\
 \Gamma[x \mapsto \tau_x], x_e : (\tau_s[x'_i \mapsto x_i, \dots])[x \mapsto \tau_x] \\
 \vdash e_b[x \mapsto \tau_x] : (A \ \tau_b \ \iota_b)[x \mapsto \tau_x] \\
 \\
 \Theta; \Delta \vdash (A \ \tau_b \ \iota_b)[x \mapsto \tau_x] :: \text{Array} \\
 \hline
 \Theta; \Delta; \Gamma[x \mapsto \tau_x] \vdash \\
 (\text{unbox } (x_i \ \dots \ x_e \ e_s[x \mapsto \tau_x]) \ e_b[x \mapsto \tau_x]) \\
 : (A \ \tau_b \ (++) \ \iota_s \ \iota_b)[x \mapsto \tau_x]
 \end{array} \quad \text{T:UNBOX}$$

□

**Lemma 4.2.15** (Preservation of types under term substitution). *Given  $\Theta; \Delta; \Gamma, x : \tau_x \vdash t : \tau$  and  $\Theta; \Delta; \Gamma \vdash e_x : \tau_x$  then  $\Theta; \Delta; \Gamma \vdash t[x \mapsto e_x] : \tau$ .*

*Proof.* We use induction on the derivation of  $\Theta; \Delta; \Gamma, x : \tau_x \vdash t : \tau$ .

**Case EQV**

$$\frac{\Theta; \Delta; \Gamma, x : \tau_x \vdash t : \tau' \quad \tau \cong \tau'}{\Theta; \Delta; \Gamma, x : \tau_x \vdash t : \tau} \text{ T:EQV}$$

By the induction hypothesis,  $\Theta; \Delta; \Gamma \vdash t[x \mapsto e_x] : \tau'$ . So we can derive

$$\frac{\Theta; \Delta; \Gamma \vdash t[x \mapsto e_x] : \tau' \quad \tau \cong \tau'}{\Theta; \Delta; \Gamma, x : \tau_x \vdash t[x \mapsto e_x] : \tau} \text{ T:EQV}$$

**Case VAR**

$$\frac{(x' : \tau) \in \Gamma}{\Theta; \Delta; \Gamma, x : \tau_x \vdash x' : \tau} \text{ T:VAR}$$

Suppose  $x' = x$ . Then  $x'[x \mapsto e_x] = e_x$ , and by assumption,  $\Theta; \Delta; \Gamma \vdash e_x : \tau_x$ . Since the type environment maps  $x$  to both  $\tau$  and  $\tau_x$ , we know that  $\tau = \tau_x$ . Therefore,  $\Theta; \Delta; \Gamma \vdash e_x : \tau$ .

Otherwise,  $x' \neq x$ , and  $x'[x \mapsto e_x] = x'$ . The type environment still contains  $(x' : \tau)$ , so we can still derive

$$\frac{(x' : \tau) \in \Gamma}{\Theta; \Delta; \Gamma \vdash x' : \tau} \text{ T:VAR}$$

**Case OP, BASE**

The variable  $x$  cannot appear free in  $t$ , so  $t[x \mapsto e_x] = t$ . Neither T:OP nor T:BASE depends on the environment, so the same rule which produced the original derivation can also derive  $\Theta; \Delta; \Gamma \vdash t : \tau$ .

**Case ARRAY**

$$\frac{\Theta; \Delta; \Gamma, x : \tau_x \vdash a : \tau_a \dots \quad \Theta; \Delta \vdash \tau_a :: \text{Atom} \quad \text{Length} \llbracket a \dots \rrbracket = \prod n \dots}{\Theta; \Delta; \Gamma, x : \tau_x \vdash (\text{array } (n \dots) a \dots) : (A \tau_a (\text{shape } n \dots))} \text{ T:ARRAY}$$

By the induction hypothesis,  $\Theta; \Delta; \Gamma \vdash a[x \mapsto e_x] : \tau_a$  for each of  $a \dots$ . So we can derive

$$\frac{\Theta; \Delta; \Gamma \vdash a[x \mapsto e_x] : \tau_a \dots \quad \Theta; \Delta \vdash \tau_a :: \text{Atom} \quad \text{Length} \llbracket a \dots \rrbracket = \prod n \dots}{\Theta; \Delta; \Gamma \vdash (\text{array } (n \dots) a[x \mapsto e_x] \dots) : (A \tau_a (\text{shape } n \dots))} \text{ T:ARRAY}$$



**Case FRAME**

$$\frac{\Theta; \Delta; \Gamma, x : \tau_x \vdash e_a : (A \tau_a \iota_a) \dots \quad \Theta; \Delta \vdash (A \tau_a \iota_a) :: \text{Array} \quad \text{Length} \llbracket \bar{a} \dots \rrbracket = \prod n \dots}{\Theta; \Delta; \Gamma, x : \tau_x \vdash (\text{frame } (n \dots) e_a \dots) : (A \tau_a (++) (\text{shape } n \dots) \iota_a)} \text{T:FRAME}$$

The induction hypothesis implies for each of  $e_a \dots$  that  $\Theta; \Delta; \Gamma \vdash e_a : (A \tau_a \iota_a)$ . This leads to

$$\frac{\Theta; \Delta; \Gamma \vdash e_a[x \mapsto e_x] : (A \tau_a \iota_a) \dots \quad \Theta; \Delta \vdash (A \tau_a \iota_a) :: \text{Array} \quad \text{Length} \llbracket \bar{a} \dots \rrbracket = \prod n \dots}{\Theta; \Delta; \Gamma \vdash (\text{frame } (n \dots) e_a[x \mapsto e_x] \dots) : (A \tau_a (++) (\text{shape } n \dots) \iota_a)} \text{T:FRAME}$$

**Case EMPTYA, EMPTYF**

For each of these rules, the only premise is a kind check, which does not mention the conclusion's type environment. So  $\Theta; \Delta; \Gamma \vdash t[x \mapsto e_x] : \tau$  can be derived from the same premise.

**Case LAM**

$$\frac{\Theta; \Delta; \Gamma, x : \tau_x, x_i : \tau_i \dots \vdash e : \tau_o}{\Theta; \Delta; \Gamma, x : \tau_x \vdash (\lambda ((x_i \tau_i) \dots) e) : (-> (\tau_i \dots) \tau_o)} \text{T:LAM}$$

The induction hypothesis gives  $\Theta; \Delta; \Gamma, x_i : \tau_i \dots \vdash e[x \mapsto e_x] : \tau_o$ , so we derive

$$\frac{\Theta; \Delta; \Gamma, x_i : \tau_i \dots \vdash e[x \mapsto e_x] : \tau_o}{\Theta; \Delta; \Gamma, x : \tau_x \vdash (\lambda ((x_i \tau_i) \dots) e[x \mapsto e_x]) : (-> (\tau_i \dots) \tau_o)} \text{T:LAM}$$

**Case TLAM**

$$\frac{\Theta; \Delta, x_u :: k_u \dots; \Gamma, x : \tau_x \vdash e : \tau_u}{\Theta; \Delta; \Gamma, x : \tau_x \vdash (\text{T}\lambda ((x_u k_u) \dots) e) : (\forall ((x_u k_u) \dots) \tau_u)} \text{T:TLAM}$$

By the induction hypothesis,  $\Theta; \Delta, x_u :: k_u \dots; \Gamma \vdash e[x \mapsto e_x] : \tau_u$ . We then derive

$$\frac{\Theta; \Delta, x_u :: k_u \dots; \Gamma \vdash e[x \mapsto e_x] : \tau_u}{\Theta; \Delta; \Gamma \vdash (\text{T}\lambda ((x_u k_u) \dots) e[x \mapsto e_x]) : (\forall ((x_u k_u) \dots) \tau_u)} \text{T:TLAM}$$

**Case ILAM**

$$\frac{\Theta, x_p :: \gamma_p \dots; \Delta; \Gamma, x : \tau_x \vdash e : \tau_p}{\Theta; \Delta; \Gamma, x : \tau_x \vdash (\text{I}\lambda ((x_p \gamma_p) \dots) e) : (\prod ((x_p \gamma_p) \dots) \tau_p)} \text{T:ILAM}$$

The induction hypothesis implies  $\Theta, x_p :: \gamma_p \dots ; \Delta; \Gamma \vdash e[x \mapsto e_x] : \tau_p$ , which leads to the derivation

$$\frac{\Theta, x_p :: \gamma_p \dots ; \Delta; \Gamma \vdash e[x \mapsto e_x] : \tau_p}{\Theta; \Delta; \Gamma \vdash (\text{I}\lambda ((x_p \gamma_p) \dots) e[x \mapsto e_x]) : (\Pi ((x_p \gamma_p) \dots) \tau_p)} \text{T:ILAM}$$

### Case BOX

$$\frac{\Theta \vdash \iota_s :: \gamma_s \dots \quad \Theta; \Delta \vdash (\Sigma ((x_s \gamma_s) \dots) \tau_s) :: \text{Atom} \quad \Theta; \Delta; \Gamma, x : \tau_x \vdash e : \tau_s[x_s \mapsto \iota_s, \dots]}{\Theta; \Delta; \Gamma, x : \tau_x \vdash (\text{box } \iota_s \dots e (\Sigma ((x_s \gamma_s) \dots) \tau_s)) : (\Sigma ((x_s \gamma_s) \dots) \tau_s)} \text{T:BOX}$$

By the induction hypothesis,  $\Theta; \Delta; \Gamma \vdash e[x \mapsto e_x] : \tau_s[x_s \mapsto \iota_s, \dots]$ . We then derive

$$\frac{\Theta \vdash \iota_s :: \gamma_s \dots \quad \Theta; \Delta \vdash (\Sigma ((x_s \gamma_s) \dots) \tau_s) :: \text{Atom} \quad \Theta; \Delta; \Gamma \vdash e[x \mapsto e_x] : \tau_s[x_s \mapsto \iota_s, \dots]}{\Theta; \Delta; \Gamma \vdash (\text{box } \iota_s \dots e[x \mapsto e_x] (\Sigma ((x_s \gamma_s) \dots) \tau_s)) : (\Sigma ((x_s \gamma_s) \dots) \tau_s)} \text{T:BOX}$$

### Case TAPP

$$\frac{\Theta; \Delta; \Gamma, x : \tau_x \vdash e : (A (\forall ((x_u k_u) \dots) (A \tau_u \iota_u))) \iota_f \quad \Theta; \Delta \vdash \tau_a :: k_u \dots}{\Theta; \Delta; \Gamma, x : \tau_x \vdash (\text{t-app } e \tau_a \dots) : (A \tau_u[x_u \mapsto \tau_a, \dots] (++) \iota_f \iota_u))} \text{T:TAPP}$$

The induction hypothesis gives a derivation for  $\Theta; \Delta; \Gamma \vdash e[x \mapsto e_x] : (A (\forall ((x_u k_u) \dots) (A \tau_u \iota_u))) \iota_f$ . We can then construct the derivation

$$\frac{\Theta; \Delta; \Gamma \vdash e[x \mapsto e_x] : (A (\forall ((x_u k_u) \dots) (A \tau_u \iota_u))) \iota_f \quad \Theta; \Delta \vdash \tau_a :: k_u \dots}{\Theta; \Delta; \Gamma \vdash (\text{t-app } e[x \mapsto e_x] \tau_a \dots) : (A \tau_u[x_u \mapsto \tau_a, \dots] (++) \iota_f \iota_u))} \text{T:TAPP}$$

### Case IAPP

$$\frac{\Theta; \Delta; \Gamma, x : \tau_x \vdash e : (A (\Pi ((x_p \gamma_p) \dots) (A \tau_p \iota_p))) \iota_f \quad \Theta \vdash \iota_a :: \gamma_p \dots}{\Theta; \Delta; \Gamma, x : \tau_x \vdash (\text{i-app } e \iota_a \dots) : (A \tau_p[x_p \mapsto \iota_a, \dots] (++) \iota_f \iota_p[x_p \mapsto \iota_a, \dots]))} \text{T:IAPP}$$

By the induction hypothesis, we have  $\Theta; \Delta; \Gamma, x : \tau_x \vdash e[x \mapsto e_x] : (A (\Pi ((x_p \gamma_p) \dots) (A \tau_p \iota_p))) \iota_f$ . This leads to

$$\frac{\Theta; \Delta; \Gamma \vdash e[x \mapsto e_x] : (A (\Pi ((x_p \gamma_p) \dots) (A \tau_p \iota_p))) \iota_f \quad \Theta \vdash \iota_a :: \gamma_p \dots}{\Theta; \Delta; \Gamma \vdash (\text{i-app } e[x \mapsto e_x] \iota_a \dots) : (A \tau_p [x_p \mapsto \iota_a, \dots] (++) \iota_f \iota_p [x_p \mapsto \iota_a, \dots]))} \text{T:IAPP}$$

### Case APP

$$\frac{\Theta; \Delta; \Gamma, x : \tau_x \vdash e_f : (A (-> ((A \tau_i \iota_i) \dots) (A \tau_o \iota_o))) \iota_f \quad \Theta; \Delta; \Gamma, x : \tau_x \vdash e_a : (A \tau_i (++) \iota_a \iota_i) \dots \quad \iota_p = \text{Max} \llbracket \iota_f \iota_a \dots \rrbracket}{\Theta; \Delta; \Gamma, x : \tau_x \vdash (e_f e_a \dots) : (A \tau_o (++) \iota_p \iota_o)} \text{T:APP}$$

The induction hypothesis implies

$$\Theta; \Delta; \Gamma \vdash e_f[x \mapsto e_x] : (A (-> ((A \tau_i \iota_i) \dots) (A \tau_o \iota_o))) \iota_f$$

and  $\Theta; \Delta; \Gamma \vdash e_a[x \mapsto e_x] : (A \tau_i (++) \iota_a \iota_i)$  for each of  $e_a \dots$ . Since the individual frames  $\iota_f, \iota_a \dots$  are unchanged, so is the principal frame  $\iota_p$ . Thus we derive

$$\frac{\Theta; \Delta; \Gamma \vdash e_f[x \mapsto e_x] : (A (-> ((A \tau_i \iota_i) \dots) (A \tau_o \iota_o))) \iota_f \quad \Theta; \Delta; \Gamma \vdash e_a[x \mapsto e_x] : (A \tau_i (++) \iota_a \iota_i) \dots \quad \iota_p = \text{Max} \llbracket \iota_f \iota_a \dots \rrbracket}{\Theta; \Delta; \Gamma \vdash (e_f[x \mapsto e_x] e_a[x \mapsto e_x] \dots) : (A \tau_o (++) \iota_p \iota_o)} \text{T:APP}$$

### Case UNBOX

$$\frac{\Theta; \Delta; \Gamma, x : \tau_x \vdash e_s : (A (\Sigma (x'_i \dots \gamma_i) \tau_s) \iota_s) \quad \Theta, x_i :: \gamma_i \dots; \Delta; \Gamma, x_e : \tau_s [x'_i \mapsto x_i, \dots], x : \tau_x \vdash e_b : (A \tau_b \iota_b) \quad \Theta; \Delta, x :: k \vdash (A \tau_b \iota_b) :: \text{Array}}{\Theta; \Delta; \Gamma, x : \tau_x \vdash (\text{unbox } (x_i \dots x_e e_s) e_b) : (A \tau_b (++) \iota_s \iota_b)} \text{T:UNBOX}$$

The induction hypothesis gives

$$\Theta; \Delta; \Gamma \vdash e_s[x \mapsto e_x] : (A (\Sigma ((x'_i \gamma_i) \dots) \tau_s) \iota_s)$$

and

$$\Theta, x_i :: \gamma_i \dots; \Delta; \Gamma, x_e : \tau_s [x'_i \mapsto x_i, \dots] \vdash e_b[x \mapsto e_x] : (A \tau_b \iota_b)$$

We then derive

$$\begin{array}{c}
\Theta; \Delta; \Gamma \vdash e_s[x \mapsto e_x] : (A (\Sigma (x'_i \dots \gamma_i) \tau_s) \iota_s) \\
\Theta, x_i :: \gamma_i \dots; \Delta; \Gamma, x_e : \tau_s[x'_i \mapsto x_i, \dots] \vdash \\
e_b[x \mapsto e_x] : (A \tau_b \iota_b) \\
\Theta; \Delta, x :: k \vdash \tau_b :: \text{Array} \\
\hline
\Theta; \Delta; \Gamma \vdash \\
(\text{unbox } (x_i \dots x_e e_s[x \mapsto e_x]) e_b[x \mapsto e_x]) \\
: (A \tau_b (++) \iota_s \iota_b)
\end{array} \quad \text{T:UNBOX}$$

□

**Theorem 4.2.4** (Ascription of well-kinded types). *Given  $\Theta; \Delta; \Gamma \vdash t : \tau$  where  $\Theta; \Delta \vdash \Gamma$ :*

- *If  $t$  is an expression, then  $\Theta; \Delta \vdash \tau :: \text{Array}$*
- *If  $t$  is an atom, then  $\Theta; \Delta \vdash \tau :: \text{Atom}$*

*Proof.* We use induction on the derivation of  $\Theta; \Delta; \Gamma \vdash t : \tau$ . First, we prove the cases where  $t$  is an expression.

**Case VAR:**

$$\frac{(x : \tau) \in \Gamma}{\Theta; \Delta; \Gamma \vdash x : \tau} \quad \text{T:ARRAY}$$

Since  $x : \tau \in \Gamma$ , this follows directly from  $\Theta; \Delta \vdash \Gamma$ .

**Case ARRAY:**

$$\begin{array}{c}
\Theta; \Delta; \Gamma \vdash a_j : \tau_a \dots \\
\Theta; \Delta \vdash \tau_a :: \text{Atom} \\
\text{Length} \llbracket a \dots \rrbracket = \prod n \dots \\
\hline
\Theta; \Delta; \Gamma \vdash (\text{array } (n \dots) a \dots) : (A \tau_a (\text{shape } n \dots))
\end{array} \quad \text{T:ARRAY}$$

We can derive

$$\begin{array}{c}
\frac{\frac{\frac{}{\Theta \vdash n :: \text{Dim}}{\text{S:NAT}} \dots}{\Theta \vdash (\text{shape } n \dots) :: \text{Shape}}{\text{S:SHAPE}}}{\Theta; \Delta \vdash \tau_a :: \text{Atom} \quad \Theta \vdash (\text{shape } n \dots) :: \text{Shape}}{\text{K:ARRAY}} \\
\Theta; \Delta \vdash (A \tau_a (\text{shape } n \dots)) :: \text{Array}
\end{array}$$

**Case FRAME:**

$$\begin{array}{c}
\Theta; \Delta; \Gamma \vdash e : (A \tau_c \iota_c) \dots \\
\Theta; \Delta \vdash (A \tau_c \iota_c) :: \text{Array} \quad \text{Length} \llbracket e \dots \rrbracket = \prod (n \dots) \\
\hline
\Theta; \Delta; \Gamma \vdash (\text{frame } (n \dots) e \dots) \\
: (A \tau_c (++) (\text{shape } n \dots) \iota_c)
\end{array} \quad \text{T:FRAME}$$

The second premise must be derived as

$$\frac{\Theta \vdash \iota_c :: \text{Shape} \quad \Theta; \Delta \vdash \tau_c :: \text{Atom}}{\Theta; \Delta \vdash (A \tau_c \iota_c) :: \text{Array}} \text{K:ARRAY}$$

Using this knowledge about  $\iota_c$ , we can show the array type's shape is indeed a Shape:

$$\frac{\Theta \vdash \iota_c :: \text{Shape} \quad \frac{\frac{\text{S:NAT}}{\Theta \vdash n :: \text{Dim}} \quad \dots}{\Theta \vdash (\text{shape } n \dots) :: \text{Shape}} \text{S:SHAPE}}{\Theta \vdash (++) (\text{shape } n \dots) \iota_c :: \text{Shape}} \text{S:APPEND}$$

Then we can derive

$$\frac{\Theta; \Delta \vdash \tau_c :: \text{Atom} \quad \Theta \vdash (++) (\text{shape } n \dots) \iota_c :: \text{Shape}}{\Theta; \Delta \vdash (A \tau_c (++) (\text{shape } n \dots) \iota_c) :: \text{Array}} \text{K:ARRAY}$$

**Case EMPTYA:**

$$\frac{\Theta; \Delta \vdash \tau_a :: \text{Atom} \quad 0 \in n \dots}{\Theta; \Delta; \Gamma \vdash (\text{array } (n \dots) \tau_a) : (A \tau_a (\text{shape } n \dots))} \text{T:EMPTYA}$$

We derive

$$\frac{\Theta; \Delta \vdash \tau_a :: \text{Atom} \quad \frac{\frac{\text{S:NAT}}{\Theta \vdash n :: \text{Dim}} \quad \dots}{\Theta \vdash (\text{shape } n \dots) :: \text{Shape}} \text{S:SHAPE}}{\Theta; \Delta \vdash (A \tau_a (\text{shape } n \dots)) :: \text{Array}} \text{K:ARRAY}$$

**Case EMPTYF:**

$$\frac{\Theta; \Delta \vdash \tau_a :: \text{Atom} \quad \Theta \vdash \iota :: \text{Shape} \quad 0 \in n \dots}{\Theta; \Delta; \Gamma \vdash (\text{frame } (n \dots) (A \tau_a \iota_c)) : (A \tau_a (++) (\text{shape } n \dots) \iota_c)} \text{T:EMPTYF}$$

First, we show that the array type's shape has sort Shape:

$$\frac{\Theta \vdash \iota_c :: \text{Shape} \quad \frac{\frac{\text{S:NAT}}{\Theta \vdash n :: \text{Dim}} \quad \dots}{\Theta \vdash (\text{shape } n \dots) :: \text{Shape}} \text{S:SHAPE}}{\Theta \vdash (++) (\text{shape } n \dots) \iota_c :: \text{Shape}} \text{S:APPEND}$$

Then we derive

$$\frac{\Theta; \Delta \vdash \tau_a :: \text{Atom} \quad \Theta \vdash (++) (\text{shape } n \dots) \iota_c :: \text{Shape}}{\Theta; \Delta \vdash (A \tau_a (++) (\text{shape } n \dots) \iota_c) :: \text{Array}} \text{K:ARRAY}$$

**Case TAPP:**

$$\frac{\Theta; \Delta; \Gamma \vdash e_f : (A (\forall ((x k) \dots) (A \tau_u \iota_u))) \iota_f \quad \Theta; \Delta \vdash \tau_a :: k \dots}{\Theta; \Delta; \Gamma \vdash (\text{t-app } e_f \tau_a \dots) : (A \tau_u[x \mapsto \tau_a, \dots] (++) \iota_f \iota_u))} \text{T:TAPP}$$

The induction hypothesis gives a kind derivation for the type of  $e_f$ :

$$\frac{\Theta \vdash \iota_f :: \text{Shape} \quad \Theta; \Delta \vdash (\forall ((x k) \dots) (A \tau_u \iota_u)) :: \text{Atom}}{\Theta; \Delta \vdash (A (\forall ((x k) \dots) (A \tau_u \iota_u))) \iota_f :: \text{Array}} \text{K:ARRAY}$$

The derivation for the second premise must have the following structure:

$$\frac{\Theta \vdash \iota_u :: \text{Shape} \quad \Theta; \Delta, x :: k \dots \vdash \tau_u :: \text{Atom}}{\Theta; \Delta, x :: k \dots \vdash (A \tau_u \iota_u) :: \text{Array}} \text{K:ARRAY} \\ \frac{}{\Theta; \Delta \vdash (\forall ((x k) \dots) (A \tau_u \iota_u)) :: \text{Atom}} \text{K:UNIV}$$

Using the kind derivation for  $\tau_u$  in the extended environment, we apply Lemma 4.2.5 (type substitution preserves kinds) to get

$$\Theta; \Delta \vdash \tau_u[x \mapsto \tau_a, \dots] :: \text{Atom}$$

Then we can derive

$$\frac{\Theta \vdash \iota_f :: \text{Shape} \quad \Theta \vdash \iota_u :: \text{Shape}}{\Theta \vdash (++) \iota_f \iota_u :: \text{Shape}} \text{S-APPEND} \\ \frac{\Theta; \Delta \vdash \tau_u[x \mapsto \tau_a, \dots] :: \text{Atom} \quad \Theta \vdash (++) \iota_f \iota_u :: \text{Shape}}{\Theta; \Delta \vdash (A \tau_u[x \mapsto \tau_a, \dots] (++) \iota_f \iota_u) :: \text{Array}} \text{K:ARRAY}$$

**Case IAPP:**

$$\frac{\Theta; \Delta; \Gamma \vdash e_f : (A (\Pi ((x \gamma) \dots) (A \tau_p \iota_p))) \iota_f \quad \Theta \vdash \iota_a :: \gamma \dots}{\Theta; \Delta; \Gamma \vdash (\text{i-app } e_f \iota_a \dots) : (A \tau_p[x \mapsto \iota_a, \dots] (++) \iota_f \iota_p[x \mapsto \iota_a, \dots]))} \text{T:IAPP}$$

The induction hypothesis implies that the type of  $e_f$  has kind Array, giving us a derivation which ends as follows:

$$\frac{\Theta \vdash \iota_f :: \text{Shape} \quad \Theta; \Delta \vdash (\Pi ((x \gamma) \dots) (A \tau_p \iota_p)) :: \text{Atom}}{\Theta; \Delta \vdash (A (\Pi ((x \gamma) \dots) (A \tau_p \iota_p))) \iota_f :: \text{Array}} \text{K:ARRAY}$$

The derivation of the second premise must itself have this form:

$$\frac{\Theta, x :: \gamma \dots \vdash \iota_p :: \text{Shape} \quad \Theta, x :: \gamma \dots; \Delta \vdash \tau_p :: \text{Atom}}{\Theta, x :: \gamma \dots; \Delta \vdash (A \tau_p \iota_p) :: \text{Array}} \text{K:ARRAY} \\ \frac{}{\Theta; \Delta \vdash (\Pi ((x \gamma) \dots) (A \tau_p \iota_p)) :: \text{Atom}} \text{K:PI}$$

Using Lemma 4.2.2 (index substitution preserves sorts) implies  $\Theta \vdash \iota_p[x \mapsto \iota_a, \dots] :: \text{Shape}$ . So we derive

$$\frac{\Theta \vdash \iota_f :: \text{Shape} \quad \Theta \vdash \iota_p[x \mapsto \iota_a, \dots] :: \text{Shape}}{\Theta \vdash (++) \iota_f \iota_p[x \mapsto \iota_a, \dots] :: \text{Shape}} \text{S:APPEND}$$

Next, Lemma 4.2.4 (index substitution preserves kinds) gives us  $\Theta; \Delta \vdash \tau_p[x \mapsto \iota_a, \dots] :: \text{Atom}$ . We now have the pieces for the derivation

$$\frac{\Theta; \Delta \vdash \tau_p[x \mapsto \iota_a, \dots] :: \text{Atom} \quad \Theta \vdash (++) \iota_f \iota_p[x \mapsto \iota_a, \dots] :: \text{Shape}}{\Theta; \Delta \vdash (A \tau_p[x \mapsto \iota_a, \dots] (++) \iota_f \iota_p[x \mapsto \iota_a, \dots])) :: \text{Array}} \text{K:ARRAY}$$

**Case UNBOX:**

$$\frac{\Theta; \Delta; \Gamma \vdash e_s : (A (\Sigma ((x'_i \gamma) \dots) \tau_s) \iota_s) \quad \Theta, x_i :: \gamma \dots; \Delta; \Gamma, x_e : \tau_s[x'_i \mapsto x_i, \dots] \vdash e_b : (A \tau_b \iota_b) \quad \Theta; \Delta \vdash (A \tau_b \iota_b) :: \text{Array}}{\Theta; \Delta; \Gamma \vdash (\text{unbox } (x_i \dots x_e e_s) e_b) : (A \tau_b (++) \iota_s \iota_b))} \text{T:UNBOX}$$

The third premise must be derived by

$$\frac{\Theta; \Delta \vdash \tau_b :: \text{Atom} \quad \Theta \vdash \iota_b :: \text{Shape}}{\Theta; \Delta \vdash (A \tau_b \iota_b) :: \text{Array}} \text{K:ARRAY}$$

The induction hypothesis gives us a kinding derivation for the box array type

$$\frac{\Theta, (x'_i \gamma) \dots; \Delta \vdash \tau_s :: \text{Array} \quad \Theta \vdash \iota_s :: \text{Shape}}{\Theta; \Delta \vdash (A (\Sigma ((x'_i \gamma) \dots) \tau_s) \iota_s) :: \text{Array}} \text{K:ARRAY}$$

From these derivations, we have the pieces needed to construct the kind derivation for the result type:

$$\frac{\Theta; \Delta \vdash \tau_b :: \text{Atom} \quad \frac{\Theta \vdash \iota_s :: \text{Shape} \quad \Theta \vdash \iota_b :: \text{Shape}}{\Theta \vdash (++) \iota_s \iota_b :: \text{Shape}} \text{S-APPEND}}{\Theta; \Delta \vdash (A \tau_b (++) \iota_s \iota_b)) :: \text{Array}} \text{K:ARRAY}$$

**Case APP:**

$$\frac{\Theta; \Delta; \Gamma \vdash e_f : (A (-> ((A \tau_i \iota_i) \dots) (A \tau_o \iota_o)) \iota_f) \quad \Theta; \Delta; \Gamma \vdash e_a : (A \tau_i (++) \iota_a \iota_i)) \dots \quad \iota_p = \bigsqcup (\iota_f, \iota_a \dots)}{\Theta; \Delta; \Gamma \vdash (e_f e_a \dots) : (A \tau_o (++) \iota_p \iota_o))} \text{T:APP}$$

Since  $e_f$  is well-typed, the induction hypothesis implies that its type has kind Array. This derivation must have the form

$$\frac{\Theta \vdash \iota_f :: \text{Shape} \quad \Theta; \Delta \vdash (-> ((A \tau_i \iota_i) \dots) (A \tau_o \iota_o)) :: \text{Atom}}{\Theta; \Delta \vdash (A (-> ((A \tau_i \iota_i) \dots) (A \tau_o \iota_o)) \iota_f) :: \text{Array}} \text{K:ARRAY}$$

The second premise must be derived via

$$\frac{\Theta; \Delta \vdash (A \tau_i \iota_i) :: \text{Array} \dots \quad \Theta; \Delta \vdash (A \tau_o \iota_o) :: \text{Array}}{\Theta; \Delta \vdash (\rightarrow ((A \tau_i \iota_i) \dots) (A \tau_o \iota_o)) :: \text{Atom}} \text{K:FN}$$

Then the kinding derivation for each argument type  $(A \tau_i \iota_i)$  must include ascription of  $\text{Atom}$  to  $\tau_i$  and  $\text{Shape}$  to  $\iota_i$  (and similar for the result type  $(A \tau_o \iota_o)$ ). Any index variables which appear in  $\iota_p$  must also appear in at least one of  $\iota_f, \iota_a \dots$ , so  $\iota_p$  must be well-formed (*i.e.*,  $\Theta \vdash \iota_p :: \text{Shape}$ ). We can then derive

$$\frac{\Theta; \Delta; \tau_o \vdash \text{Atom} : \frac{\Theta \vdash \iota_p :: \text{Shape} \quad \Theta \vdash \iota_o :: \text{Shape}}{\Theta \vdash (++) \iota_p \iota_o :: \text{Shape}} \text{S-APPEND} \dots}{\Theta; \Delta \vdash (A \tau_o (++) \iota_p \iota_o) :: \text{Array}} \text{K:ARRAY}$$

Now, we describe the atom cases.

**Case TLAM:**

$$\frac{\Theta; \Delta, x :: k \dots; \Gamma \vdash e_u : \tau_u}{\Theta; \Delta; \Gamma \vdash (\text{T}\lambda ((x k) \dots) e_u) : (\forall ((x k) \dots) \tau_u)} \text{T:TLAM}$$

By the induction hypothesis,  $\Theta; \Delta, x :: k \dots \vdash \tau_u :: \text{Array}$ , so we can derive

$$\frac{\Theta; \Delta, x :: k \dots \vdash \tau_u :: \text{Array}}{\Theta; \Delta \vdash (\forall ((x k) \dots) \tau_u) :: \text{Atom}} \text{K:UNIV}$$

**Case ILAM:**

$$\frac{\Theta, x :: \gamma \dots; \Delta; \Gamma \vdash e_p : \tau_p}{\Theta; \Delta; \Gamma \vdash (\text{I}\lambda ((x \gamma) \dots) e_p) : (\prod ((x \gamma) \dots) \tau_p)} \text{T:ILAM}$$

The induction hypothesis implies  $\Theta, x :: \gamma \dots; \Delta \vdash \tau_p :: \text{Array}$ . We then derive

$$\frac{\Theta, x :: \gamma \dots; \Delta \vdash \tau_p :: \text{Array}}{\Theta; \Delta \vdash (\prod ((x \gamma) \dots) e_p) :: \text{Atom}} \text{K:PI}$$

**Case BOX:**

$$\frac{\Theta \vdash \iota :: \gamma \dots \quad \Theta; \Delta \vdash (\Sigma ((x \gamma) \dots) \tau_s) :: \text{Atom} \quad \Theta; \Delta; \Gamma \vdash e_s : \tau_s[x \mapsto \iota, \dots]}{\Theta; \Delta; \Gamma \vdash (\text{box } \iota \dots e_s (\Sigma ((x \gamma) \dots) \tau_s)) : (\Sigma ((x \gamma) \dots) \tau_s)} \text{T:BOX}$$

The required result is a premise in the original derivation.

**Case LAM:**

$$\frac{\Theta; \Delta; \Gamma \vdash e_f : \tau_o \quad \Theta; \Delta \vdash \tau_i :: \text{Array} \dots}{\Theta; \Delta; \Gamma \vdash (\lambda ((x \tau_i) \dots) e_f) : (\rightarrow (\tau_i \dots) \tau_o)} \text{T:LAM}$$



The induction hypothesis implies that  $\Theta; \Delta \vdash \tau_o :: \text{Array}$ , so we can derive

$$\frac{\Theta; \Delta \vdash \tau_i :: \text{Array} \dots \quad \Theta; \Delta \vdash \tau_o :: \text{Array}}{\Theta; \Delta \vdash (-> (\tau_i \dots) \tau_o) :: \text{Atom}} \text{K:FN}$$

In this final case,  $t$  may be an expression or an atom.

**Case EQV:**

$$\frac{\Theta; \Delta; \Gamma \vdash t : \tau' \quad \tau' \cong \tau}{\Theta; \Delta; \Gamma \vdash t : \tau} \text{T:EQV}$$

The induction hypothesis gives  $\Theta; \Delta \vdash \tau' :: k$ , where  $k = \text{Atom}$  if  $t$  is an atom and  $k = \text{Array}$  if  $t$  is an expression. The required result ascribing the same kind to  $\tau$ , that is  $\Theta; \Delta \vdash \tau :: k$ , follows directly from Lemma 4.2.10.  $\square$



## PROOFS (4.4: TYPE SOUNDNESS)

**Lemma 4.4.1** (Progress). *Given an expression  $e$  such that  $\cdot; \cdot \vdash e : \tau$ , one of the following holds:*

- $e$  is a value  $v$
- There exists  $e'$  such that  $e \mapsto e'$
- $e$  is  $\mathbb{V}[(\text{array } () \vartheta) v \dots]$  where  $\vartheta$  is a partial function applied to appropriately typed values outside its domain.

*Proof.* We use induction on the derivation of  $\cdot; \cdot \vdash e : \tau$ . We consider only cases for typing rules which apply to expressions (as opposed to atoms). Note that a T:VAR derivation is impossible with an empty type environment.

### Case ARRAY

$$\frac{\cdot; \cdot \vdash \mathfrak{a} : \tau_a \dots \quad \cdot; \cdot \vdash \tau_a :: \text{Atom} \quad \text{Length}[\llbracket \mathfrak{a} \dots \rrbracket] = \prod n \dots}{\cdot; \cdot \vdash (\text{array } (n \dots) \mathfrak{a} \dots) : (A \tau_a (\text{shape } n \dots))} \text{T:ARRAY}$$

We have two possibilities. One is that all of  $\mathfrak{a} \dots$  are atomic values,  $v \dots$ , in which case  $e = (\text{array } (n \dots) v \dots)$  is a value. Otherwise, at least one  $\mathfrak{a}$  is not an atomic value. The non-value must be of the form  $\mathfrak{a}_s = (\text{box } (x \gamma) \dots e_s \tau_a)$ , with non-value  $e_s$ . The derivation of  $\cdot; \cdot \vdash \mathfrak{a} : \tau_a$  implies via the induction hypothesis that either there exists some  $e'_s$  such that  $e_s \mapsto e'_s$  or  $e_s$  has the form  $\mathbb{V}_s[(\text{array } () \vartheta) v \dots]$  with misapplied partial function  $\vartheta$ . We build the evaluation context  $\mathbb{V} = (\text{array } (n \dots) \mathfrak{a}_0 \dots (\text{box } (x \gamma) \dots \mathbb{V}_s \tau_a) \mathfrak{a}_1 \dots)$  where  $\mathfrak{a}_0 \dots$  and  $\mathfrak{a}_1 \dots$  are the atoms appearing respectively before and after  $\mathfrak{a}_s$  in  $e$ . If  $e_s \mapsto e'_s$ , then  $\mathbb{V}[e_s] \mapsto \mathbb{V}[e'_s]$ . Otherwise,  $e = \mathbb{V}[(\text{array } () \vartheta) v \dots]$ .

### Case FRAME

$$\frac{\cdot; \cdot \vdash e_a : (A \tau_a \iota_a) \dots \quad \cdot; \cdot \vdash (A \tau_a \iota_a) :: \text{Array} \quad \text{Length}[\llbracket e \dots \rrbracket] = \prod n \dots}{\cdot; \cdot \vdash (\text{frame } (n \dots) e_a \dots) : (A \tau_a (+( \text{shape } n \dots) \iota_a))} \text{T:FRAME}$$

By the induction hypothesis, each of  $e_a \dots$  is value, is reducible, or is a primitive operator misapplication of the form  $\mathbb{V}_a[(\text{array } () \vartheta) v \dots]$ . If they are all values, then each has the form  $(\text{array } (n' \dots) v \dots)$ , so  $e \mapsto (\text{array } (n \dots n' \dots) \text{Concat}[\llbracket (v \dots) \dots \rrbracket])$ . Note that all of the array literals serving as cells in the frame must have the same shape, or their types would differ, making  $e$  ill-typed.

If some  $e_e \in e_a \dots$  is not a value, then the induction hypothesis implies that it is reducible to  $e'_e$  or it has the form  $\mathbb{V}_e [((\text{array } () \text{ v}) v \dots)]$ . We construct the evaluation context  $\mathbb{V} = (\text{frame } (n \dots) e_0 \dots e_e e_1 \dots)$ , where  $(e_a \dots) = (e_0 \dots e_e e_1 \dots)$ . If  $e_e \mapsto e'_e$ , then  $\mathbb{V}[e_e] \mapsto \mathbb{V}[e'_e]$ . Otherwise,  $e = \mathbb{V} [((\text{array } () \text{ v}) v \dots)]$ .

**Case TAPP**

$$\frac{\begin{array}{c} \cdot; \cdot \vdash e_f : (A (\forall ((x k) \dots) (A \tau_u \iota_u))) \iota_f \\ \cdot; \cdot \vdash \tau_a :: k \dots \end{array}}{\cdot; \cdot \vdash (\text{t-app } e_f \tau_a \dots) : (A \tau_u [x \mapsto \tau_a, \dots] (++) \iota_f \iota_u))} \text{T:TAPP}$$

By the induction hypothesis,  $e_f$  is a value, is reducible, or misapplies a partial function. If  $e_f$  is a value of type

$$(A (\forall ((x k) \dots) (A \tau_u \iota_u))) \iota_f$$

then Lemma 4.2.7 (canonical forms) implies that it is an array literal containing type abstractions—*i.e.*,

$$e_f = (\text{array } (n \dots) (\text{T}\lambda ((x_u k) \dots) e_u) \dots)$$

This means we have a  $t\beta$  redex. If  $e_f$  is itself reducible to  $e'_f$ , then for some context  $\mathbb{V}_f$  and redex  $e_r$ ,  $e_f = \mathbb{V}_f [e_r]$ , and  $e_r \mapsto e'_r$ . Alternatively,  $e_f = \mathbb{V}_f [((\text{array } () \text{ v}) v \dots)]$ . Either way, construct  $\mathbb{V} = (\text{t-app } \mathbb{V}_f \tau_a \dots)$ . In the first case,  $e = \mathbb{V} [e_r] \mapsto \mathbb{V} [e'_r]$ . In the second,  $e = \mathbb{V} [((\text{array } () \text{ v}) v \dots)]$ .

**Case IAPP**

$$\frac{\begin{array}{c} \cdot; \cdot \vdash e_f : (A (\Pi ((x \gamma) \dots) (A \tau_p \iota_p))) \iota_f \quad \cdot \vdash \iota_a :: \gamma \dots \\ \cdot; \cdot \vdash (\text{i-app } e_f \iota_a \dots) \\ : (A \tau_p [x \mapsto \iota_a, \dots] (++) \iota_f \iota_p [x \mapsto \iota_a, \dots]) \end{array}}{\cdot; \cdot \vdash (\text{i-app } e_f \iota_a \dots) : (A \tau_p [x \mapsto \iota_a, \dots] (++) \iota_f \iota_p [x \mapsto \iota_a, \dots])} \text{T:IAPP}$$

As in the previous case,  $e_f$  is a value, is reducible, or misapplies a partial function. If  $e_f$  is a value, the canonical forms lemma implies that it has the form  $(\text{array } (n \dots) (\text{I}\lambda ((x_p \gamma) \dots) e_p) \dots)$ , which makes  $e$  an  $i\beta$  redex.

Otherwise  $e_f$  itself is reducible, *i.e.*, of the form  $\mathbb{V}_f [e_r]$  for some redex  $e_r$  reducing to  $e'_r$ , or it is of the form  $\mathbb{V}_f [((\text{array } () \text{ v}) v \dots)]$ . Let the context  $\mathbb{V} = (\text{i-app } \mathbb{V}_f \iota_a \dots)$ . Then we have either  $e = \mathbb{V} [e_r] \mapsto \mathbb{V} [e'_r]$  or  $e = \mathbb{V} [((\text{array } () \text{ v}) v \dots)]$ .

**Case UNBOX**

$$\frac{\begin{array}{c} \cdot; \cdot \vdash e_s : (A (\Sigma ((x'_i \gamma) \dots) \tau_s) \iota_s) \\ x_i :: \gamma \dots ; x_e : \tau_s [x'_i \mapsto x_i, \dots] \vdash e_b : (A \tau_b \iota_b) \\ \cdot; \cdot \vdash (A \tau_b \iota_b) :: \text{Array} \end{array}}{\cdot; \cdot \vdash (\text{unbox } (x_i \dots x_e e_s) e_b) : (A \tau_b (++) \iota_s \iota_b)} \text{T:UNBOX}$$

By the induction hypothesis,  $e_s$  is a value, reducible, or a misapplication of a partial function. If  $e_s$  is a value, the canonical forms lemma

implies  $e_s = (\text{array } () (\text{box } \iota_s \dots v \tau))$ , so  $e$  is an *unbox* redex. If  $e_s$  is reducible, *i.e.*, it is  $\mathbb{V}_s[e_r]$  where the redex  $e_r \mapsto e'_r$ , then let  $\mathbb{V} = (\text{unbox } (x_i \dots x_e \mathbb{V}_s) e_b)$ . Thus  $e = \mathbb{V}[e_r] \mapsto \mathbb{V}[e'_r]$ . Otherwise,  $e_s = \mathbb{V}_s[(\text{array } () \mathfrak{o}) v \dots]$ , so the same construction of  $\mathbb{V}$  gives  $e = \mathbb{V}[(\text{array } () \mathfrak{o}) v \dots]$ .

**Case APP**

$$\frac{\begin{array}{l} \cdot; \cdot; \cdot \vdash e_f : (A (-> ((A \tau_i \iota_i) \dots) (A \tau_o \iota_o)) \iota_f) \\ \cdot; \cdot; \cdot \vdash e_a : (A \tau_i (++) \iota_a \iota_i) \dots \quad \iota_p = \text{Max} \llbracket \iota_f \iota_a \dots \rrbracket \end{array}}{\cdot; \cdot; \cdot \vdash (e_f e_a \dots) : (A \tau_o (++) \iota_p \iota_o)} \text{ T:APP}$$

Suppose  $e_f$  is not a value. Then the induction hypothesis implies that *either*  $e_f \mapsto e'_f$ , meaning  $e_f = \mathbb{V}_f[e_r]$  for some redex  $e_r \mapsto e'_r$  *or*  $e_f = \mathbb{V}_f[(\text{array } () \mathfrak{o}) v \dots]$  with incompatible but properly-typed arguments. Let  $\mathbb{V} = (\mathbb{V}_f e_a \dots)$ . If  $e_f \mapsto e'_f$ , with  $e_f = \mathbb{V}_f[e_r]$ , then  $e = \mathbb{V}[e_r] \mapsto \mathbb{V}[e'_r]$ . Otherwise,  $e = \mathbb{V}[(\text{array } () \mathfrak{o}) v \dots]$ .

We assume from now that  $e_f$  is a value. If any of  $e_a \dots$  is not a value, then a similar argument applies. Choose  $e_c$  as the leftmost non-value argument, so that  $(e_a \dots) = (v_a \dots e_c e'_c \dots)$ . Then induction hypothesis implies that  $e_c$  is a context  $\mathbb{V}_c$  filled by either the redex  $e_r$  or the misapplication  $(\text{array } () \mathfrak{o}) v \dots$ . Then the context  $\mathbb{V} = (e_f v_a \dots \mathbb{V}_c e'_c \dots)$ . If  $e_c = \mathbb{V}_x[e_r] \mapsto \mathbb{V}_x[e'_r]$ , then  $\mathbb{V}[e_r] \mapsto \mathbb{V}[e'_r]$ . If  $e_c = \mathbb{V}_x[(\text{array } () \mathfrak{o}) v \dots]$ , then  $e = \mathbb{V}[(\text{array } () \mathfrak{o}) v \dots]$ .

Having addressed the cases where not all of  $e_f, e_a \dots$  are values, we now consider  $e = (v_f v_a \dots)$ . By canonical forms,  $v_f$  has the form  $(\text{array } (n_f \dots) \mathfrak{f} \dots)$ . Then uniqueness of typing implies

$$\models \iota_f \equiv (\text{shape } n_f \dots)$$

We proceed by case analysis on  $\iota_f, \iota_a \dots$  (*n.b.*, per the typing derivation, they are all prefix-orderable). With one exception—where  $e$  itself is misapplication of a partial function—each line of argument leads to a particular applicable reduction rule.

**SUBCASE 1:**  $\iota_f = (\text{shape})$ , and each  $\iota_a = (\text{shape})$ . Then  $n_f \dots$  must be the empty sequence, and  $v_f$  must contain only a single atom,  $\mathfrak{f}$ . So  $v_f = (\text{array } () \mathfrak{f})$ , and the respective types of the arguments  $v_a \dots$  are  $(A \tau_i (++) \iota_a \iota_i) \dots$ , or equivalently  $(A \tau_i \iota_i) \dots$ . If  $\mathfrak{f} = \mathfrak{o}$ , then we have the (partially annotated) expression

$$e = ((\text{array } () \mathfrak{o}^{(-> ((A \tau_i \iota_i) \dots) (A \tau_o \iota_o))}) v_a^{(A \tau_i \iota_i)} \dots)$$

If  $\mathfrak{o}$  is a partial function with  $v_a \dots$  as out-of-domain inputs, such as division by zero, then the third condition of the progress lemma holds. Otherwise, we have a  $\delta$  redex.

On the other hand, if  $\bar{f} = (\lambda ((x (A \tau_i \iota_i)) \dots) e_b)$ , the annotated expression is

$$\begin{aligned} & ((\text{array } () \\ & \quad (\lambda ((x (A \tau_i \iota_i)) \dots) e_b)^{(\rightarrow ((A \tau_i \iota_i) \dots) (A \tau_o \iota_o))}) \\ & \quad v_a^{(A \tau_i \iota_i) \dots}) \end{aligned}$$

This is a  $\beta$  redex.

SUBCASE 2:  $\iota_f = (\text{shape } n_f \dots)$ , and each  $\iota_a = (\text{shape } n_f \dots)$  for nonempty sequence  $n_f \dots$ . Then  $v_f$  is

$$(\text{array } (n_f \dots) \bar{f} \dots)^{(A \rightarrow ((A \tau_i (\text{shape } n_i \dots)) \dots) (A \tau_o \iota_o)) (\text{shape } n_f \dots)}$$

and it is applied to arguments

$$v_a \dots = (\text{array } (n_f \dots n_i \dots) \bar{v} \dots)^{(A \tau_i (\text{shape } n_f \dots n_i \dots)) \dots}$$

This is a *map* redex.

SUBCASE 3:  $\iota_f, \iota_a \dots$  are not all equal but still prefix orderable. The form of  $e$  is similar to the previous subcase, except that each argument array  $v_a$  replaces  $n_f \dots$  with its own particular frame shape. We then have a *lift* redex.  $\square$

**Lemma 4.4.2** (Preservation). *Let  $\Theta, \Delta, \Gamma$  be a well-formed environment, i.e.,  $\Theta; \Delta \vdash \Gamma$ . If  $\Theta; \Delta; \Gamma \vdash e : \tau$  and  $e \mapsto e'$  then  $\Theta; \Delta; \Gamma \vdash e' : \tau$ .*

*Proof.* We use induction on the derivation of  $\cdot; \cdot \vdash e : \tau$ . As with Lemma 4.4.1 (progress), we only consider typing rules that can apply to an expression. We elide T:VAR because a variable does not reduce any further.

**Case EQV**

$$\frac{\Theta; \Delta; \Gamma \vdash e : \tau' \quad \tau \cong \tau'}{\Theta; \Delta; \Gamma \vdash e : \tau} \text{ T:EQV}$$

By the induction hypothesis,  $\Theta; \Delta; \Gamma \vdash e' : \tau'$ . Then applying T:EQV to that derivation produces

$$\frac{\Theta; \Delta; \Gamma \vdash e' : \tau' \quad \tau \cong \tau'}{\Theta; \Delta; \Gamma \vdash e' : \tau} \text{ T:EQV}$$

**Case ARRAY**

$$\frac{\Theta; \Delta; \Gamma \vdash a : \tau_a \dots \quad \Theta; \Delta \vdash \tau_a :: \text{Atom} \quad \text{Length } \llbracket a \dots \rrbracket = \prod n \dots}{\Theta; \Delta; \Gamma \vdash (\text{array } (n \dots) a \dots) : (A \tau_a (\text{shape } n \dots))} \text{ T:ARRAY}$$

An array literal is not itself a redex, so the only way for  $e$  to reduce to  $e'$  is if some atom  $\mathfrak{a}_r$  in the array is reducible. Let  $(\mathfrak{a}_0 \dots \mathfrak{a}_r \mathfrak{a}_1 \dots) = (\mathfrak{a} \dots)$ . The only atom form which may contain an expression that takes a reduction step is a box. So  $\mathfrak{a}_r$  must have the form

$$(\text{box } \iota \dots e_s (\Sigma ((x \gamma) \dots) \tau_s))$$

where  $e_s \mapsto e'_s$  and  $(\Sigma ((x \gamma) \dots) \tau_s) \cong \tau_a$ . The typing derivation for  $\mathfrak{a}_r$  must, except perhaps for use of T:EQV, end with

$$\frac{\Theta \vdash \iota :: \gamma \dots \quad \Theta; \Delta \vdash (\Sigma ((x \gamma) \dots) \tau_s) :: \text{Atom} \quad \Theta; \Delta; \Gamma \vdash e_s : \tau_s[x \mapsto \iota, \dots]}{\Theta; \Delta; \Gamma \vdash (\text{box } \iota \dots e_s (\Sigma ((x \gamma) \dots) \tau_s)) : (\Sigma ((x \gamma) \dots) \tau_s)} \text{ T:BOX}$$

The induction hypothesis implies that  $\Theta; \Delta; \Gamma \vdash e'_s : \tau_s[x \mapsto \iota, \dots]$ , so  $\mathfrak{a}_r$  can be ascribed the same type  $(\Sigma ((x \gamma) \dots) \tau_s)$ , which is equivalent to  $\tau_a$ . Since  $\mathfrak{a}_r \mapsto \mathfrak{a}'_r$ , which still has type  $\tau_a$ , we derive

$$\frac{\Theta; \Delta; \Gamma \vdash \mathfrak{a}_0 : \tau_a \dots \quad \Theta; \Delta; \Gamma \vdash \mathfrak{a}'_r : \tau_a \quad \Theta; \Delta; \Gamma \vdash \mathfrak{a}_1 : \tau_a \dots \quad \Theta; \Delta \vdash \tau_a :: \text{Atom} \quad \text{Length} \llbracket \mathfrak{a} \dots \rrbracket = \prod n \dots}{\Theta; \Delta; \Gamma \vdash (\text{array } (n \dots) \mathfrak{a} \dots) : (A \tau_a (\text{shape } n \dots))} \text{ T:ARRAY}$$

### Case FRAME

$$\frac{\Theta; \Delta; \Gamma \vdash e_a : (A \tau_a \iota_a) \dots \quad \Theta; \Delta \vdash (A \tau_a \iota_a) :: \text{Array} \quad \text{Length} \llbracket e \dots \rrbracket = \prod n_f \dots}{\Theta; \Delta; \Gamma \vdash (\text{frame } (n_f \dots) e_a \dots) : (A \tau_a (++) (\text{shape } n_f \dots) \iota_a)} \text{ T:FRAME}$$

As in the T:ARRAY case, if  $e$  itself is not a redex, then in order for it to reduce to  $e'$ , it must contain some reducible cell, *i.e.*, there is some  $e_r \in e_a \dots$  such that  $e_r \mapsto e'_r$ .

Since the type derivation for  $(\text{frame } (n_f \dots) e_a \dots)$  must ascribe the type  $(A \tau_a \iota_a)$  to each  $e_a$ , including  $e_r$ , the induction hypothesis gives  $\Theta; \Delta; \Gamma \vdash e'_r : (A \tau_a \iota_a)$ . We then patch that result into the type derivation we had. With  $e_a \dots = (e_0 \dots e_r e_1 \dots)$ , we derive

$$\frac{\Theta; \Delta; \Gamma \vdash e_0 : (A \tau_a \iota_a) \dots \quad \Theta; \Delta; \Gamma \vdash e_r : (A \tau_a \iota_a) \quad \Theta; \Delta; \Gamma \vdash e_1 : (A \tau_a \iota_a) \dots \quad \Theta; \Delta \vdash (A \tau_a \iota_a) :: \text{Array} \quad \text{Length} \llbracket e \dots \rrbracket = \prod n_f \dots}{\Theta; \Delta; \Gamma \vdash (\text{frame } (n_f \dots) e_a) : (A \tau_a (++) (\text{shape } n_f \dots) \iota_a)} \text{ T:FRAME}$$

If  $e$  is a redex, it must be a *collapse* redex. Then  $e$  is a frame of array literals, *i.e.*,  $(\text{frame } (n_f \dots) (\text{array } (n_c \dots) \mathfrak{v} \dots) \dots)$ , and it must step

to  $e' = (\text{array } (n_f \dots n_c \dots) \text{Concat}[\![\mathbf{v} \dots]\!] \dots)$ . Each of  $e'$ 's cells is a well-typed array literal, with type derivation:

$$\frac{\Theta; \Delta; \Gamma \vdash \mathbf{v} : \tau_a \dots \quad \Theta; \Delta \vdash \tau_a :: \text{Atom} \quad \text{Length}[\![\mathbf{v} \dots]\!] = \prod n_c \dots}{\Theta; \Delta; \Gamma \vdash (\text{array } (n_c \dots) \mathbf{v} \dots) : (A \tau_a (\text{shape } n_c \dots))} \text{T:ARRAY}$$

The type derivation for each cell in the frame requires that  $\text{Length}[\![\mathbf{v} \dots]\!] = \prod n_c \dots$ . Concatenating  $\prod n_f \dots$  sequences of atomic values each of which contains  $\prod n_c \dots$  elements gives a sequence whose length is  $\prod (n_f \dots n_c \dots)$ . So we derive a similar type for the collapsed array:

$$\frac{\Theta; \Delta; \Gamma \vdash \mathbf{v} : \tau_a \dots \quad \Theta; \Delta \vdash \tau_a :: \text{Atom} \quad \text{Length}[\![\text{Concat}[\![\mathbf{v} \dots]\!] \dots]\!] = \prod (n_f \dots n_c \dots)}{\Theta; \Delta; \Gamma \vdash (\text{array } (n_f \dots n_c \dots) \text{Concat}[\![\mathbf{v} \dots]\!] \dots) : (A \tau_a (\text{shape } n_f \dots n_c \dots))} \text{T:ARRAY}$$

Strictly speaking, our goal is to ascribe the type  $(A \tau_a (++) (\text{shape } n_f \dots) \iota_a)) = (A \tau_a (++) (\text{shape } n_f \dots) (\text{shape } n_c \dots))$  rather than the type we have just derived, but they are equivalent via  $\text{TEQV:ARRAY}$  because  $(++) (\text{shape } n_f \dots) (\text{shape } n_c \dots) \equiv (\text{shape } n_f \dots n_c \dots)$ .  $\text{T:EQV}$  completes the derivation for

$$\Theta; \Delta; \Gamma \vdash e' : (A \tau_a (++) (\text{shape } n_f \dots) \iota_a))$$

### Case TAPP

$$\frac{\Theta; \Delta; \Gamma \vdash e_f : (A (\forall ((x k) \dots) (A \tau_u \iota_u)) \iota_f) \quad \Theta; \Delta \vdash \tau_a :: k \dots}{\Theta; \Delta; \Gamma \vdash (\text{t-app } e_f \tau_a \dots) : (A \tau_u [x \mapsto \tau_a, \dots] (++) \iota_f \iota_u))} \text{T:TAPP}$$

As in previous cases, if  $e_f \mapsto e'_f$ , the induction hypothesis gives us a derivation

$$\Theta; \Delta; \Gamma \vdash e'_f : (A (\forall ((x k) \dots) (A \tau_u \iota_u)) \iota_f)$$

which we can then use for  $(\text{t-app } e'_f \tau_a \dots)$ .

Otherwise,  $e \mapsto e'$  is only possible if we have a  $t\beta$  redex, that is

$$e_f = (\text{array } (n_f \dots) (\text{T}\lambda ((x_u k) \dots) e_u) \dots)$$

and

$$\iota_f = (\text{shape } n_f \dots)$$

Following the  $t\beta$  reduction,  $e' = (\text{frame } (n_f \dots) e_u[x_u \mapsto \tau_a, \dots] \dots)$ . Since  $e_f$  is a well-typed array of type abstractions, it must be the case that



$\Theta; \Delta, x_u :: k \dots; \Gamma \vdash e_u : (A \tau_u \iota_u)$  for each of the abstraction bodies,  $e_u \dots$ . Lemma 4.2.14 (preservation of types under type substitution) implies that we can give  $e_u[x_u \mapsto \tau_a, \dots]$  the “same” type as  $e_u$ :

$$\Theta; \Delta; \Gamma[x_u \mapsto \tau_a, \dots] \vdash e_u[x_u \mapsto \tau_a, \dots] : (A \tau_u[x_u \mapsto \tau_a, \dots] \iota_u)$$

The type derivation for  $e_f$  requires that the individual type abstractions’ types all be equivalent to  $(\forall ((x k) \dots) \tau_f)$  despite possibly being written to bind different type names, so each result cell type  $(A \tau_u[x_u \mapsto \tau_a, \dots] \iota_u)$  is equivalent to  $\tau_f[x \mapsto \tau_a, \dots]$ . Per Barendregt’s convention, an abstraction’s type variables  $x_u \dots$  are not used in the range of  $\Gamma$ , the type environment used in checking that abstraction (otherwise, the environment structure  $\Theta; \Delta; \Gamma$  would be ill-formed), so  $\Gamma[x_u \mapsto \tau_a, \dots] = \Gamma$ . Theorem 4.2.4 (ascription of well-kinded types) implies that the pieces used to form the result type,  $\tau_u[x \mapsto \tau_a, \dots]$  and  $\iota_u$ , are well-formed at kind `Atom` and sort `Shape` respectively, so the cell type  $(A \tau_u[x \mapsto \tau_a, \dots] \iota_u)$  has kind `Array`. Finally, the number of result cells matches the number of atoms in  $e_f$ , which is the product of  $e_f$ ’s dimensions  $n_f \dots$ . Thus we can derive

$$\frac{\begin{array}{l} \Theta; \Delta; \Gamma \vdash e_u[x_u \mapsto \tau_a, \dots] : (A \tau_u[x \mapsto \tau_a, \dots] \iota_u) \dots \\ \Theta; \Delta \vdash (A \tau_u[x \mapsto \tau_a, \dots] \iota_u) :: \text{Array} \\ \text{Length} \llbracket e_u \dots \rrbracket = \prod n \dots \end{array}}{\Theta; \Delta; \Gamma \vdash (\text{frame } (n_f \dots) e_u[x_u \mapsto \tau_a, \dots] \dots) : (A \tau_u[x \mapsto \tau_a, \dots] (++) (\text{shape } n \dots) \iota_u))} \text{T:FRAME}$$

### Case IAPP

$$\frac{\Theta; \Delta; \Gamma \vdash e_f : (A (\prod ((x \gamma) \dots) (A \tau_p \iota_p)) \iota_f)}{\Theta; \Delta; \Gamma \vdash (\text{i-app } e_f \iota_a) : (A \tau_p[x \mapsto \iota_a, \dots] (++) \iota_f \iota_p[x \mapsto \iota_a, \dots]))} \text{T:IAPP}$$

If  $e_f \mapsto e'_f$ , then the induction hypothesis implies that  $\Theta; \Delta; \Gamma \vdash e'_f (A (\prod ((x \gamma) \dots) (A \tau_p \iota_p)) \iota_f)$ . Then the type derivation for  $e'_f$  can be substituted into the original derivation for  $e$ , ascribing the same type to the reduced term  $(\text{i-app } e'_f \iota_a \dots)$ .

Otherwise,  $e$  must be an  $i\beta$  redex, with  $e_f = (\text{array } (n_f \dots) v \dots)$ , each  $v$  of the form  $(\text{I}\lambda ((x' \gamma) \dots) e_u)$ , and  $\vdash \iota_f \equiv (\text{shape } n_f \dots)$ . We must also be able to derive, for each  $v$

$$\Theta; \Delta; \Gamma \vdash v : (\prod ((x \gamma) \dots) (A \tau_p \iota_p))$$

Note that  $x' \dots$  might not be the same variables as  $x \dots$ . If not, then the type derivation for  $v$  must end with `T:EQV` relating the required array type  $(A \tau_p \iota_p)$  and an actually derived array type  $(A \tau'_p \iota'_p)$ . That is, use of `T:EQV` requires deriving both

$$\frac{\Theta, x' :: \gamma \dots; \Delta; \Gamma \vdash e_u : (A \tau'_p \iota'_p)}{\Theta; \Delta; \Gamma \vdash (\text{I}\lambda ((x' \gamma) \dots) e_u) : (\prod ((x' \gamma) \dots) (A \tau'_p \iota'_p))} \text{T:ILAM}$$

and, with free variables  $x_f \dots$ ,

$$\frac{\frac{\tau_p[x \mapsto x_f, \dots] \cong \tau'_p[x' \mapsto x_f, \dots]}{\models \iota_p[x \mapsto x_f, \dots] \equiv \iota'_p[x' \mapsto x_f, \dots]}}{(A \tau_p \iota_p)[x \mapsto x_f, \dots] \cong (A \tau'_p \iota'_p)[x' \mapsto x_f, \dots]} \text{TEQV-ARRAY}$$

$$\frac{(A \tau_p \iota_p)[x \mapsto x_f, \dots] \cong (A \tau'_p \iota'_p)[x' \mapsto x_f, \dots]}{(\prod ((x \gamma) \dots) (A \tau_p \iota_p)) \cong (\prod ((x' \gamma) \dots) (A \tau'_p \iota'_p))} \text{TEQV:PI}$$

We now ascribe a type to  $e_u[x' \mapsto \iota_a, \dots]$ . Previously,  $e_u$  was given the type  $(A \tau'_p \iota'_p)$ , so Lemma 4.2.13 (preservation of types under index substitution) implies

$$\Theta; \Delta; \Gamma[x' \mapsto \iota_a, \dots] \vdash e_u[x' \mapsto \iota_a, \dots] : (A \tau'_p \iota'_p)[x' \mapsto \iota_a, \dots]$$

Well-formedness of the environment means that no free index variables are used in the range of  $\Gamma$ , so  $\Gamma[x' \mapsto \iota_a, \dots] = \Gamma$ . This simplifies our typing result to  $\Theta; \Delta; \Gamma \vdash e_u[x' \mapsto \iota_a, \dots] : (A \tau'_p \iota'_p)[x' \mapsto \iota_a, \dots]$ . The ascribed type is equal to  $(A \tau'_p[x' \mapsto \iota_a, \dots] \iota'_p[x' \mapsto \iota_a, \dots])$ . We must show that it is equivalent to  $(A \tau_p[x \mapsto \iota_a, \dots] \iota_p[x \mapsto \iota_a, \dots])$ .

We consider first the element types  $\tau_p[x \mapsto x_f, \dots][x_f \mapsto \iota_a, \dots]$  and  $\tau'_p[x' \mapsto x_f, \dots][x_f \mapsto \iota_a, \dots]$ , which can be simplified respectively to  $\tau_p[x \mapsto \iota_a, \dots]$  and  $\tau'_p[x' \mapsto \iota_a, \dots]$ . Since index substitution preserves type equivalence (per Lemma 4.2.11), our earlier  $\tau_p[x \mapsto x_f, \dots] \cong \tau'_p[x' \mapsto x_f, \dots]$  implies that  $\tau_p[x \mapsto \iota_a, \dots] \cong \tau'_p[x' \mapsto \iota_a, \dots]$ .

Similarly, preservation of index equality under substitution implies that  $\models \iota_p[x \mapsto \iota_a, \dots] \equiv \iota'_p[x' \mapsto \iota_a, \dots]$ . So we derive the type equivalence

$$\frac{\frac{\tau_p[x \mapsto \iota_a, \dots] \cong \tau'_p[x' \mapsto \iota_a, \dots]}{\models \iota_p[x \mapsto \iota_a, \dots] \equiv \iota'_p[x' \mapsto \iota_a, \dots]}}{(A \tau_p \iota_p)[x \mapsto \iota_a, \dots] \cong (A \tau'_p \iota'_p)[x' \mapsto \iota_a, \dots]} \text{TEQV:ARRAY}$$

which then allows us to derive for each of  $e_u \dots$

$$\frac{\Theta; \Delta; \Gamma \vdash e_u[x' \mapsto \iota_a, \dots] : (A \tau'_p \iota'_p)[x' \mapsto \iota_a, \dots]}{(A \tau'_p \iota'_p)[x' \mapsto \iota_a, \dots] \cong (A \tau_p \iota_p)[x \mapsto \iota_a, \dots]} \text{T:EQV}$$

$$\Theta; \Delta; \Gamma \vdash e_u[x' \mapsto \iota_a, \dots] : (A \tau_p \iota_p)[x \mapsto \iota_a, \dots]$$

This enables a type derivation for the reduction result  $e'$ , which according to  $i\beta$  is  $(\text{frame } (n_f \dots) e_u[x' \mapsto \iota_a, \dots] \dots)$ :

$$\frac{\Theta; \Delta; \Gamma \vdash e_u[x' \mapsto \iota_a, \dots] : (A \tau_p \iota_p)[x \mapsto \iota_a, \dots] \dots}{\text{Length } \llbracket e_u \dots \rrbracket = \prod (n_f \dots)} \text{T:FRAME}$$

$$\Theta; \Delta; \Gamma \vdash e'$$

$$: (A \tau_p[x \mapsto \iota_a, \dots] (++) (\text{shape } n_f \dots) \iota_p[x \mapsto \iota_a, \dots]))$$

### Case UNBOX

$$\frac{\Theta; \Delta; \Gamma \vdash e_s : (A (\Sigma ((x'_i \gamma) \dots) \tau_s) \iota_s)}{\Theta, x_i :: \gamma \dots; \Delta; \Gamma, x_e : \tau_s[x'_i \mapsto x_i, \dots] \vdash e_b : (A \tau_b \iota_b)} \text{T:UNBOX}$$

$$\frac{\Theta; \Delta \vdash (A \tau_b \iota_b) :: \text{Array}}{\Theta; \Delta; \Gamma \vdash (\text{unbox } (x_i \dots x_e e_s) e_b) : (A \tau_b (++) \iota_s \iota_b))}$$

If  $e$  is reducible because  $e_s \mapsto e'_s$ , then the induction hypothesis implies  $\Theta; \Delta; \Gamma \vdash e'_s : (A (\Sigma ((x'_i \gamma) \dots) \tau_s) \iota_s)$ . This can be used to adapt the original type derivation for  $e$  to fit  $e' = (\text{unbox } (x_i \dots x_e e'_s) e_b)$ :

$$\frac{\begin{array}{c} \Theta; \Delta; \Gamma \vdash e'_s : (A (\Sigma ((x'_i \gamma) \dots) \tau_s) \iota_s) \\ \Theta, x_i :: \gamma \dots; \Delta; \Gamma, x_e : \tau_s[x'_i \mapsto x_i, \dots] \vdash e_b : (A \tau_b \iota_b) \\ \Theta; \Delta \vdash (A \tau_b \iota_b) :: \text{Array} \end{array}}{\Theta; \Delta; \Gamma \vdash (\text{unbox } (x_i \dots x_e e'_s) e_b) : (A \tau_b (++) \iota_s \iota_b)} \text{ T:UNBOX}$$

Otherwise,  $e$  must be an *unbox* redex, where all of the following hold:

1.  $e_s$  is of the form  $(\text{array } (n_s \dots) (\text{box } \iota_s \dots v_s \tau_\sigma) \dots)$
2.  $\tau_\sigma = (\Sigma ((x''_i \gamma) \dots) \tau'_s) \cong (\Sigma ((x'_i \gamma) \dots) \tau_s)$
3.  $\iota_s = (\text{shape } n_s \dots)$
4.  $\text{Length}[(\text{box } \iota_s \dots v_s \tau_\sigma) \dots] = \prod n_s \dots$

The type derivation for  $e_s$  must include ascription of  $\tau_\sigma$  to each box within the array:

$$\frac{\begin{array}{c} \Theta \vdash \iota_\sigma :: \gamma \dots \quad \Theta; \Delta \vdash \tau_\sigma :: \text{Atom} \\ \Theta; \Delta; \Gamma \vdash v_\sigma : \tau'_s[x''_i \mapsto \iota_\sigma, \dots] \end{array}}{\Theta; \Delta; \Gamma \vdash (\text{box } \iota_\sigma \dots v_\sigma \tau_\sigma) : \tau_\sigma} \text{ T:BOX}$$

Equivalence of  $\tau_\sigma$  and  $(\Sigma ((x'_i \gamma) \dots) \tau_s)$  means that each box's  $v_\sigma$  can also be typed as  $\tau_s[x'_i \mapsto \iota_\sigma, \dots]$ .

The resulting term  $e'$  is  $e_b[x_i \mapsto \iota_\sigma \dots, x_e \mapsto v_\sigma]$ , which is equal to  $e_b[x_i \mapsto \iota_\sigma, \dots][x_e \mapsto v_\sigma]$ . Since we have a type derivation for  $e_b$  in an extended environment, we will apply Lemmas 4.2.13 and 4.2.15 (preservation of types under index and expression substitution) to produce a type derivation in the original environment.

Using the fact that each of  $\iota_\sigma \dots$  has its required sort (necessary for the previous derivation), Lemma 4.2.13 gives us

$$\begin{array}{c} \Theta; \Delta; (\Gamma, x_e : \tau_s[x'_i \mapsto x_i, \dots])[x_i \mapsto \iota_\sigma, \dots] \vdash \\ e_b[x_i \mapsto \iota_\sigma, \dots] : (\tau_b[x_i \mapsto x'_i, \dots])[x_i \mapsto \iota_\sigma, \dots] \end{array}$$

Well-formedness of the original environment means  $\Gamma$  does not refer to the new index variables  $x_i \dots$ , which are instead bound by the *unbox* expression. So the substituted environment

$$(\Gamma, x_e : \tau_s[x'_i \mapsto x_i, \dots])[x_i \mapsto \iota_\sigma, \dots]$$

is equal to  $\Gamma, x_e : \tau_s[x'_i \mapsto \iota_\sigma, \dots]$ , turning the type derivation into

$$\Theta; \Delta; \Gamma, x_e : \tau_s[x'_i \mapsto \iota_\sigma, \dots] \vdash e_b[x_i \mapsto \iota_\sigma, \dots] : \tau_b[x_i \mapsto x'_i, \dots]$$

The index variables  $x_i \dots$  are not bound in  $\Theta$ , but we still have  $\Theta; \Delta \vdash (A \tau_b \iota_b) :: \text{Array}$ . This means  $\tau_b$  and  $\iota_b$  must be well-formed without those bindings, *i.e.*, none of  $x_i \dots$  appear free in  $\tau_b$  or  $\iota_b$ . Thus  $(A \tau_b \iota_b)[x_i \mapsto \iota_s, \dots]$  is simply  $(A \tau_b \iota_b)$ . Our derivation now concludes  $\Theta; \Delta; \Gamma, x_e : \tau_s[x'_i \mapsto \iota_\sigma, \dots] \vdash e_b[x_i \mapsto \iota_\sigma, \dots] : (A \tau_b \iota_b)$ .

Since  $\Theta; \Delta; \Gamma \vdash v_s : \tau_s[x'_i \mapsto \iota_\sigma, \dots]$ , Lemma 4.2.15 gives a derivation of  $\Theta; \Delta; \Gamma \vdash e_b[x_i \mapsto \iota_\sigma, \dots][x_e \mapsto v_\sigma] : (A \tau_b \iota_b)$  for each box's index contents  $\iota_\sigma \dots$  and term contents  $v_\sigma$ . We can therefore type the frame result of the *unbox* reduction step as follows:

$$\frac{\Theta; \Delta; \Gamma \vdash e_b[x_i \mapsto \iota_\sigma, \dots][x_e \mapsto v_\sigma] : (A \tau_b \iota_b) \dots \quad \Theta; \Delta \vdash (A \tau_b \iota_b) :: \text{Array} \quad \text{Length} \llbracket e_b[x_i \mapsto \iota_\sigma, \dots][x_e \mapsto v_\sigma] \rrbracket = \prod n_s \dots}{\Theta; \Delta; \Gamma \vdash (\text{frame } (n_s \dots) e_b[x_i \mapsto \iota_\sigma, \dots][x_e \mapsto v_\sigma]) : (A \tau_b (++) \iota_b)} \text{T:FRAME}$$

### Case APP

$$\frac{\Theta; \Delta; \Gamma \vdash e_f : (A (-> ((A \tau_i \iota_i) \dots) (A \tau_o \iota_o)) \iota_f) \quad \Theta; \Delta; \Gamma \vdash e_a : (A \tau_i (++) \iota_a \iota_i) \dots \quad \iota_p = \text{Max} \llbracket \iota_f \iota_a \dots \rrbracket}{\Theta; \Delta; \Gamma \vdash (e_f e_a \dots) : (A \tau_o (++) \iota_p \iota_o)} \text{T:APP}$$

If either  $e_f \mapsto e'_f$  or there is some argument  $e_A \in e_a \dots$  such that  $e_A \mapsto e'_A$ , then the induction hypothesis implies that the result  $e'_f$  or  $e'_A$  retains the same type as  $e_f$  or  $e_A$ . The type derivation for  $e$  can be updated with the new sub-derivation for  $e_f$  or  $e_A$  to derive the same type for  $e'$ .

Otherwise,  $e$  must itself be a redex. The possible reductions for an application form are *lift*, *map*,  $\beta$ , and  $\delta$ .

**SUBCASE 1:** If  $e$  is a *lift* redex, then it has the form  $((\text{array } (n_f \dots) v_f \dots) (\text{array } (n_a \dots n_i \dots) v_a \dots) \dots)$ . In order to match the left hand side of the *lift* rule, we must have  $\models \iota_f \equiv (\text{shape } n_f \dots)$  and that for each argument's own type derivation,  $\models \iota_a \equiv (\text{shape } n_a \dots)$ . The frame portion of each array's shape (*i.e.*,  $n_f \dots$  for function position and  $n_a \dots$  for argument position) is replaced with the principal frame,  $n_p \dots$ . The individual atoms used in the new function and argument arrays all come from their corresponding original arrays and therefore retain the same types. That is, the atoms  $v'_f \dots$  in the new function array  $e'_f$ , that is

$$(\text{array } (n_p \dots) \text{Concat} \llbracket \text{Rep}_{n_{fe}} \llbracket \text{Split}_1 \llbracket v_f \dots \rrbracket \rrbracket \rrbracket)$$

all have type  $(-> ((A \tau_i \iota_i) \dots) (A \tau_o \iota_o))$ , and the atoms  $v'_a \dots$  in any new argument array  $e'_a$ , which is

$$(\text{array } (n_p \dots n_i \dots) \text{Concat} \llbracket \text{Rep}_{n_{ae}} \llbracket \text{Split}_{n_{ac}} \llbracket v_a \dots \rrbracket \rrbracket \rrbracket)$$

are all typed as the corresponding  $\tau_i$ . While *Split* breaks a sequence into subsequences (preserving the total number of elements) and *Concat* merges a sequence-of-sequences into a single sequence (also preserving the total number of elements), *Rep<sub>n</sub>* produces  $n$  copies of each element. The factor used by *lift* reduction for replication of each array's cells is the quotient of the number of cells in the principal frame and the number of cells in the original array. Since the original frame shape is a prefix of the principal frame, divisibility is guaranteed. This also ensures that the number of atoms in each new argument array is

$$\begin{aligned} & n_{ae} * \left( \prod n_a \dots \right) * \left( \prod n_i \dots \right) \\ &= \left( \prod n_p \dots \right) * \left( \prod n_i \dots \right) \\ &= \prod (n_p \dots n_i \dots) \end{aligned}$$

Similarly, the number of atoms in the new function array is

$$\begin{aligned} & n_{fe} * \left( \prod n_f \dots \right) \\ &= \prod n_p \dots \end{aligned}$$

Based on the types and quantity of the function and argument arrays' atoms, we can derive

$$\frac{\begin{array}{c} \Theta; \Delta; \Gamma \vdash e'_f \\ : (A (-> ((A \tau_i \iota_i) \dots) (A \tau_o \iota_o)) (\text{shape } n_p \dots)) \\ \Theta; \Delta; \Gamma \vdash e'_a : (A \tau_i (++) (\text{shape } n_p \dots) \iota_i)) \dots \\ \iota_p = \text{Max} \llbracket \iota_p \iota_p \dots \rrbracket \end{array}}{\Theta; \Delta; \Gamma \vdash e' : (A \tau_o (++) \iota_p \iota_o))} \quad \text{T:APP}$$

SUBCASE 2: If  $e$  is a *map* redex, then it has the form

$$((\text{array } (n_f \dots) \mathfrak{a}_f \dots) (\text{array } (n_f \dots n_a \dots) \mathfrak{a}_a \dots) \dots)$$

Matching the left-hand side of the *map* reduction rule requires that  $\models \iota_i \equiv (\text{shape } n_i \dots)$ . Each argument  $e_a = (\text{array } (n_f \dots n_a \dots) \mathfrak{a}_a \dots)$  has its sequence of atoms split into segments whose length is the product of  $n_i \dots$ , the corresponding input type's dimensions. Transposition groups the first segment from each argument, then the second segment from each argument, and so on. So the nested sequence  $((\mathfrak{v}_c \dots) \dots) \dots$  has for each  $j^{\text{th}}$   $((\mathfrak{v}_c \dots) \dots)$  a sequence of atoms corresponding to each of the function input types: the  $(j, k)^{\text{th}}$  atom sequence contains the atoms which make up the  $j^{\text{th}}$  cell of the  $k^{\text{th}}$  argument. Since the length of this single  $\mathfrak{v}_c \dots$  is the product of the corresponding input type's dimensions and they all have the same type as the atoms in the corresponding input type, the array literal built from them,  $(\text{array } (n_i \dots) \mathfrak{v}_c \dots)$  has type  $(A \tau_i (\text{shape } n_i \dots))$ . Each function array  $(\text{array } () \mathfrak{v}_f)$  contains a

single atom taken from the original  $e_f$ , implying that it must have type  $(\rightarrow ((A \tau_i \iota_i) \dots) (A \tau_o \iota_o))$ , which is equivalent to  $(\rightarrow ((A \tau_i (\text{shape } n_i \dots)) \dots) (A \tau_o \iota_o))$ . Since the single-cell argument arrays' types all match the singleton function array's input types with principal frame of  $(\text{shape})$ , each application form itself has type  $(A \tau_o \iota_o)$ . That is, each result-cell application form  $e_c = ((\text{array } () \mathbf{v}_f) (\text{array } (n_i \dots) \mathbf{v}_c \dots) \dots)$  in the resulting frame form can be typed as:

$$\begin{array}{c} \Theta; \Delta; \Gamma \vdash (\text{array } () \mathbf{v}_f) \\ : (A (\rightarrow ((A \tau_i \iota_i) \dots) (A \tau_o \iota_o)) (\text{shape})) \\ \\ \Theta; \Delta; \Gamma \vdash (\text{array } (n_i \dots) \mathbf{v}_c \dots) : (A \tau_i \iota_i) \dots \\ \\ \hline \iota_p = (\text{shape}) \quad \text{T:APP} \\ \Theta; \Delta; \Gamma \vdash e_c : (A \tau_o \iota_o) \end{array}$$

Recall that our goal type in this case is  $\tau = (A \tau_o (++) \iota_p \iota_o)$ , where  $\iota_p = (\text{shape } n_f \dots)$ . That is,  $\tau$  is  $(A \tau_o (++) (\text{shape } n_f \dots) \iota_o)$ . Using the above derivations for the result cells  $e_c \dots$ , we then derive

$$\frac{\Theta; \Delta; \Gamma \vdash e_c : (A \tau_o \iota_o) \dots \quad \Theta; \Delta \vdash \tau :: \text{Array}}{\Theta; \Delta; \Gamma \vdash (\text{frame } (n_f \dots) e_c \dots) : (A \tau_o (++) (\text{shape } n_f \dots) \iota_o)} \quad \text{T:FRAME}$$

**SUBCASE 3:** If  $e$  is a  $\beta$  redex, then  $e_f$  must have the form  $(\text{array } () (\lambda ((x (A \tau_i \iota_i)) \dots) e_0))$ . Then  $e \mapsto_{\beta} e' = e_0[x \mapsto e_a, \dots]$ . We also know for each of  $e_a \dots$  that  $\Theta; \Delta; \Gamma \vdash e_a : (A \tau_i \iota_i)$ . In the type derivation for  $e$  itself, we must have  $\iota_f$  and every  $\iota_a$  equal to  $(\text{shape})$ , so  $\iota_p = (\text{shape})$ . Thus  $(A \tau_o (++) \iota_p \iota_o) \cong (A \tau_o \iota_o)$ . The type of the function atom in  $e_f$  is derived either by **T:LAM**, in which case, the derivation must ascribe  $(A \tau_o \iota_o)$  to  $e_0$ , or by **T:EQV**, in which case there must be some earlier use of **T:LAM** which ascribes some type equivalent type to  $e_0$ . Either way, we have  $\Theta; \Delta; \Gamma, x : (A \tau_i \iota_i) \dots \vdash e_0 : (A \tau_o \iota_o)$ . The substitution lemma (Lemma 4.2.15) then gives  $\Theta; \Delta; \Gamma \vdash e' : (A \tau_o \iota_o)$ .

**SUBCASE 4:** If  $e$  is a  $\delta$  redex, then the required result follows from the requirements for primitive operators and their types.  $\square$

**Lemma 8.1.2** (Environment monotonicity for instantiation). *Let  $\bar{\Gamma}; \Phi \vdash \tau :: k$ . Given one of*

- $\bar{\Gamma}; \Phi \vdash \hat{\mathcal{X}} :: \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbb{C}$
- $\bar{\Gamma}; \Phi \vdash \tau :: \hat{\mathcal{X}} \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbb{C}$

*then  $\bar{\Gamma} \leq \bar{\Gamma}'$ , and  $\Phi \subseteq \Phi'$ .*

*Proof.* We use induction on the instantiation judgment derivation. In most cases, pairs of corresponding subtype and supertype instantiation rules carry equivalent proof obligations and induction hypotheses. The exception is instantiation of polymorphic types.

**Case ILOW:SOLVE, IHIGH:SOLVE, ILOW:REACH, IHIGH:REACH**

These cases are direct applications of the variable-solution rule from the definition of the  $\leq$  relation. There is no change to  $\Phi$ , so  $\Phi = \Phi'$ .

**Case ILOW:ARRAY, IHIGH:ARRAY**

By the induction hypothesis, we have  $\bar{\Gamma} = \bar{\Gamma}_l, \hat{\alpha}, \hat{\sigma}, \hat{\mathcal{X}} \mapsto (A \hat{\alpha} \hat{\sigma}), \bar{\Gamma}_r \leq \bar{\Gamma}_1$  and  $\Phi = \Phi_0 \subseteq \Phi_1$ . The solver specification ensures that  $\bar{\Gamma}_1 \leq \bar{\Gamma}_2 = \bar{\Gamma}'$  and  $\Phi_1 \subseteq \Phi_2 = \Phi'$ . Then transitivity gives  $\bar{\Gamma} \leq \bar{\Gamma}'$  and  $\Phi \subseteq \Phi'$ .

**Case ILOW:FN\*, IHIGH:FN\***

Let  $\bar{\Gamma}_{IH} = \bar{\Gamma}_l, \hat{\mathcal{X}}_i \dots, \hat{\mathcal{X}}_o, \hat{\sigma}, \hat{\mathcal{X}} \mapsto (A (-) (\hat{\mathcal{X}}_i \dots) \hat{\mathcal{X}}_o) \hat{\sigma}), \bar{\Gamma}_r$ . By applying the  $\leq$  rules for adding existential variables and solving an existing existential,  $\bar{\Gamma} \leq \bar{\Gamma}_{IH}$ . The induction hypothesis gives  $\bar{\Gamma}_{IH} \leq \bar{\Gamma}_1 \leq \dots \leq \bar{\Gamma}_{n+1}$  and  $\Phi = \Phi_0 \subseteq \Phi_1 \subseteq \dots \subseteq \Phi_{n+1}$ . The solver specification gives  $\bar{\Gamma}_{n+1} \leq \bar{\Gamma}_{n+2} = \bar{\Gamma}'$  and  $\Phi_{n+1} \subseteq \Phi_{n+2} = \Phi'$ . Then transitivity gives  $\bar{\Gamma} \leq \bar{\Gamma}'$  and  $\Phi \subseteq \Phi'$ .

**Case ILOW:ALL\*, ILOW:PI\***

By the induction hypothesis, we have  $\Phi_0 \subseteq \Phi_1$  and  $\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r, x_a \dots \leq \bar{\Gamma}_1, x_a \dots, \bar{\Gamma}_2$ . Splitting the environments at the universal variables  $x_a \dots$  gives  $\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r \leq \bar{\Gamma}_1$ .

**Case IHIGH:ALL\*, IHIGH:PI\***

Let  $\bar{\Gamma}_{IH} = \bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r, \blacktriangleright_{x_a}, \hat{x}_a \dots$ . The induction hypothesis gives  $\bar{\Gamma}_{IH} \leq \bar{\Gamma}_1, \blacktriangleright_{x_a}, \bar{\Gamma}_2$  and  $\Phi_0 \subseteq \Phi_1$ . Splitting the environments at the scope marker  $\blacktriangleright_{x_a}$ , we have  $\bar{\Gamma} = \bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r \leq \bar{\Gamma}_1 = \bar{\Gamma}'$ .

**Case ILOW:SIGMA\***

Let  $\bar{\Gamma}_{IH} = \bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r, \blacktriangleright_{x_f}, \hat{S}_a \dots$ . By the induction hypothesis, we have  $\bar{\Gamma}_{IH} \leq \bar{\Gamma}_1, \blacktriangleright_{x_f}, \bar{\Gamma}_2$  and  $\Phi_0 \subseteq \Phi_1$ . Splitting environments at  $\blacktriangleright_{x_f}$  gives  $\bar{\Gamma} = \bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r \leq \bar{\Gamma}_1 = \bar{\Gamma}'$ .

**Case IHIGH:SIGMA\***

Let  $\bar{\Gamma}_{IH} = \bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r, S_a \dots$ . The induction hypothesis implies that  $\bar{\Gamma}_{IH} \leq$

$\bar{\Gamma}_1, \mathcal{S}_a \dots, \bar{\Gamma}_2$  and  $\Phi_0 \subseteq \Phi_1$ . By splitting environments at  $\mathcal{S}_a \dots$ , we conclude  $\bar{\Gamma} = \bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r \leq \bar{\Gamma}_1 = \bar{\Gamma}'$ .  $\square$

**Lemma 8.1.3** (Environment monotonicity for subtyping). *If  $\bar{\Gamma}; \Phi \vdash \tau_l \leq \tau_h \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbb{C}$ , then  $\bar{\Gamma} \leq \bar{\Gamma}'$ , and  $\Phi \subseteq \Phi'$ .*

*Proof.* We use induction on the subtyping judgment derivation.

**Case SUB:BASE, SUB:VAR, SUB:EVVAR**

These cases are trivial.  $\bar{\Gamma} = \bar{\Gamma}'$  and  $\Phi = \Phi'$ .

**Case SUB:INSTL, SUB:INSTR**

Monotonicity for instantiation implies  $\bar{\Gamma}_0 \leq \bar{\Gamma}_1$  and  $\Phi_0 \subseteq \Phi_1$ , which is exactly the goal.

**Case SUB:ARRAY**

By the induction hypothesis,  $\bar{\Gamma} = \bar{\Gamma}_0 \leq \bar{\Gamma}_1$  and  $\Phi = \Phi_0 \subseteq \Phi_1$ . The solver specification implies  $\bar{\Gamma}_1 \leq \bar{\Gamma}_2 = \bar{\Gamma}'$  and  $\Phi_1 \subseteq \Phi_2 = \Phi'$ . Transitivity then implies  $\bar{\Gamma} \leq \bar{\Gamma}'$  and  $\Phi \subseteq \Phi'$ .

**Case SUB:FN\***

By the solver specification,  $\bar{\Gamma} = \bar{\Gamma}_0 \leq \bar{\Gamma}_1$  and  $\Phi = \Phi_0 \subseteq \Phi_1$ . The induction hypothesis gives  $\bar{\Gamma}_1 \leq \bar{\Gamma}_2 \leq \dots \leq \bar{\Gamma}_{n+1} = \bar{\Gamma}'$  and  $\Phi_1 \subseteq \Phi_2 \subseteq \dots \subseteq \Phi_{n+1} = \Phi'$ . Transitivity implies  $\bar{\Gamma} \leq \bar{\Gamma}'$  and  $\Phi \subseteq \Phi'$ .

**Case SUB:V\*L, SUB:PI\*L**

By the induction hypothesis,  $\Phi = \Phi_0 \subseteq \Phi_1 = \Phi'$  and  $\bar{\Gamma}_0, \blacktriangleright_{x_f}, \hat{\mathcal{X}} \dots \leq \bar{\Gamma}_1, \blacktriangleright_{x_f}, \bar{\Gamma}_2$ . Splitting this result at  $\blacktriangleright_{x_f}$ , we conclude  $\bar{\Gamma} = \bar{\Gamma}_0 \leq \bar{\Gamma}_1 = \bar{\Gamma}'$ .

**Case SUB:V\*R, SUB:PI\*R**

This case is similar to that for SUB:V\*L: the archive portion of the environment hypothesis is our obligation  $\Phi \subseteq \Phi'$ , and we split the environment portion of the induction hypothesis at the universal variables  $\mathcal{X} \dots$  or  $\mathcal{S} \dots$  to conclude  $\bar{\Gamma} = \bar{\Gamma}_0 \leq \bar{\Gamma}_1 = \bar{\Gamma}'$ .

**Case SUB:SIGMA\*L**

$\bar{\Gamma} = \bar{\Gamma}_0 \leq \bar{\Gamma}_0, \hat{\sigma}_h \leq \bar{\Gamma}_1$  (by the solver specification). The induction hypothesis implies  $\bar{\Gamma}_1, \mathcal{S} \dots \leq \bar{\Gamma}_2, \mathcal{S} \dots, \bar{\Gamma}_3$ . Splitting at  $\mathcal{S} \dots$  gives  $\bar{\Gamma}_1 \leq \bar{\Gamma}_2 = \bar{\Gamma}'$ . We also have  $\Phi = \Phi_0 \subseteq \Phi_1$  (again, by the solver specification) and  $\Phi_1 \subseteq \Phi_2 = \Phi'$  (by the induction hypothesis). So  $\bar{\Gamma} \leq \bar{\Gamma}'$ , and  $\Phi \subseteq \Phi'$ .

**Case SUB:SIGMA\*R**

$\bar{\Gamma} = \bar{\Gamma}_0 \leq \bar{\Gamma}_0, \blacktriangleright_{x_s}, \hat{\mathcal{S}} \dots \leq \bar{\Gamma}_1, \blacktriangleright_{x_s}, \bar{\Gamma}_2$ , by the induction hypothesis. Splitting at  $\blacktriangleright_{x_s}$  implies  $\bar{\Gamma}_0 \leq \bar{\Gamma}_1$ . The induction hypothesis also implies  $\Phi_0 \subseteq \Phi_1$ .

**Case SUB:INST→L, SUB:INST→R**

Let  $\bar{\Gamma}_{IH} = \bar{\Gamma}_l, \hat{\alpha}_i \dots, \hat{\sigma}_i \dots, \hat{\alpha}_o, \hat{\sigma}_o, \hat{\mathcal{X}} \mapsto \tau_f, \bar{\Gamma}_r$ . This only adds new entries  $\hat{\alpha}_i \dots, \hat{\sigma}_i \dots, \hat{\alpha}_o, \hat{\sigma}_o$  and solves the existing entry for  $\hat{\mathcal{X}}$ , so  $\bar{\Gamma} = \bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r \leq \bar{\Gamma}_{IH}$ . By the induction hypothesis,  $\bar{\Gamma}_{IH} \leq \bar{\Gamma}_1 = \bar{\Gamma}'$ , so transitivity gives  $\bar{\Gamma} \leq \bar{\Gamma}'$ . The induction hypothesis also implies  $\Phi = \Phi_0 \subseteq \Phi_1 = \Phi'$ .  $\square$



**Lemma 8.1.4** (Environment monotonicity for bidirectional judgments).

Given one of

- $\bar{\Gamma}; \Phi \vdash \bar{t} \Leftarrow \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow t$
- $\bar{\Gamma}; \Phi \vdash \bar{t} \Rightarrow \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow t$
- $\bar{\Gamma}; \Phi \vdash (\bar{t}_f : \tau_f) \bullet [\bar{t}_a \dots] \Rightarrow \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow t$ ,  
where  $\bar{t} = (\bar{t}_f \ \bar{t}_a \dots)$

then  $\bar{\Gamma} \leq \bar{\Gamma}'$ , and  $\Phi \subseteq \Phi'$ .

*Proof.* We use induction on the bidirectional type derivation.

**Case SYN:ANNOT, CHK:SIGMA**

The induction hypothesis is exactly the goal.

**Case SYN:VAR, APP:FN0**

This case is trivial:  $\bar{\Gamma} = \bar{\Gamma}'$  and  $\Phi = \Phi'$ .

**Case SYN:UNBOX**

By the induction hypothesis,  $\Phi = \Phi_0 \subseteq \Phi_1 \subseteq \Phi_2$  and  $\bar{\Gamma} = \bar{\Gamma}_0 \leq \bar{\Gamma}_1$ ,  $\bar{\Gamma}_1, \widehat{\alpha}_b, \widehat{\sigma}_b, x_i \dots, x_e : \tau_s[x'_i \mapsto x_i, \dots] \leq \bar{\Gamma}_2, x_i \dots, \bar{\Gamma}_3$ . Splitting the second  $\leq$  at  $x_i \dots$ , we have  $\bar{\Gamma}_1, \widehat{\alpha}_b, \widehat{\sigma}_b \leq \bar{\Gamma}_2 = \bar{\Gamma}'$ . The definition of  $\leq$  gives  $\bar{\Gamma}_1 \leq \bar{\Gamma}_1, \widehat{\alpha}_b, \widehat{\sigma}_b$ , since we only add two existential variables. By transitivity,  $\bar{\Gamma} \leq \bar{\Gamma}'$ .

**Case SYN:APP**

Using transitivity and the induction hypothesis,  $\bar{\Gamma} = \bar{\Gamma}_0 \leq \bar{\Gamma}_1 \leq \bar{\Gamma}_2 = \bar{\Gamma}'$  and  $\Phi = \Phi_0 \subseteq \Phi_1 \subseteq \Phi_2 = \Phi'$ .

**Case SYN:FN**

Let  $\bar{\Gamma}_{IH} = \bar{\Gamma}_0, \blacktriangleright_{x_f}, \text{NewVars} \llbracket x, \varsigma \rrbracket \dots, x : \text{ElabType} \llbracket x, \varsigma \rrbracket \dots$ . By the induction hypothesis,  $\bar{\Gamma}_{IH} \leq \bar{\Gamma}_1, \blacktriangleright_{x_f}, \bar{\Gamma}_2$ . Splitting at  $\blacktriangleright_{x_f}$  gives  $\bar{\Gamma} = \bar{\Gamma}_0 \leq \bar{\Gamma}_1 = \bar{\Gamma}'$ . The induction hypothesis also guarantees  $\Phi = \Phi_0 \subseteq \Phi_1 = \Phi'$ .

**Case SYN:ARRAY, SYN:FRAME**

By transitivity and the induction hypothesis,  $\bar{\Gamma} = \bar{\Gamma}_0 \leq \bar{\Gamma}_1 \leq \dots \leq \bar{\Gamma}_m = \bar{\Gamma}'$  and  $\Phi = \Phi_0 \subseteq \Phi_1 \subseteq \dots \subseteq \Phi_m = \Phi'$ .

**Case CHK:SUB**

By transitivity, environment monotonicity for subtyping, and the induction hypothesis,  $\bar{\Gamma} = \bar{\Gamma}_0 \leq \bar{\Gamma}_1 \leq \bar{\Gamma}_2 = \bar{\Gamma}'$  and  $\Phi = \Phi_0 \subseteq \Phi_1 \subseteq \Phi_2 = \Phi'$ .

**Case CHK:FN**

Let  $\bar{\Gamma}_{IH} = \bar{\Gamma}_0, \blacktriangleright_{x_f}, \text{NewVars} \llbracket x, \varsigma \rrbracket \dots, x : \text{ElabType} \llbracket x, \varsigma \rrbracket \dots$ . Transitivity and environment monotonicity for subtyping imply  $\bar{\Gamma}_{IH} \leq \bar{\Gamma}_1 \leq \dots \leq \bar{\Gamma}_n$  and  $\Phi = \Phi_0 \subseteq \Phi_1 \subseteq \dots \subseteq \Phi_n$ . By the induction hypothesis,  $\Phi_n \subseteq \Phi_{n+1} = \Phi'$  and  $\bar{\Gamma}_n \leq \bar{\Gamma}_{n+1}, \blacktriangleright_{x_f}, \bar{\Gamma}_{n+2}$ . Transitivity then implies  $\Phi \subseteq \Phi'$  and  $\bar{\Gamma}_{IH} \leq \bar{\Gamma}_{n+1}, \blacktriangleright_{x_f}, \bar{\Gamma}_{n+2}$ . By splitting at  $\blacktriangleright_{x_f}$ , we conclude  $\bar{\Gamma} = \bar{\Gamma}_0 \leq \bar{\Gamma}_{n+1} = \bar{\Gamma}'$ .

**Case CHK:PI, CHK:ALL**

The induction hypothesis gives  $\Phi = \Phi_0 \subseteq \Phi_1 = \Phi'$  and  $\bar{\Gamma}_0, x \dots \leq \bar{\Gamma}_1 x \dots, \bar{\Gamma}_2$ . Splitting at  $x \dots$  implies  $\bar{\Gamma} = \bar{\Gamma}_0 \leq \bar{\Gamma}_1 = \bar{\Gamma}'$ .

**Case CHK:ARRAY, CHK:FRAME**

Transitivity, the solver specification, and the induction hypothesis imply  $\bar{\Gamma} = \bar{\Gamma}_0 \leq \bar{\Gamma}_1 \leq \dots \leq \bar{\Gamma}_n \leq \bar{\Gamma}_{n+1} = \bar{\Gamma}'$  and  $\Phi = \Phi_0 \subseteq \Phi_1 \subseteq \dots \subseteq \Phi_n \subseteq \Phi_{n+1} = \Phi'$ .

**Case APP:ALL, APP:PI**

By the induction hypothesis,  $\bar{\Gamma}_a, \hat{x} \dots \leq \bar{\Gamma}_1 = \bar{\Gamma}'$  and  $\Phi = \Phi_0 \subseteq \Phi_1 = \Phi'$ . From the definition of  $\leq$ , introducing the new existentials means  $\bar{\Gamma} = \bar{\Gamma}_0 \leq \bar{\Gamma}_a, \hat{x} \dots$ . Then transitivity implies  $\bar{\Gamma} \leq \bar{\Gamma}'$ .

**Case APP:FN\*F, APP:FN\*A**

Transitivity, the solver specification, and the induction hypothesis imply  $\bar{\Gamma}_0, \widehat{\sigma}_F, \widehat{\sigma}_E \leq \bar{\Gamma}_1 \leq \bar{\Gamma}_2 \leq \bar{\Gamma}_3 = \bar{\Gamma}'$  and  $\Phi = \Phi_0 \subseteq \Phi_1 \subseteq \Phi_2 \subseteq \Phi_3 = \Phi'$ . The definition of  $\leq$  gives  $\bar{\Gamma} = \bar{\Gamma}_0 \leq \bar{\Gamma}_0, \widehat{\sigma}_F, \widehat{\sigma}_E$ . Finally, transitivity implies  $\bar{\Gamma} \leq \bar{\Gamma}'$ .

**Case APP:FN\*C**

By transitivity and the induction hypothesis,  $\bar{\Gamma} = \bar{\Gamma}_0 \leq \bar{\Gamma}_1 \leq \bar{\Gamma}_2 = \bar{\Gamma}'$ .  $\square$

**Lemma 8.1.5** (Variable scoping for instantiation). *Given one of*

- $\bar{\Gamma}; \Phi \vdash \mathcal{X} : \leq \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbb{C}$
- $\bar{\Gamma}; \Phi \vdash \tau \leq : \mathcal{X} \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbb{C}$

*then all of the following hold:*

- $TB \llbracket \bar{\Gamma} \rrbracket = TB \llbracket \bar{\Gamma}' \rrbracket$
- $KB \llbracket \bar{\Gamma} \rrbracket = KB \llbracket \bar{\Gamma}' \rrbracket$
- $SB \llbracket \bar{\Gamma} \rrbracket = SB \llbracket \bar{\Gamma}' \rrbracket$

*Proof.* We use induction on the instantiation judgment derivation.

**Case ILOW:SOLVE, IHIGH:SOLVE**

The only altered entry is that for  $\widehat{\mathcal{X}}$ . No new variables are introduced.

**Case ILOW:SOLVE, IHIGH:SOLVE**

The only altered entry is that for  $\widehat{\mathcal{X}}_1$ . No new variables are introduced.

**Case ILOW:ARRAY, IHIGH:ARRAY**

The induction hypothesis implies that all term and universal variables in  $\bar{\Gamma}_1$  also appear in  $\bar{\Gamma}_l, \hat{\alpha}, \hat{\sigma}, \hat{\mathcal{X}} \mapsto (A \hat{\alpha} \hat{\sigma}), \bar{\Gamma}_r$ . Those entries can only be in  $\bar{\Gamma}_l$  or  $\bar{\Gamma}_r$ , so they must appear in  $\bar{\Gamma} = \bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r$ . Since the solver, according to specification, does not introduce new variables, nothing is added from  $\bar{\Gamma}_1$  to  $\bar{\Gamma}_2 = \bar{\Gamma}'$ . So the set of variables appearing in  $\bar{\Gamma}$  is the same as the set appearing in  $\bar{\Gamma}'$ .

**Case ILOW:FN\*, IHIGH:FN\***

Similar to the previous case, the induction hypothesis, along with transitivity, implies that  $\bar{\Gamma}_l, \widehat{\mathcal{X}}_i \dots, \widehat{\mathcal{X}}_o, \hat{\sigma}, \hat{\mathcal{X}} \mapsto (A (\dashv (\widehat{\mathcal{X}}_i \dots) \widehat{\mathcal{X}}_o) \hat{\sigma}), \bar{\Gamma}_r$ , the

input environment of the first premise includes the same term and universal variables as  $\bar{\Gamma}_{n+1}$ , and the solver adds no new variables in constructing  $\bar{\Gamma}_{n+2} = \bar{\Gamma}'$ . Any term or universal variable in the first premise's input environment must appear in  $\bar{\Gamma}_l$  or  $\bar{\Gamma}_r$ , so it also appears in  $\bar{\Gamma} = \bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r$ .

**Case ILOW:ALL\*, ILOW:PI\***

According to the induction hypothesis, the premise's output environment  $\bar{\Gamma}_1, x_a \dots, \bar{\Gamma}_2$  introduces no new bindings over the premise's input environment  $\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r, x_a \dots$ . The universal variables  $x_a \dots$  are excluded from the conclusion's output environment, so the output environment  $\bar{\Gamma}_1$  cannot add bindings not present in  $\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r$ .

**Case IHIGH:ALL\*, IHIGH:PI\***

The induction hypothesis guarantees no new variables appear going from  $\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r, \blacktriangleright_{x_a}, \hat{x}_a \dots$  to  $\bar{\Gamma}_1, \blacktriangleright_{x_a}, \bar{\Gamma}_2$ . Since no variables were added to  $\bar{\Gamma} = \bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r$  to construct the premise's input environment, we know that the premise's output environment also contains no variables not present in  $\bar{\Gamma}$ . Since  $\bar{\Gamma}' = \bar{\Gamma}_1$  has a subset of the entries of  $\bar{\Gamma}_1, \blacktriangleright_{x_a}, \bar{\Gamma}_2$ , it also cannot contain any new term or universal variables.

**Case ILOW:SIGMA\***

From the induction hypothesis, we know that all variable bindings in the premise's output environment  $\bar{\Gamma}_1, \blacktriangleright_{x_f}, \bar{\Gamma}_2$  also appear in its input environment  $\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r, \blacktriangleright_{x_f}, \hat{\mathcal{S}}_a \dots$ . Since the input environment also introduces no new term- or universal-variable bindings beyond those in the conclusion's input environment  $\bar{\Gamma}$ , the output environment  $\bar{\Gamma}'$  must also contain no new bindings.

**Case IHIGH:SIGMA\***

The induction hypothesis implies that the output environment of the second premise,  $\bar{\Gamma}_1, \mathcal{S}_a \dots, \bar{\Gamma}_2$ , binds the same term variables and universal variables as the input environment,  $\bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r, \mathcal{S}_a \dots$ . The only variable bindings these environments have which are not found in  $\bar{\Gamma} = \bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r$  are the index variables  $\mathcal{S}_a \dots$ , which are also excluded from the output environment  $\bar{\Gamma}' = \bar{\Gamma}_1$ .  $\square$

**Lemma 8.1.6** (Variable scoping for subtyping). *If  $\bar{\Gamma}; \Phi \vdash \tau_l \leq \tau_h \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbf{C}$ , then all of the following hold:*

- $EVars \llbracket \bar{\Gamma} \rrbracket = EVars \llbracket \bar{\Gamma}' \rrbracket$
- $TVars \llbracket \bar{\Gamma} \rrbracket = TVars \llbracket \bar{\Gamma}' \rrbracket$
- $IVars \llbracket \bar{\Gamma} \rrbracket = IVars \llbracket \bar{\Gamma}' \rrbracket$

*Proof.* We use induction on the subtyping judgment derivation.

**Case SUB:BASE, SUB:VAR, SUB:EXVAR**

These cases are trivial: the input and output environments are equal.

**Case SUB:INSTL, SUB:INSTR**

This follows from the variable scoping lemma for instantiation.

**Case SUB:ARRAY**

The induction hypothesis implies that both  $\bar{\Gamma}_1$  and  $\bar{\Gamma} = \bar{\Gamma}_0$  bind the same term and universal variables. The solver specification gives the same guarantee about  $\bar{\Gamma}_1$  and  $\bar{\Gamma}_2 = \bar{\Gamma}'$ .

**Case SUB:FN\***

From the solver specification, we have  $\bar{\Gamma} = \bar{\Gamma}_0$  and  $\bar{\Gamma}_1$  binding the same term and universal variables. The induction hypothesis implies the same for  $\bar{\Gamma}_2 \dots \bar{\Gamma}_{n+1} = \bar{\Gamma}'$ .

**Case SUB:ALL\*L, SUB:PI\*L**

From  $\bar{\Gamma} = \bar{\Gamma}_0$  to the first premise's input environment  $\bar{\Gamma}_0, \blacktriangleright_{x_f}, \hat{\mathcal{X}} \dots$  (alternatively,  $\bar{\Gamma}_0, \blacktriangleright_{x_f}, \hat{\mathcal{S}} \dots$ ), no term or universal variables are introduced. The induction hypothesis implies none are added in constructing the corresponding output environment  $\bar{\Gamma}_1, \blacktriangleright_{x_f}, \bar{\Gamma}_2$ . Then  $\bar{\Gamma}' = \bar{\Gamma}_1$  can only contain a subset of those variables.

**Case SUB:ALL\*R**

By the induction hypothesis, no variables are added to  $\bar{\Gamma}_0, \mathcal{X} \dots$  (or  $\bar{\Gamma}_0, \mathcal{S} \dots$ ) to construct the premise's output environment  $\bar{\Gamma}_1, \mathcal{X} \dots, \bar{\Gamma}_2$  (or  $\bar{\Gamma}_1, \mathcal{S} \dots, \bar{\Gamma}_2$ ). The only variables in that output environment which do not appear in the original environment  $\bar{\Gamma}$  are the universal variables ( $\mathcal{X} \dots$  or  $\mathcal{S} \dots$ ), which are excluded from  $\bar{\Gamma}' = \bar{\Gamma}_1$ .

**Case SUB:SIGMA\*L**

The only variable bindings added in the premises are the index variables  $\mathcal{S} \dots$ , according to the solver specification and induction hypothesis. All of  $\mathcal{S} \dots$  appear between  $\bar{\Gamma}_2$  and  $\bar{\Gamma}_3$  in the premise's output environment, so the conclusion's output environment  $\bar{\Gamma}' = \bar{\Gamma}_2$  contains the same term- and universal-variable bindings as  $\bar{\Gamma} = \bar{\Gamma}_0$ .

**Case SUB:SIGMA\*R**

The premise's input environment,  $\bar{\Gamma}_0, \blacktriangleright_{x_f}, \mathcal{S} \dots$ , binds the same variables as  $\bar{\Gamma} = \bar{\Gamma}_0$ . By the induction hypothesis, no new term, type, or index variables are added in constructing  $\bar{\Gamma}_1, \blacktriangleright_{x_f}, \bar{\Gamma}_2$ . So the conclusion's output environment  $\bar{\Gamma}' = \bar{\Gamma}_1$ , which is a prefix of the premise's output environment, cannot contain new variable bindings.

**Case SUB:INST→L, SUB:INST→R**

From the original input environment  $\bar{\Gamma} = \bar{\Gamma}_l, \hat{\mathcal{X}}, \bar{\Gamma}_r$ , we only add existential variables to produce the premise's input environment. The induction hypothesis implies that no term variables or universal type or index variables are added to produce the premise's output environment  $\bar{\Gamma}_1$  which is the same as the conclusion's output environment  $\bar{\Gamma}'$ .  $\square$

**Lemma 8.1.7** (Variable scoping for bidirectional judgments). *Given one of*

- $\bar{\Gamma}; \Phi \vdash \bar{t} \Leftarrow \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow t$
- $\bar{\Gamma}; \Phi \vdash \bar{t} \Rightarrow \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow t$
- $\bar{\Gamma}; \Phi \vdash (\bar{t}_f : \tau_f) \bullet [\bar{t}_a \dots] \Rightarrow \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow t$

then all of the following hold:

- $EVars \llbracket \bar{\Gamma} \rrbracket = EVars \llbracket \bar{\Gamma}' \rrbracket$
- $TVars \llbracket \bar{\Gamma} \rrbracket = TVars \llbracket \bar{\Gamma}' \rrbracket$
- $IVars \llbracket \bar{\Gamma} \rrbracket = IVars \llbracket \bar{\Gamma}' \rrbracket$

*Proof.* We use induction on the bidirectional typing derivation.

**Case SYN:ANNOT, CHK:SIGMA**

The induction hypothesis is exactly the goal.

**Case SYN:VAR, APP:FN0**

This case is trivial:  $\bar{\Gamma} = \bar{\Gamma}'$ .

**Case SYN:UNBOX**

By the induction hypothesis, going from  $\bar{\Gamma} = \bar{\Gamma}_0$  to  $\bar{\Gamma}_1$  does not add new variables, nor does going from  $\bar{\Gamma}_1, \widehat{\alpha}_b, \widehat{\sigma}_b, x_i \dots, x_e : \tau_s[x'_i \mapsto x_i, \dots]$  to  $\bar{\Gamma}_2, x_i \dots, \bar{\Gamma}_3$ . So the only variables added going from  $\bar{\Gamma}$  to  $\bar{\Gamma}_2, x_i \dots, \bar{\Gamma}_3$  are  $x_i \dots$  and  $x_e$ , which are excluded from the output environment  $\bar{\Gamma}_2 = \bar{\Gamma}'$ .

**Case SYN:APP**

The induction hypothesis and transitivity imply that  $\bar{\Gamma} = \bar{\Gamma}_0$ ,  $\bar{\Gamma}_1$ , and  $\bar{\Gamma}_2 = \bar{\Gamma}'$  all contain the same term and universal type and index variables.

**Case SYN:FN**

By the induction hypothesis, the premise does not introduce any new variables in its output environment. Since all variables added to  $\bar{\Gamma} = \bar{\Gamma}_0$  to construct the premise's input environment appear to the right of  $\blacktriangleright_{x_f}$ , they are all part of the  $\bar{\Gamma}_2$  component which is excluded from the conclusion's output environment  $\bar{\Gamma}' = \bar{\Gamma}_1$ .

**Case SYN:ARRAY, SYN:FRAME**

By the induction hypothesis and transitivity of equality, the sets of variables included in the premises input and output environments— $\bar{\Gamma} = \bar{\Gamma}_0, \bar{\Gamma}_1, \dots, \bar{\Gamma}_m = \bar{\Gamma}'$ —are all the same.

**Case CHK:SUB**

The induction hypothesis ensures that  $\bar{\Gamma} = \bar{\Gamma}_0$  and  $\bar{\Gamma}_1$  bind the same term and universal variables. The variable scoping lemma for subtyping does the same for  $\bar{\Gamma}_1$  and  $\bar{\Gamma}_2 = \bar{\Gamma}'$ .

**Case CHK:FN**

No term variables or universal variables are added to  $\bar{\Gamma} = \bar{\Gamma}_0$  to construct the first premise's input environment. By the scoping lemma for subtyping, the first  $n$  premises also introduce none in constructing their output environments  $\bar{\Gamma}_1, \dots, \bar{\Gamma}_n$ . The induction hypothesis then implies none are introduced in  $\bar{\Gamma}_{n+1}, \blacktriangleright_{x_f}, \bar{\Gamma}_{n+2}$ . Since  $\bar{\Gamma}' = \bar{\Gamma}_{n+1}$  can contain only a subset of that output environment's variable bindings,  $\bar{\Gamma}$  and  $\bar{\Gamma}'$  must contain the same sets of bindings.

**Case CHK:PI, CHK:ALL**

By the induction hypothesis, no new variables are introduced by the

premise going from  $\bar{\Gamma}_0, x \dots$  to  $\bar{\Gamma}_1, x \dots, \bar{\Gamma}_2$ . The only new variables involved which are not part of  $\overline{env} = \overline{env}_0$  are the universal variables  $x \dots$ , and they are excluded from the output environment  $\bar{\Gamma}' = \bar{\Gamma}_1$ .

**Case CHK:ARRAY, CHK:FRAME**

The induction hypothesis implies that the first  $m$  output environments,  $\bar{\Gamma}_1, \dots, \bar{\Gamma}_m$ , all have the same set of bindings as  $\bar{\Gamma} = \bar{\Gamma}_0$ . Since the solver does not introduce new term or universal variable bindings,  $\bar{\Gamma}' = \bar{\Gamma}_{m+1}$  also has the same set.

**Case APP:ALL, APP:PI**

The premise's input environment  $\bar{\Gamma}_a, \hat{x} \dots$  contains the same term and universal variable bindings as  $\bar{\Gamma} = \bar{\Gamma}_0$ . By the induction hypothesis, this is the same set of bindings as  $\bar{\Gamma}_1 = \bar{\Gamma}$ .

**Case APP:FN\*F, APP:FN\*A**

By the induction hypothesis, solver specification, and transitivity, all premises' input and output environments contain the same sets of term variables and universal type and index variables. This includes the conclusion's output environment  $\bar{\Gamma}' = \bar{\Gamma}_3$ . The first premise's input environment  $\bar{\Gamma}_0, \widehat{\sigma}_F, \widehat{\sigma}_E$  also has the same set of bindings as the conclusion's input environment  $\bar{\Gamma} = \bar{\Gamma}_0$ .

**Case APP:FN\*C**

By transitivity and the induction hypothesis, all of  $\bar{\Gamma} = \bar{\Gamma}_0, \bar{\Gamma}_1$ , and  $\bar{\Gamma}_2 = \bar{\Gamma}'$  contain the same term variable and universal variable bindings.  $\square$

**Theorem 8.1.1** (Instantiation coercion). *Given all of*

- $\bar{\Gamma}; \Phi \vdash \hat{\mathcal{X}} : \leq \tau \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbf{C}_t$
- $KB \llbracket \bar{\Gamma} \rrbracket; SB \llbracket \bar{\Gamma} \rrbracket \vdash \tau :: k$
- $\bar{\Gamma}' \succeq \bar{\Gamma}''$
- $SAT \llbracket \Phi' \rrbracket$

*then  $\mathbf{C}_t$  coerces from  $\bar{\Gamma}'' \llbracket \hat{\mathcal{X}} \rrbracket$  to  $\bar{\Gamma}'' \llbracket \tau \rrbracket$  and given all of*

- $\bar{\Gamma}; \Phi \vdash \tau \leq: \hat{\mathcal{X}} \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbf{C}_t$
- $KB \llbracket \bar{\Gamma} \rrbracket; SB \llbracket \bar{\Gamma} \rrbracket \vdash \tau :: k$
- $\bar{\Gamma}' \succeq \bar{\Gamma}''$
- $SAT \llbracket \Phi' \rrbracket$

*then  $\mathbf{C}_t$  coerces from  $\bar{\Gamma}'' \llbracket \tau \rrbracket$  to  $\bar{\Gamma}'' \llbracket \hat{\mathcal{X}} \rrbracket$ .*

*Proof.* We use induction on the instantiation derivation.

**Case ILOW:SOLVE, IHIGH:SOLVE, ILOW:REACH, IHIGH:REACH**

$\bar{\Gamma}'' \llbracket \tau \rrbracket = \bar{\Gamma}'' \llbracket \hat{\mathcal{X}} \rrbracket$ , so the coercion must have the same input and output types.  $\mathbf{C} = \square$  is the identity coercion.

**Case ILOW:ARRAY, IHIGH:ARRAY**

By the induction hypothesis,  $\mathbb{A}$  coerces from  $\hat{a}$  to  $\Upsilon$  (or from  $\Upsilon$  to  $\hat{a}$ ) under  $\bar{\Gamma}_1$ . The solver specification implies that  $\bar{\Gamma}_2$  must also contain the variable bindings present in  $\bar{\Gamma}_1$ . The existential variable solutions in  $\bar{\Gamma}''$  (which must be retained from earlier environments, according to the solver specification) ensure that  $\bar{\Gamma} \llbracket \hat{\mathcal{X}} \rrbracket = \bar{\Gamma} \llbracket (A \hat{a} \hat{\sigma}) \rrbracket = \bar{\Gamma} \llbracket (A \hat{a} I) \rrbracket$ . Then the Lift coercion lemma implies  $\mathbb{E}_t = \text{Lift} C_{\hat{a}} \llbracket \mathbb{A} \rrbracket$  coerces from  $\hat{\mathcal{X}}$  to  $(A \Upsilon I)$  (or from  $(A \Upsilon I)$  to  $\hat{\mathcal{X}}$ ) under  $\bar{\Gamma}''$ .

**Case ILOW:FN\*, IHIGH:FN\***

Monotonicity and the solver specification imply that  $\bar{\Gamma}'' = \bar{\Gamma}_{n+2}$  contains the bindings necessary for coercing using the contexts generated by the premises,  $\mathbb{E}_i \dots$  and  $\mathbb{E}_o$ . The induction hypothesis then implies that each  $\mathbb{E}_i$  coerces from the corresponding  $T_i$  to  $\hat{\mathcal{X}}_i$  (or from  $\hat{\mathcal{X}}_i$  to  $T_i$ ) and  $\mathbb{E}_o$  from  $\hat{\mathcal{X}}_o$  to  $T_o$  (or from  $T_o$  to  $\hat{\mathcal{X}}_o$ ) under  $\bar{\Gamma}''$ . Since  $\bar{\Gamma}''$  must retain the existential variable solutions which arose in the premises,  $\bar{\Gamma}'' \llbracket \hat{\mathcal{X}} \rrbracket = \bar{\Gamma}'' \llbracket (A (-> (\hat{\mathcal{X}}_i \dots) \hat{\mathcal{X}}_o) \hat{\sigma}) \rrbracket = \bar{\Gamma}'' \llbracket (A (-> (\hat{\mathcal{X}}_i \dots) \hat{\mathcal{X}}_o) I) \rrbracket$ .

By the Function coercion lemma,  $\mathbb{E}_t$  therefore coerces from  $\bar{\Gamma}'' \llbracket \hat{\mathcal{X}} \rrbracket$  to  $(A (-> (T_i \dots) T_o) I)$  (or coerces from  $(A (-> (T_i \dots) T_o) I)$  to  $\bar{\Gamma}'' \llbracket \hat{\mathcal{X}} \rrbracket$ ) under  $\bar{\Gamma}''$ .

**Case ILOW:ALL\*, ILOW:PI\***

By the induction hypothesis,  $\mathbb{E}$  coerces from  $\hat{\mathcal{X}}$  to  $(A \Upsilon I_c)$  under  $\bar{\Gamma}_1$ . The type-abstraction (or index-abstraction) code used to build the Each context coerces from  $(A \Upsilon I_c)$  to  $(A (\forall (x_a \dots) (A \Upsilon I_c)) (\text{shape}))$  (or to  $(A (\prod (x_a \dots) (A \Upsilon I_c)) (\text{shape}))$ ). So the Each coercion itself coerces from  $(A \Upsilon (++) I_f I_c)$  to  $(A (\forall (x_a \dots) (A \Upsilon I_c)) I_f)$  (or to  $(A (\prod (x_a \dots) (A \Upsilon I_c)) I_f)$ ). Composing the premise's resulting coercion with the Each coercion thus produces the required result.

**Case IHIGH:V\*, IHIGH:PI\***

The context  $(\text{t-app } \square \bar{\Gamma}_2 \llbracket \hat{x}_a \rrbracket \dots)$  (or  $(\text{i-app } \square \bar{\Gamma}_2 \llbracket \hat{x}_a \rrbracket \dots)$ ) coerces from the polymorphic type  $(A (\forall (x_a \dots) (A \Upsilon I_c)) I_f)$  (or from  $(A (\prod (x_a \dots) (A \Upsilon I_c)) I_f)$ ) to the goal type

$$\begin{aligned} & (A \Upsilon [x_a \mapsto \bar{\Gamma}_2 \llbracket \hat{x}_a \rrbracket, \dots] (++) I_f I_c) \\ & = (A \Upsilon (++) I_f I_c) [x_a \mapsto \bar{\Gamma}_2 \llbracket \hat{x}_a \rrbracket, \dots] \end{aligned}$$

under  $\bar{\Gamma}'' = \bar{\Gamma}_1$ . Note that any existential variables solved in  $\bar{\Gamma}_2$  are replaced with their solutions, so only  $\bar{\Gamma}_1$  is needed. By the induction hypothesis,  $\mathbb{E}$  coerces from  $(A \Upsilon (++) I_f I_c) [x_a \mapsto \hat{x}_a, \dots]$  to  $\hat{x}$  under  $\bar{\Gamma}_1, \blacktriangleright_{x_f}, \bar{\Gamma}_2$ , so it coerces from  $(A \Upsilon (++) I_f I_c) [x_a \mapsto \bar{\Gamma}_2 \llbracket \hat{x}_a \rrbracket, \dots]$  to  $\hat{x}$  under  $\bar{\Gamma}_1$ . Composing these contexts thus coerces from  $(A (\forall (x_a \dots) (A \Upsilon I_c)) I_f)$  (or  $(A (\prod (x_a \dots) (A \Upsilon I_c)) I_f)$ ) to  $\hat{x}$  under  $\bar{\Gamma}_1$ .

**Case ILOW:SIGMA\***

As in the previous case, any existential variables solved in  $\bar{\Gamma}_2$  are replaced with their solutions by substitution in the context generated by the conclusion. So  $\bar{\Gamma}''$ , which extends  $\bar{\Gamma}_1$  is only used to resolve variables not resolved by  $\bar{\Gamma}_2$ . In order to show that the Each context coerces from the intermediate type

$$(\text{A } \bar{\Gamma} \text{ } (++) I_f I_c) [\mathcal{S} \mapsto \mathcal{S}_a, \dots]$$

to the goal type

$$(\text{A } (\Sigma (\mathcal{S}_a \dots)) (\text{A } \bar{\Gamma} I_c)) I_f)$$

we must ensure that we have the correct witnesses when constructing the `box`. That is, when substituted for the  $\Sigma$ -bound variables in  $(\text{A } \bar{\Gamma} I_c)$ , they must produce  $\bar{\Gamma}_2 \llbracket (\text{A } \bar{\Gamma} I_c) \rrbracket$ , the type of the `box` contents. This requirement is upheld because the cell type used by `EachC` each occurrence of a  $\Sigma$ -bound variable  $\mathcal{S}_a$  replaced by  $\widehat{\mathcal{S}}_a$ .

The induction hypothesis implies that the premise's output context  $\mathbb{E}$  coerces from  $\bar{\Gamma}'' \llbracket \mathcal{X} \rrbracket$  to the intermediate type. So composition of coercions implies that the conclusion's resulting context coerces from  $\bar{\Gamma}'' \llbracket \mathcal{X} \rrbracket$  to the goal type.

**Case IHIGH:SIGMA\***

The first premise ensures that the body of the dependent sum is a well-formed type even without the  $\Sigma$ -bound index variables. So the `unbox` context wrapped around a term of the source type is itself typable as  $(\text{A } \bar{\Gamma} \text{ } (++) I_f I_c)$ . The induction hypothesis implies that the premise produces a context which coerces from that intermediate type to the goal type  $\bar{\Gamma}'' \llbracket \hat{\mathcal{X}} \rrbracket$ . So their composition coerces from the source type

$$(\text{A } (\Sigma (\mathcal{S}_a \dots)) (\text{A } \bar{\Gamma} I_c)) I_f)$$

to the goal type  $\bar{\Gamma}'' \llbracket \hat{\mathcal{X}} \rrbracket$ . □

**Theorem 8.1.2** (Subtyping coercion). *Given all of*

- $KB \llbracket \bar{\Gamma} \rrbracket ; SB \llbracket \bar{\Gamma} \rrbracket \vdash \tau_l :: k$
- $KB \llbracket \bar{\Gamma} \rrbracket ; SB \llbracket \bar{\Gamma} \rrbracket \vdash \tau_h :: k$
- $\bar{\Gamma}; \Phi \vdash \tau_l \leq \tau_h \dashv \bar{\Gamma}'; \Phi' \hookrightarrow \mathbf{C}_t$
- $\bar{\Gamma}' \leq \bar{\Gamma}''$
- $SAT \llbracket \Phi' \rrbracket$

then  $\mathbf{C}_t$  coerces from  $\bar{\Gamma}'' \llbracket \tau_l \rrbracket$  to  $\bar{\Gamma}'' \llbracket \tau_h \rrbracket$ .



*Proof.* We use induction on the subtyping derivation.

**Case SUB:BASE, SUB:VAR, SUB:EXVAR**

We have  $\tau_l = \tau_h$ , so the identity coercion  $\square$  suffices.

**Case SUB:INSTL, SUB:INSTR**

The goal is a direct result of the coercion theorem for instantiation.

**Case SUB:ARRAY**

The induction hypothesis implies that  $\mathbf{C}$  coerces from  $\Upsilon_l$  to  $\Upsilon_h$  under  $\bar{\Gamma}_1$ . By the solver specification, preexisting variable bindings and solutions are preserved in  $\bar{\Gamma}_2 = \bar{\Gamma}''$ , so the coercion is preserved as well. The solver also ensures that  $\bar{\Gamma}'' \llbracket I_l \rrbracket = \bar{\Gamma}'' \llbracket I_h \rrbracket$ . Then the Lift lemma implies that  $\mathbf{C}_t$  coerces from  $(A \Upsilon_l I_l)$  to  $(A \Upsilon_h I_h)$  under  $\bar{\Gamma}''$ .

**Case SUB:FN\***

The solver specification implies that  $\bar{\Gamma}'' \llbracket I_f \rrbracket = \bar{\Gamma}'' \llbracket I'_f \rrbracket$ . Monotonicity implies that the variable bindings and solutions in the premises' output environments are preserved in  $\bar{\Gamma}'' = \bar{\Gamma}_{n+1}$ . By the induction hypothesis, each argument context  $\mathbf{C}_i$  coerces from  $\tau'_i$  to  $\tau_i$  under  $\bar{\Gamma}''$ , and the result context  $\mathbf{C}_o$  coerces from  $\tau_o$  to  $\tau'_o$  under  $\bar{\Gamma}''$ . Then the Function coercion lemma implies that  $\mathbf{C}_t$  coerces from  $\tau_l$  to  $\tau_h$ .

**Case SUB: $\forall$ \*L, SUB: $\Pi$ \*L**

The type (or index) application in the generated code coerces from the polymorphic type  $(A (\forall (\mathcal{X} \dots) (A \Upsilon_l I_c)) I_f)$  to the instantiation  $(A \Upsilon_l (++) I_f I_c) \llbracket \mathcal{X} \mapsto \bar{\Gamma}_2 \llbracket \hat{\mathcal{X}} \rrbracket, \dots \rrbracket$  (or from  $(A (\Pi (\mathcal{S} \dots) (A \Upsilon_l I_c)) I_f)$  to  $(A \Upsilon_l (++) I_f I_c) \llbracket \mathcal{S} \mapsto \bar{\Gamma}_2 \llbracket \hat{\mathcal{S}} \rrbracket, \dots \rrbracket$ ) under  $\bar{\Gamma}_1 = \bar{\Gamma}''$ . By the induction hypothesis,  $\mathbf{C}$  coerces from  $(A \Upsilon_l (++) I_f I_c)$  to  $(A \Upsilon_h I_h)$  under  $\bar{\Gamma}_1, \mathcal{X} \dots, \bar{\Gamma}_2$ . Thus composing the premise's context and the type (or index) application context produces a coercion from  $\tau_l$  to  $\tau_h$  under  $\bar{\Gamma}''$ .

**Case SUB: $\forall$ \*R, SUB: $\Pi$ \*R**

By the induction hypothesis, the premise's generated context  $\mathbf{C}$  coerces from  $(A \Upsilon_l I_l)$  to  $(A \Upsilon_h (++) I_f I_c)$  under  $\bar{\Gamma}_1, \mathcal{X} \dots, \bar{\Gamma}_2$ . Note that  $\bar{\Gamma}_2$  cannot contain new variable bindings according to the variable scoping lemma. So this coercion also works under  $\bar{\Gamma}_1, \mathcal{X} \dots$  (or under  $\bar{\Gamma}_1, \mathcal{S} \dots$ ). This premise also ensures that  $\bar{\Gamma}_1 \llbracket I_l \rrbracket = \bar{\Gamma}_1 \llbracket (++) I_f I_c \rrbracket$ . The type-abstraction (or index-abstraction) context in the conclusion provides a binder for the universal variables and produces an array of type  $(A (\forall (\mathcal{X} \dots) (A \Upsilon_h I_c)) (\text{shape}))$  (or  $(A (\Pi (\mathcal{S} \dots) (A \Upsilon_h I_c)) (\text{shape}))$ ) when the hole is filled with a term of type  $(A \Upsilon_h I_c)$ . We thus have a coercion from  $(A \Upsilon_h I_c)$  to  $(A (\forall (\mathcal{X} \dots) (A \Upsilon_h I_c)) (\text{shape}))$  (or  $(A (\Pi (\mathcal{S} \dots) (A \Upsilon_h I_c)) (\text{shape}))$ ) under  $\bar{\Gamma}_1 = \bar{\Gamma}''$ . The Each context must then coerce from  $(A \Upsilon_l (++) I_f I_c)$  to  $(A (\Pi (\mathcal{S} \dots) (A \Upsilon_h I_c)) I_f)$  under  $\bar{\Gamma}''$ .  $\bar{\Gamma}'' \llbracket I_l \rrbracket = \bar{\Gamma}'' \llbracket (++) I_f I_c \rrbracket$ , so this is a coercion from  $\tau_l$  to  $\tau_h$ .

**Case SUB:SIGMA\*L**

Each `box` in an array typed as the source type

$$(A (\Sigma (\mathcal{S} \dots) T_h) I_f)$$

must have contents typed as  $(A \Upsilon_l I_l)$ , though perhaps only typable with the  $\Sigma$ -bound variables. The conclusion's resulting context thus coerces the contents of each `box` within the source term to the goal cell type  $(A \Upsilon_h \hat{\sigma}_h)$ . The type of the `unbox` form itself, accounting for the frame shape  $I_f$ , is therefore  $(++ I_f \hat{\sigma}_h)$ . By the solver specification (and environment monotonicity), this is equal to the goal shape  $I_h$  after substitution according to  $\bar{\Gamma}''$ . So the frame-lifted `unbox` form has the type

$$\bar{\Gamma}'' \llbracket (A \Upsilon_h I_h) \rrbracket$$

Since the premise's context  $\mathbb{E}$  coerces to the goal cell type (by the induction hypothesis), which must only mention a subset of the index variables mentioned by the goal type itself, the `unbox` form's body type is well-formed even without the  $\Sigma$ -bound variables.

**Case SUB:SIGMA\*R**

The context  $\mathbb{E}$  generated by the premise coerces from the starting type to the unwrapped form of the goal type. The conclusion's `Each` context then wraps each cell in a `box`. As in the `ILOW:SIGMA*` case of Theorem 8.1.1 (Instantiation coercion), the existential witnesses we use here are the correct ones because they were substituted into the body of the cell type where the  $\Sigma$ -bound variables appeared initially. Composing  $\mathbb{E}$  and the `Each` context coerces from the starting type to the intermediate type

$$(A \Upsilon_h (++ I_f I_c)) \llbracket \mathcal{S} \mapsto \hat{\mathcal{S}}, \dots \rrbracket$$

and from there to the goal type

$$(A (\Sigma (\mathcal{S} \dots) (A \Upsilon_h I_h)) I_f)$$

**Case SUB:INST→L, SUB:INST→R**

The premise's input environment solves  $\hat{\mathcal{X}}$  as  $\Upsilon_f$ . Thus any coercion from  $(A \Upsilon_f I_l)$  to  $(A (-> (\tau_i \dots) \tau_o) I_h)$  under  $\bar{\Gamma}_1 = \bar{\Gamma}''$  also coerces from  $(A \hat{\mathcal{X}} I_l)$  to  $(A (-> (\tau_i \dots) \tau_o) I_h)$  and vice versa. The induction hypothesis implies that  $\mathbf{C}$  performs that coercion, so it satisfies the requirements for  $\mathbf{C}_t$ .  $\square$

**Theorem 8.1.3** (Elaboration soundness). *Given  $\bar{\Gamma}' \hat{\succeq} \bar{\Gamma}''$  with no existential variables appearing in  $\bar{\Gamma}'' \llbracket t \rrbracket$  and  $\text{SAT} \llbracket \Phi' \rrbracket$ , the following all hold:*

- $\bar{\Gamma}; \Phi \vdash \bar{t} \Leftarrow \tau \dashv \bar{\Gamma}'; \Phi' \leftrightarrow t$  implies that  
 $TB \llbracket \bar{\Gamma}'' \rrbracket; KB \llbracket \bar{\Gamma}'' \rrbracket; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket t \rrbracket : \bar{\Gamma}'' \llbracket \tau \rrbracket$
- $\bar{\Gamma}; \Phi \vdash \bar{t} \Rightarrow \tau \dashv \bar{\Gamma}'; \Phi' \leftrightarrow t$  implies that  
 $TB \llbracket \bar{\Gamma}'' \rrbracket; KB \llbracket \bar{\Gamma}'' \rrbracket; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket t \rrbracket : \bar{\Gamma}'' \llbracket \tau \rrbracket$

- $\bar{\Gamma}; \Phi \vdash (\bar{e}_f : \tau_f) \bullet [\bar{e}_a \dots] \Rightarrow \tau_r \dashv \bar{\Gamma}' ; \Phi' \hookrightarrow e_r$  implies that from  $TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket e_f \rrbracket : \bar{\Gamma}'' \llbracket \tau_f \rrbracket$  we can derive  $TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket e_r \rrbracket : \bar{\Gamma}'' \llbracket \tau_r \rrbracket$

*Proof.* We use induction on the bidirectional typing derivation.

**Case SYN:ANNOT**

The induction hypothesis is exactly the goal.

**Case SYN:VAR**

This case is trivial.

**Case SYN:UNBOX**

The variable scoping lemma ensures that  $\bar{\Gamma}_3$  contains no new variable bindings after  $x_e$ , so its contents are not needed in order to ascribe a type to the emitted expression. Monotonicity implies  $\bar{\Gamma}_1 \leq \bar{\Gamma}_2$  (by splitting at the bound index variables  $x_i \dots$ ) and  $\bar{\Gamma}_1 \leq \bar{\Gamma}_0 = \bar{\Gamma}$ . Since we have  $\bar{\Gamma}_1 \hat{\leq} \bar{\Gamma}''$  and  $\bar{\Gamma}_2 \hat{\leq} \bar{\Gamma}''$ , the induction hypothesis gives a type derivation for

$$TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket e_s \rrbracket : \bar{\Gamma}'' \llbracket (A \text{ (box } x'_i \dots \tau_s) I_s) \rrbracket$$

and for

$$TB \llbracket \bar{\Gamma}_p \rrbracket ; KB \llbracket \bar{\Gamma}_p \rrbracket ; SB \llbracket \bar{\Gamma}_p \rrbracket \vdash \bar{\Gamma}'' \llbracket e_s \rrbracket : \bar{\Gamma}'' \llbracket (A \text{ (box } x'_i \dots \tau_s) I_s) \rrbracket$$

where  $\bar{\Gamma}_p = \bar{\Gamma}'' , x_i \dots , x_e : \tau_s [x'_i \mapsto x_i, \dots]$ .

These are the premises for T-UNBOX, along with well-formedness of  $(A \hat{\alpha}_b \hat{\sigma}_b)$  under  $\bar{\Gamma}''$  (which contains the solutions for  $\hat{\alpha}_b$  and  $\hat{\sigma}_b$ ), thereby deriving

$$TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket (\text{unbox } (x_i \dots x_e e_s) e_b) \rrbracket : \bar{\Gamma}'' \llbracket (A \text{ } \Upsilon_b \text{ } (++) I_s \hat{\sigma}_b) \rrbracket$$

**Case SYN:APP**

The induction hypothesis for the first premise gives a derivation for

$$TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket e_f \rrbracket : \bar{\Gamma}'' \llbracket \tau_f \rrbracket$$

The induction hypothesis for the second premise ensures that from the previous derivation we can build a derivation for

$$TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket (e'_f e_a \dots) \rrbracket : \bar{\Gamma}'' \llbracket \tau_r \rrbracket$$

**Case SYN:FN**

The induction hypothesis implies that for any  $\bar{\Gamma}_f$  where  $\bar{\Gamma}_1, \blacktriangleright_{x_f}, \bar{\Gamma}_2 \hat{\leq} \bar{\Gamma}_f$ , we can type the function body:

$$TB \llbracket \bar{\Gamma}_f \rrbracket ; KB \llbracket \bar{\Gamma}_f \rrbracket ; SB \llbracket \bar{\Gamma}_f \rrbracket \vdash \bar{\Gamma}_f \llbracket e \rrbracket : \bar{\Gamma}_f \llbracket \tau_o \rrbracket$$

By the scoping lemma, the only variable bindings present in  $\bar{\Gamma}_2$  are the function's formal parameters. Any existential variables not solved by  $\bar{\Gamma}_2$  must be solved by the completion environment  $\bar{\Gamma}''$ . So the type annotations in the elaborated term are well-formed under  $\bar{\Gamma}''$ . Then by extending  $\bar{\Gamma}''$  with bindings for the formal parameters, T-LAM derives

$$TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \\ \bar{\Gamma}'' \llbracket \bar{\Gamma}_2 \llbracket (\lambda ((x \tau_i) \dots) e) \rrbracket \rrbracket : \bar{\Gamma}'' \llbracket \bar{\Gamma}_2 \llbracket (-> (\tau_i \dots) \tau_o) \rrbracket \rrbracket$$

### Case SYN:ARRAY

By the induction hypothesis and monotonicity, each atom in the array literal can be typed as

$$TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket a \rrbracket : \bar{\Gamma}'' \llbracket \Upsilon \rrbracket$$

We also have the required number of them. So T-ARRAY can derive

$$TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket (\text{array } (n \dots) a' \dots) \rrbracket \\ : \bar{\Gamma}'' \llbracket (A \Upsilon (\text{shape } n \dots)) \rrbracket$$

### Case SYN:FRAME

By the induction hypothesis and monotonicity, each cell in the frame form can be typed as

$$TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket e \rrbracket : \bar{\Gamma}'' \llbracket (A \Upsilon \iota) \rrbracket$$

We also have the required number of them. So T-FRAME can derive

$$TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket (\text{frame } (n \dots) \bar{e} \bar{e}' \dots) \rrbracket \\ : \bar{\Gamma}'' \llbracket (A \Upsilon (++) (\text{shape } n \dots) \iota) \rrbracket$$

### Case CHK:SUB

Monotonicity implies that  $\bar{\Gamma}_1 \hat{\leq} \bar{\Gamma}''$ . So the induction hypothesis gives a derivation for

$$TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket t \rrbracket : \bar{\Gamma}'' \llbracket \tau_l \rrbracket$$

Then coercion soundness for subtyping means that we can also derive

$$TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket C[t] \rrbracket : \bar{\Gamma}'' \llbracket \tau_h \rrbracket$$

### Case CHK:FN

This case differs from SYN:FN in the need to match up the goal's input

types with types derived from rank annotations on formal parameters. The induction hypothesis ascribes the same type to the function body  $\bar{e}$  in the SYN:FN case, but the bindings for the formals are supertypes of the specified input types. That is, with  $\bar{\Gamma}_f = \bar{\Gamma}'', x : \text{ElabType} \llbracket x, \varsigma \rrbracket \dots$ ,

$$TB \llbracket \bar{\Gamma}_f \rrbracket ; KB \llbracket \bar{\Gamma}_f \rrbracket ; SB \llbracket \bar{\Gamma}_f \rrbracket \vdash \bar{\Gamma}'' \llbracket e \rrbracket : \bar{\Gamma}'' \llbracket \tau_o \rrbracket$$

The subtyping coercion soundness theorem ensures that the resulting contexts  $\mathbb{E} \dots$  coerce from  $\tau_i \dots$  to the types generated by rank elaboration. So the substitution in the emitted function body  $e$  ensures that the body will type check when  $\bar{\Gamma}$  is augmented with bindings  $x : \tau_i \dots$ :

$$TB \llbracket \bar{\Gamma}'', x : \tau_i \dots \rrbracket ; KB \llbracket \bar{\Gamma}'', x : \tau_i \dots \rrbracket ; SB \llbracket \bar{\Gamma}'', x : \tau_i \dots \rrbracket \vdash \\ \bar{\Gamma}'' \llbracket e[x \mapsto \mathbb{E}[x], \dots] \rrbracket : \bar{\Gamma}'' \llbracket \tau_o \rrbracket$$

This premise allows T-LAM to ascribe the type  $(\rightarrow (\tau_i \dots) \tau_o)$  to the emitted function  $(\lambda ((x \tau_i) \dots) e[x \mapsto \mathbb{E}[x], \dots])$ .

#### Case CHK:SIGMA

From the induction hypothesis, we have

$$TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket e \rrbracket : \bar{\Gamma}'' \llbracket \tau[x \mapsto \iota, \dots] \rrbracket$$

The ascribed type is equivalent to  $\bar{\Gamma}'' \llbracket \tau \rrbracket [x \mapsto \bar{\Gamma}'' \llbracket \iota \rrbracket, \dots]$ . This serves as the typing premise for T-BOX to conclude the elaborated term has type  $(\Sigma (x \dots) \bar{\Gamma}'' \llbracket \tau \rrbracket)$ . The well-formedness requirements are covered by the premises of CHK:SIGMA.

#### Case CHK:PI

By the scoping lemma, no new variable bindings can appear in  $\bar{\Gamma}_2$ . Since  $\bar{\Gamma}_1 \hat{\leq} \bar{\Gamma}''$ , and variable bindings do not affect substitution, the induction hypothesis implies

$$TB \llbracket \bar{\Gamma}'', x \dots \rrbracket ; KB \llbracket \bar{\Gamma}'', x \dots \rrbracket ; SB \llbracket \bar{\Gamma}'', x \dots \rrbracket \vdash \bar{\Gamma}'' \llbracket \mathbf{v} \rrbracket : \bar{\Gamma}'' \llbracket \Upsilon \rrbracket$$

Then T-ILAM uses this as its premise to derive

$$TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket (\text{I}\lambda (x \dots) (\text{array} () \mathbf{v})) \rrbracket \\ : \bar{\Gamma}'' \llbracket (\Pi (x \dots) (\text{A } \Upsilon (\text{shape}))) \rrbracket$$

#### Case CHK:ALL

This case proceeds much like CHK:PI. From the induction hypothesis, we have

$$TB \llbracket \bar{\Gamma}'', x \dots \rrbracket ; KB \llbracket \bar{\Gamma}'', x \dots \rrbracket ; SB \llbracket \bar{\Gamma}'', x \dots \rrbracket \vdash \bar{\Gamma}'' \llbracket \mathbf{v} \rrbracket : \bar{\Gamma}'' \llbracket \Upsilon \rrbracket$$

This serves as the premise for T-TLAM, giving the conclusion

$$TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket (\top \lambda (x \dots) (\text{array } () \mathfrak{v})) \rrbracket \\ : \bar{\Gamma}'' \llbracket (\forall (x \dots) (A \top (\text{shape}))) \rrbracket$$

**Case CHK:ARRAY**

This case is similar to SYN:ARRAY, except that we do not directly take the array literal's annotated shape. Instead, the final premise asks the solver to equate the annotated shape with the goal shape. If  $SAT \llbracket \Phi_{n+1} \rrbracket$ , then this request succeeded, so a T-EQV step after T-ARRAY can ascribe the desired type.

**Case CHK:FRAME**

This case is similar to SYN:ARRAY, using a goal shape instead of the frame form's own shape. As in CHK:ARRAY, the solver equates the goal shape with the concatenation of the annotated frame shape and cell shape. If successful, *i.e.*,  $SAT \llbracket \Phi_{n+1} \rrbracket$ , then the T-FRAME derivation can be followed with T-EQV to ascribe the desired type.

**Case APP:ALL**

By the induction hypothesis, from the premise

$$TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket (\text{t-app } e_f \hat{x} \dots) \rrbracket \\ : \bar{\Gamma}'' \llbracket (A \tau_f (++) \iota_a \iota_f) [x \mapsto \hat{x}, \dots] \rrbracket$$

we can derive

$$TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket (e'_f e_a \dots) \rrbracket : \bar{\Gamma}'' \llbracket \tau_r \rrbracket$$

Our goal is to show that the same result can be derived from the type ascribed to  $e_f$  itself. Taking as a premise

$$TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket e_f \rrbracket \\ : \bar{\Gamma}'' \llbracket (A (\forall (x \dots) (A \tau_f \iota_f)) \iota_a) \rrbracket$$

allows T-TAPP to conclude the earlier result ascribing the goal type to  $(\text{t-app } e_f \hat{x} \dots)$ . So the required implication holds.

**Case APP:PI**

This case is similar to APP:ALL. The induction hypothesis states that the following premise

$$TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket (\text{i-app } e_f \hat{x} \dots) \rrbracket \\ : \bar{\Gamma}'' \llbracket (A \tau_f (++) \iota_a \iota_f) [x \mapsto \hat{x}, \dots] \rrbracket$$

allows a type derivation to reach the following conclusion

$$TB \llbracket \bar{\Gamma}'' \rrbracket ; KB \llbracket \bar{\Gamma}'' \rrbracket ; SB \llbracket \bar{\Gamma}'' \rrbracket \vdash \bar{\Gamma}'' \llbracket (e'_f e_a \dots) \rrbracket : \bar{\Gamma}'' \llbracket \tau_r \rrbracket$$

Taking as a premise the expectation noted in the conclusion of APP:ALL that

$$\begin{aligned} TB \left[ \bar{\Gamma}'' \right]; KB \left[ \bar{\Gamma}'' \right]; SB \left[ \bar{\Gamma}'' \right] \vdash \bar{\Gamma}'' \left[ e_f \right] \\ : \bar{\Gamma}'' \left[ (A (\Pi (x \dots) (A \tau_f \iota_f)) \iota_a) \right] \end{aligned}$$

leads via T-IAPP to the premise proposed by the induction hypothesis, and in turn the required conclusion.

**Case APP:FN\*F, APP:FN\*A**

The goal is to show that from the “goal premise,” which is

$$\begin{aligned} TB \left[ \bar{\Gamma}'' \right]; KB \left[ \bar{\Gamma}'' \right]; SB \left[ \bar{\Gamma}'' \right] \vdash \bar{\Gamma}'' \left[ e_f \right] \\ : \bar{\Gamma}'' \left[ (A (-> ((A \tau_i I_i) \tau'_a \dots) \tau_o) I_f) \right] \end{aligned}$$

we can derive the “goal conclusion”

$$TB \left[ \bar{\Gamma}'' \right]; KB \left[ \bar{\Gamma}'' \right]; SB \left[ \bar{\Gamma}'' \right] \vdash \bar{\Gamma}'' \left[ (e_f e_a e'_a \dots) \right] : \bar{\Gamma}'' \left[ \tau_r \right]$$

From the induction hypothesis, we have a derivation (*ab initio*) for

$$TB \left[ \bar{\Gamma}'' \right]; KB \left[ \bar{\Gamma}'' \right]; SB \left[ \bar{\Gamma}'' \right] \vdash \bar{\Gamma}'' \left[ e_a \right] : \bar{\Gamma}'' \left[ (A \tau_i ( ++ I_F I_i)) \right]$$

taking  $I_F = (\text{shape})$  for the APP:FN\*C case, and we also have the ability to construct from (in the APP:FN\*F and APP:FN\*C cases)

$$\begin{aligned} TB \left[ \bar{\Gamma}'' \right]; KB \left[ \bar{\Gamma}'' \right]; SB \left[ \bar{\Gamma}'' \right] \vdash \bar{\Gamma}'' \left[ e_f \right] \\ : \bar{\Gamma}'' \left[ (A (-> (\tau'_a \dots) \tau_o) I_f) \right] \end{aligned}$$

or (in the APP:FN\*A case)<sup>34</sup>

$$\begin{aligned} TB \left[ \bar{\Gamma}'' \right]; KB \left[ \bar{\Gamma}'' \right]; SB \left[ \bar{\Gamma}'' \right] \vdash \bar{\Gamma}'' \left[ e_f \right] \\ : \bar{\Gamma}'' \left[ (A (-> (\tau'_a \dots) \tau_o) I_F) \right] \end{aligned}$$

a derivation for

$$TB \left[ \bar{\Gamma}'' \right]; KB \left[ \bar{\Gamma}'' \right]; SB \left[ \bar{\Gamma}'' \right] \vdash \bar{\Gamma}'' \left[ (e_f e'_a \dots) \right] : \bar{\Gamma}'' \left[ \tau_r \right]$$

We also have from the solver a guarantee of frame compatibility between the function  $e_f$  and the first argument  $e_a$ .

The judgment which the induction hypothesis offers for the application premise must use T-APP with some principal frame  $I_p$ . Premises for that use of T-APP must also ascribe to the arguments  $e'_a \dots$  types which are compatible with the respective  $\tau'_a \dots$  and have frames compatible with  $I_p$ . The frame shape ascribed to the function  $e_f$  in the assumption must also be compatible with  $I_p$ . That shape is either  $I_f$  (for APP:FN\*F and APP:FN\*C) or  $I_F$  (for APP:FN\*A), whichever turns out to be larger.

<sup>34</sup> The only difference between these two is the shape ascribed to  $e_f$ , either  $I_f$  or  $I_F$ .

Either way, the solver has guaranteed that  $I_p$  is compatible with the shape of  $e_a$ .

So the goal premise, ascribing the higher-arity type to  $e_f$ , allows  $e_f$  to take the one extra argument  $e_a$  and recycle the portions of the derivation offered by the induction hypothesis. The output type  $e_f$  is assumed to have does not change. For  $\text{APP:FN}^*\text{F}$  and  $\text{APP:FN}^*\text{C}$ , the frame shape of  $e_f$  also does not change. For  $\text{APP:FN}^*\text{A}$ , the frame shape changes, but it does so by shrinking from having the same frame as  $e_a$ . That is, in the induction hypothesis's derivation,  $e_f$  had forced the principal frame to be at least as large as  $I_F$ , and  $e_a$  now sets that same lower bound. Thus the principal frame shape remains unchanged, and the goal conclusion is reached with the same result type  $\tau_r$ .

**Case APP:FN0**

From the assumption that  $e$  is an array of nullary functions with output type  $(A \tau_o I_o)$  and shape  $I_f$ , T-APP concludes that applying  $e$  to no arguments gives output of type  $(A \tau_o ( ++ I_f I_o ))$ .  $\square$



# D

## PROOFS (10.2: CORRECTNESS OF TRANSLATION)

**Lemma 10.2.1** (Erasure in context). *Given an evaluation context  $\mathbb{V}$  and expression  $e$ , where  $\mathbb{V}[e]$  is well-typed,  $\mathcal{E}[\mathbb{V}[e]] = \mathcal{C}[\mathbb{V}][\mathcal{E}[e]]$ .*

*Proof.* We use induction on  $\mathbb{V}$ .

ARRAY LITERAL CONTAINING UNEVALUATED BOX:

$$\begin{aligned}
 \mathbb{V} &= (\text{array } (n \dots) \mathbf{v} \dots (\text{box } \iota \dots \mathbb{V}' \tau) \mathbf{a} \dots) \\
 \mathcal{E}[(\text{array } (n \dots) \mathbf{v} \dots (\text{box } \iota \dots \mathbb{V}' \tau) \mathbf{a} \dots)[e]] &= \mathcal{E}[(\text{array } (n \dots) \mathbf{v} \dots (\text{box } \iota \dots \mathbb{V}'[e] \tau) \mathbf{a} \dots)] \\
 &= (\text{array } (n \dots) \mathcal{A}[\mathbf{v}] \dots \mathcal{A}[(\text{box } \iota \dots \mathbb{V}'[e] \tau)] \mathcal{A}[\mathbf{a}] \dots) \\
 &= (\text{array } (n \dots) \mathcal{A}[\mathbf{v}] \dots (\text{box } \iota \dots \mathcal{E}[\mathbb{V}'[e]]) \mathcal{A}[\mathbf{a}] \dots) \\
 &= (\text{array } (n \dots) \mathcal{A}[\mathbf{v}] \dots (\text{box } \iota \dots \mathcal{C}[\mathbb{V}'][\mathcal{E}[e]]) \mathcal{A}[\mathbf{a}] \dots) \\
 &= (\text{array } (n \dots) \mathcal{A}[\mathbf{v}] \dots (\text{box } \iota \dots \mathcal{C}[\mathbb{V}']) \mathcal{A}[\mathbf{a}] \dots)[\mathcal{E}[e]] \\
 &= \mathcal{C}[(\text{array } (n \dots) \mathbf{v} \dots (\text{box } \iota \dots \mathbb{V}' \tau) \mathbf{a} \dots)][\mathcal{E}[e]]
 \end{aligned}$$

FRAME CONTAINING UNEVALUATED CELLS:

$$\begin{aligned}
 \mathbb{V} &= (\text{frame } (n \dots) \mathbf{v} \dots \mathbb{V}' e' \dots)^{\tau_r} \\
 \mathcal{E}[(\text{frame } (n \dots) \mathbf{v} \dots \mathbb{V}' e' \dots)^{\tau_r}[e]] &= \mathcal{E}[(\text{frame } (n \dots) \mathbf{v} \dots \mathbb{V}'[e] e' \dots)^{\tau_r}] \\
 &= (\text{frame } (\mathcal{T}[\tau_r]) \mathcal{E}[\mathbf{v}] \dots \mathcal{E}[\mathbb{V}'[e]] \mathcal{E}[e'] \dots) \\
 &= (\text{frame } (\mathcal{T}[\tau_r]) \mathcal{E}[\mathbf{v}] \dots \mathcal{C}[\mathbb{V}'][\mathcal{E}[e]] \mathcal{E}[e'] \dots) \\
 &= (\text{frame } (\mathcal{T}[\tau_r]) \mathcal{E}[\mathbf{v}] \dots \mathcal{C}[\mathbb{V}'] \mathcal{E}[e'] \dots)[\mathcal{E}[e]] \\
 &= \mathcal{C}[(\text{frame } (n \dots) \mathbf{v} \dots \mathbb{V}' e' \dots)^{\tau_r}][\mathcal{E}[e]]
 \end{aligned}$$

APPLICATION WITH UNEVALUATED FUNCTION:

$$\begin{aligned}
 \mathbb{V} &= (\mathbb{V}'^{(\lambda (\tau_i \dots) \tau_o) \iota_f} e_a \dots)^{\tau_r} \\
 \mathcal{E}[(\mathbb{V}'^{(\lambda (\tau_i \dots) \tau_o) \iota_f} e_a \dots)^{\tau_r}[e]] &= \mathcal{E}[(\mathbb{V}'^{(\lambda (\tau_i \dots) \tau_o) \iota_f} [e] e_a \dots)^{\tau_r}] \\
 &= (\mathcal{E}[\mathbb{V}'[e]] (\mathcal{E}[e_a] \mathcal{T}[\tau_i]) \dots \mathcal{T}[\tau_r]) \\
 &= (\mathcal{C}[\mathbb{V}'] [e] (\mathcal{E}[e_a] \mathcal{T}[\tau_i]) \dots \mathcal{T}[\tau_r]) \\
 &= (\mathcal{C}[\mathbb{V}'] (\mathcal{E}[e_a] \mathcal{T}[\tau_i]) \dots \mathcal{T}[\tau_r])[e] \\
 &= \mathcal{C}[(\mathbb{V}'^{(\lambda (\tau_i \dots) \tau_o) \iota_f} e_a \dots)] [e]
 \end{aligned}$$

APPLICATION WITH UNEVALUATED ARGUMENT:

$$\begin{aligned}
\mathbb{V} &= (e_f^{(A \rightarrow (\tau_1 \dots \tau_2 \tau_3 \dots) \tau_o) \iota_f}) v_1 \dots \mathbb{V}' e_3 \dots)^{\tau_r} \\
\mathcal{E} \left[ (e_f^{(A \rightarrow (\tau_1 \dots \tau_2 \tau_3 \dots) \tau_o) \iota_f}) v_1 \dots \mathbb{V}' e_3 \dots)^{\tau_r} [e] \right] & \\
&= \mathcal{E} \left[ (e_f^{(A \rightarrow (\tau_1 \dots \tau_2 \tau_3 \dots) \tau_o) \iota_f}) v_1 \dots \mathbb{V}' [e] e_3 \dots)^{\tau_r} \right] \\
&= (\mathcal{E}[e_f] (\mathcal{E}[v_1] \mathcal{T}[\tau_1]) \dots (\mathcal{E}[\mathbb{V}'[e]] \mathcal{T}[\tau_2]) \\
&\quad (\mathcal{E}[e_3] \mathcal{T}[\tau_3]) \dots \mathcal{T}[\tau_r]) \\
&= (\mathcal{E}[e_f] (\mathcal{E}[v_1] \mathcal{T}[\tau_1]) \dots (\mathcal{C}[\mathbb{V}'][\mathcal{E}[e]] \mathcal{T}[\tau_2]) \\
&\quad (\mathcal{E}[e_3] \mathcal{T}[\tau_3]) \dots \mathcal{T}[\tau_r]) \\
&= (\mathcal{E}[e_f] (\mathcal{E}[v_1] \mathcal{T}[\tau_1]) \dots (\mathcal{C}[\mathbb{V}'] \mathcal{T}[\tau_2]) \\
&\quad (\mathcal{E}[e_3] \mathcal{T}[\tau_3]) \dots \mathcal{T}[\tau_r]) [\mathcal{E}[e]] \\
&= \mathcal{C} \left[ (e_f^{(A \rightarrow (\tau_1 \dots \tau_2 \tau_3 \dots) \tau_o) \iota_f}) v_1 \dots \mathbb{V}' e_3 \dots)^{\tau_r} \right] [\mathcal{E}[e]]
\end{aligned}$$

TYPE APPLICATION:

$$\begin{aligned}
\mathbb{V} &= (\text{t-app } \mathbb{V}' \tau_a \dots)^{\tau_r} \\
\mathcal{E} \left[ (\text{t-app } \mathbb{V}' \tau_a \dots)^{\tau_r} [e] \right] & \\
\mathcal{E} \left[ (\text{t-app } \mathbb{V}' [e] \tau_a \dots)^{\tau_r} \right] & \\
&= (\text{i-app } \mathcal{E}[\mathbb{V}'[e]] \mathcal{T}[\tau_a] \dots \mathcal{T}[\tau_r]) \\
&= (\text{i-app } \mathcal{C}[\mathbb{V}'][\mathcal{E}[e]] \mathcal{T}[\tau_a] \dots \mathcal{T}[\tau_r]) \\
&= (\text{i-app } \mathcal{C}[\mathbb{V}'] \mathcal{T}[\tau_a] \dots \mathcal{T}[\tau_r]) [\mathcal{E}[e]] \\
&= \mathcal{C} \left[ (\text{t-app } \mathbb{V}' \tau_a \dots)^{\tau_r} \right] [\mathcal{E}[e]]
\end{aligned}$$

INDEX APPLICATION:

$$\begin{aligned}
\mathbb{V} &= (\text{i-app } \mathbb{V}' \iota_a \dots)^{\tau_r} \\
\mathcal{E} \left[ (\text{i-app } \mathbb{V}' \iota_a \dots)^{\tau_r} [e] \right] & \\
&= \mathcal{E} \left[ (\text{i-app } \mathbb{V}' [e] \iota_a \dots)^{\tau_r} \right] \\
&= (\text{i-app } \mathcal{E}[\mathbb{V}'[e]] \iota_a \dots \mathcal{T}[\tau_r]) \\
&= (\text{i-app } \mathcal{C}[\mathbb{V}'][\mathcal{E}[e]] \iota_a \dots \mathcal{T}[\tau_r]) \\
&= (\text{i-app } \mathcal{C}[\mathbb{V}'] \iota_a \dots \mathcal{T}[\tau_r]) [\mathcal{E}[e]] \\
&= \mathcal{C} \left[ (\text{i-app } \mathbb{V}' \iota_a \dots)^{\tau_r} \right] [\mathcal{E}[e]]
\end{aligned}$$

UNBOXING:

$$\begin{aligned}
\mathbb{V} &= (\text{unbox } (x_i \dots x_e \mathbb{V}') e_b^{\tau_b}) \\
\mathcal{E} \left[ (\text{unbox } (x_i \dots x_e \mathbb{V}') e_b^{\tau_b}) [e] \right] & \\
&= \mathcal{E} \left[ (\text{unbox } (x_i \dots x_e \mathbb{V}' [e]) e_b^{\tau_b}) \right] \\
&= (\text{unbox } (x_i \dots x_e \mathcal{E}[\mathbb{V}'[e]]) \mathcal{E}[e_b^{\tau_b}] \mathcal{T}[\tau_b]) \\
&= (\text{unbox } (x_i \dots x_e \mathcal{C}[\mathbb{V}'][\mathcal{E}[e]]) \mathcal{E}[e_b^{\tau_b}] \mathcal{T}[\tau_b])
\end{aligned}$$

$$\begin{aligned}
&= (\text{unbox } (x_i \dots x_e \mathcal{C}[\mathbb{V}']) \mathcal{E}[[e_b^{t_b}]] \mathcal{T}[\tau_b])[\mathcal{E}[e]] \\
&= \mathcal{C}[(\text{unbox } (x_i \dots x_e \mathbb{V}') e_b)][\mathcal{E}[e]]
\end{aligned}$$

□

**Lemma 10.2.2** (Substituting terms into terms commutes with erasure).

$$\mathcal{R}[[t[x \mapsto \mathcal{E}[e_x]]]] = \mathcal{R}[[t][x \mapsto \mathcal{E}[e_x]]]$$

*Proof.* We use induction on  $t$ . We skip the cases where  $x$  does not appear free in  $t$ , as substitution would not change  $t$ .

TERM ABSTRACTION:

$$t = (\lambda ((x_i \tau) \dots) e), \text{ where } x \notin x_i \dots$$

$$\begin{aligned}
&\mathcal{R}[(\lambda ((x_i \tau) \dots) e)[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{A}[(\lambda ((x_i \tau) \dots) e)[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{A}[(\lambda ((x_i \tau) \dots) e[x \mapsto \mathcal{E}[e_x]])] \\
&= (\lambda (x_i \dots) \mathcal{E}[e[x \mapsto \mathcal{E}[e_x]]]) \\
&= (\lambda (x_i \dots) \mathcal{E}[e][x \mapsto \mathcal{E}[e_x]]) \\
&= (\lambda (x_i \dots) \mathcal{E}[e])[x \mapsto \mathcal{E}[e_x]] \\
&= \mathcal{A}[(\lambda ((x_i \tau) \dots) e)][x \mapsto \mathcal{E}[e_x]] \\
&= \mathcal{R}[(\lambda ((x_i \tau) \dots) e)][x \mapsto \mathcal{E}[e_x]]
\end{aligned}$$

TYPE ABSTRACTION:

$$t = (\text{T}\lambda ((x_i k) \dots) v)$$

$$\begin{aligned}
&\mathcal{R}[(\text{T}\lambda ((x_i k) \dots) v)[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{A}[(\text{T}\lambda ((x_i k) \dots) v)[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{A}[(\text{T}\lambda ((x_i k) \dots) v[x \mapsto \mathcal{E}[e_x]])] \\
&= (\text{I}\lambda (x_i \dots) \mathcal{E}[v[x \mapsto \mathcal{E}[e_x]]]) \\
&= (\text{I}\lambda (x_i \dots) \mathcal{E}[v][x \mapsto \mathcal{E}[e_x]]) \\
&= (\text{I}\lambda (x_i \dots) \mathcal{E}[v])[x \mapsto \mathcal{E}[e_x]] \\
&= \mathcal{A}[(\text{T}\lambda ((x_i k) \dots) v)][x \mapsto \mathcal{E}[e_x]] \\
&= \mathcal{R}[(\text{T}\lambda ((x_i k) \dots) v)][x \mapsto \mathcal{E}[e_x]]
\end{aligned}$$

INDEX ABSTRACTION:

$$t = (\text{I}\lambda ((x_i \gamma) \dots) v)$$

$$\begin{aligned}
&\mathcal{R}[(\text{I}\lambda ((x_i \gamma) \dots) v)[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{A}[(\text{I}\lambda ((x_i \gamma) \dots) v)[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{A}[(\text{I}\lambda ((x_i \gamma) \dots) v[x \mapsto \mathcal{E}[e_x]])] \\
&= (\text{I}\lambda (x_i \dots) \mathcal{E}[v[x \mapsto \mathcal{E}[e_x]]]) \\
&= (\text{I}\lambda (x_i \dots) \mathcal{E}[v][x \mapsto \mathcal{E}[e_x]]) \\
&= (\text{I}\lambda (x_i \dots) \mathcal{E}[v])[x \mapsto \mathcal{E}[e_x]]
\end{aligned}$$

$$\begin{aligned}
&= \mathcal{A}[(\text{I}\lambda ((x_i \gamma) \dots) v)][x \mapsto \mathcal{E}[e_x]] \\
&= \mathcal{R}[(\text{I}\lambda ((x_i \gamma) \dots) v)][x \mapsto \mathcal{E}[e_x]]
\end{aligned}$$

BOX:

$$t = (\text{box } \iota \dots e_s \tau)$$

$$\begin{aligned}
&\mathcal{R}[(\text{box } \iota \dots e_s \tau)[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{A}[(\text{box } \iota \dots e_s \tau)[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{A}[(\text{box } \iota \dots e_s[x \mapsto \mathcal{E}[e_x]] \tau)] \\
&= (\text{box } \iota \dots \mathcal{E}[e_s[x \mapsto \mathcal{E}[e_x]]]) \\
&= (\text{box } \iota \dots \mathcal{E}[e_s][x \mapsto \mathcal{E}[e_x]]) \\
&= (\text{box } \iota \dots \mathcal{E}[e_s])[x \mapsto \mathcal{E}[e_x]] \\
&= \mathcal{A}[(\text{box } \iota \dots e_s \tau)][x \mapsto \mathcal{E}[e_x]] \\
&= \mathcal{R}[(\text{box } \iota \dots e_s \tau)][x \mapsto \mathcal{E}[e_x]]
\end{aligned}$$

VARIABLE:

$$t = x$$

$$\begin{aligned}
&\mathcal{R}[x[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{E}[x[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{E}[e_x] \\
&= x[x \mapsto \mathcal{E}[e_x]] \\
&= \mathcal{E}[x][x \mapsto \mathcal{E}[e_x]] \\
&= \mathcal{R}[x][x \mapsto \mathcal{E}[e_x]]
\end{aligned}$$

ARRAY:

$$t = (\text{array } (n \dots) \mathfrak{a} \dots)$$

$$\begin{aligned}
&\mathcal{R}[(\text{array } (n \dots) \mathfrak{a} \dots)[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{E}[(\text{array } (n \dots) \mathfrak{a} \dots)[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{E}[(\text{array } (n \dots) \mathfrak{a}[x \mapsto \mathcal{E}[e_x]] \dots)] \\
&= (\text{array } (n \dots) \mathcal{A}[\mathfrak{a}[x \mapsto \mathcal{E}[e_x]]] \dots) \\
&= (\text{array } (n \dots) \mathcal{A}[\mathfrak{a}][x \mapsto \mathcal{E}[e_x]] \dots) \\
&= (\text{array } (n \dots) \mathcal{A}[\mathfrak{a}] \dots)[x \mapsto \mathcal{E}[e_x]] \\
&= \mathcal{E}[(\text{array } (n \dots) \mathfrak{a} \dots)][x \mapsto \mathcal{E}[e_x]] \\
&= \mathcal{R}[(\text{array } (n \dots) \mathfrak{a} \dots)][x \mapsto \mathcal{E}[e_x]]
\end{aligned}$$

FRAME:

$$t = (\text{frame } (n \dots) e_c \dots)^{\tau_r}$$

$$\begin{aligned}
& \mathcal{R}[(\text{frame } (n \dots) e_c \dots)^{\tau_r}[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{E}[(\text{frame } (n \dots) e_c \dots)^{\tau_r}[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{E}[(\text{frame } (n \dots) e_c[x \mapsto \mathcal{E}[e_x]] \dots)^{\tau_r}] \\
&= (\text{frame } (\mathcal{T}[\tau_r]) \mathcal{E}[e_c[x \mapsto \mathcal{E}[e_x]]] \dots) \\
&= (\text{frame } (\mathcal{T}[\tau_r]) \mathcal{E}[e_c][x \mapsto \mathcal{E}[e_x]] \dots) \\
&= (\text{frame } (\mathcal{T}[\tau_r]) \mathcal{E}[e_c] \dots)[x \mapsto \mathcal{E}[e_x]] \\
&= \mathcal{E}[(\text{frame } (n \dots) e_c \dots)^{\tau_r}[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{R}[(\text{frame } (n \dots) e_c \dots)^{\tau_r}[x \mapsto \mathcal{E}[e_x]]]
\end{aligned}$$

APPLICATION:

$$t = (e_f^{(A \rightarrow (\tau_i \dots) \tau_o) \iota_f} e_a \dots)^{\tau_r}$$

$$\begin{aligned}
& \mathcal{R}[(e_f^{(A \rightarrow (\tau_i \dots) \tau_o) \iota_f} e_a \dots)^{\tau_r}[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{E}[(e_f^{(A \rightarrow (\tau_i \dots) \tau_o) \iota_f} e_a \dots)^{\tau_r}[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{E}[(e_f^{(A \rightarrow (\tau_i \dots) \tau_o) \iota_f} [x \mapsto \mathcal{E}[e_x]] e_a[x \mapsto \mathcal{E}[e_x]] \dots)^{\tau_r}] \\
&= (\mathcal{E}[e_f[x \mapsto \mathcal{E}[e_x]]] (\mathcal{E}[e_a[x \mapsto \mathcal{E}[e_x]]] \mathcal{T}[\tau_i]) \dots \mathcal{T}[\tau_r]) \\
&= (\mathcal{E}[e_f][x \mapsto \mathcal{E}[e_x]] (\mathcal{E}[e_a][x \mapsto \mathcal{E}[e_x]] \mathcal{T}[\tau_i]) \dots \mathcal{T}[\tau_r]) \\
&= (\mathcal{E}[e_f] (\mathcal{E}[e_a] \mathcal{T}[\tau_i]) \dots \mathcal{T}[\tau_r])[x \mapsto \mathcal{E}[e_x]] \\
&= \mathcal{E}[(e_f^{(A \rightarrow (\tau_i \dots) \tau_o) \iota_f} e_a \dots)^{\tau_r}[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{R}[(e_f^{(A \rightarrow (\tau_i \dots) \tau_o) \iota_f} e_a \dots)^{\tau_r}[x \mapsto \mathcal{E}[e_x]]]
\end{aligned}$$

TYPE APPLICATION:

$$t = (\text{t-app } e_f \tau_a \dots)^{\tau_r}$$

$$\begin{aligned}
& \mathcal{R}[(\text{t-app } e_f \tau_a \dots)^{\tau_r}[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{E}[(\text{t-app } e_f \tau_a \dots)^{\tau_r}[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{E}[(\text{t-app } e_f[x \mapsto \mathcal{E}[e_x]] \tau_a \dots)^{\tau_r}] \\
&= (\text{i-app } \mathcal{E}[e_f[x \mapsto \mathcal{E}[e_x]]] \mathcal{T}[\tau_a] \dots \mathcal{T}[\tau_r]) \\
&= (\text{i-app } \mathcal{E}[e_f][x \mapsto \mathcal{E}[e_x]] \mathcal{T}[\tau_a] \dots \mathcal{T}[\tau_r]) \\
&= (\text{i-app } \mathcal{E}[e_f] \mathcal{T}[\tau_a] \dots \mathcal{T}[\tau_r])[x \mapsto \mathcal{E}[e_x]] \\
&= \mathcal{E}[(\text{t-app } e_f \tau_a \dots)^{\tau_r}[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{R}[(\text{t-app } e_f \tau_a \dots)^{\tau_r}[x \mapsto \mathcal{E}[e_x]]]
\end{aligned}$$

INDEX APPLICATION:

$$t = (\text{i-app } e_f \iota_a \dots)^{\tau_r}$$

$$\begin{aligned}
& \mathcal{R}[(\text{i-app } e_f \iota_a \dots)^{\tau_r}[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{E}[(\text{i-app } e_f \iota_a \dots)^{\tau_r}[x \mapsto \mathcal{E}[e_x]]] \\
&= \mathcal{E}[(\text{i-app } e_f[x \mapsto \mathcal{E}[e_x]] \iota_a \dots)^{\tau_r}]
\end{aligned}$$

$$\begin{aligned}
&= (\text{i-app } \mathcal{E}[\![e_f[x \mapsto \mathcal{E}[e_x]]]\!] \iota_a \dots T[\![\tau_r]\!] ) \\
&= (\text{i-app } \mathcal{E}[\![e_f]\!][x \mapsto \mathcal{E}[e_x]] \iota_a \dots T[\![\tau_r]\!] ) \\
&= (\text{i-app } \mathcal{E}[\![e_f]\!] \iota_a \dots T[\![\tau_r]\!] ) [x \mapsto \mathcal{E}[e_x]] \\
&= \mathcal{E}[\!(\text{i-app } e_f \iota_a \dots )^{\tau_r}\!] [x \mapsto \mathcal{E}[e_x]] \\
&= \mathcal{R}[\!(\text{i-app } e_f \iota_a \dots )^{\tau_r}\!] [x \mapsto \mathcal{E}[e_x]]
\end{aligned}$$

UNBOXING:

$$\begin{aligned}
t &= (\text{unbox } (x_i \dots x_e e_s) e_b^{\tau_b}) \\
\mathcal{R}[\!(\text{unbox } (x_i \dots x_e e_s) e_b^{\tau_b}) [x \mapsto \mathcal{E}[e_x]]\!] & \\
&= \mathcal{E}[\!(\text{unbox } (x_i \dots x_e e_s) e_b^{\tau_b}) [x \mapsto \mathcal{E}[e_x]]\!] \\
&= \mathcal{E}[\!(\text{unbox } (x_i \dots x_e e_s [x \mapsto \mathcal{E}[e_x]]) e_b^{\tau_b} [x \mapsto \mathcal{E}[e_x]])\!] \\
&= (\text{unbox } (x_i \dots x_e \mathcal{E}[\![e_s[x \mapsto \mathcal{E}[e_x]]]\!] ) \mathcal{E}[\![e_b^{\tau_b}[x \mapsto \mathcal{E}[e_x]]]\!] T[\![\tau_b]\!] ) \\
&= (\text{unbox } (x_i \dots x_e \mathcal{E}[\![e_s]\!][x \mapsto \mathcal{E}[e_x]]) \mathcal{E}[\![e_b^{\tau_b}]\!][x \mapsto \mathcal{E}[e_x]] T[\![\tau_b]\!] ) \\
&= (\text{unbox } (x_i \dots x_e e_s) e_b T[\![\tau_b]\!] ) [x \mapsto \mathcal{E}[e_x]] \\
&= \mathcal{E}[\!(\text{unbox } (x_i \dots x_e e_s) e_b)\!] [x \mapsto \mathcal{E}[e_x]] \\
&= \mathcal{R}[\!(\text{unbox } (x_i \dots x_e e_s) e_b)\!] [x \mapsto \mathcal{E}[e_x]]
\end{aligned}$$

□

**Lemma 10.2.3** (Substituting types into types commutes with erasure).

$$T[\![\tau[x \mapsto \tau_x]]\!] = T[\![\tau]\!] [x \mapsto T[\![\tau_x]\!]]$$

*Proof.* We use induction on  $\tau$ . We elide the cases where  $x$  does not appear free in  $\tau$ .

VARIABLE:

$$\begin{aligned}
\tau &= x \\
T[\![x[x \mapsto \tau_x]]\!] & \\
&= T[\![\tau_x]\!] \\
&= x[x \mapsto T[\![\tau_x]\!]] \\
&= T[\![x]\!] [x \mapsto T[\![\tau_x]\!]]
\end{aligned}$$

FUNCTION:

$$\begin{aligned}
\tau &= (-> (\tau_i \dots) \tau_o) \\
T[\![(-> (\tau_i \dots) \tau_o)[x \mapsto \tau_x]]\!] & \\
&= T[\![(-> (\tau_i[x \mapsto \tau_x] \dots) \tau_o[x \mapsto \tau_x])]\!] \\
&= (\text{shape}) \\
&= T[\![(-> (\tau_i \dots) \tau_o)\!] [x \mapsto T[\![\tau_x]\!]]
\end{aligned}$$

UNIVERSAL:

$$\tau = (\forall ((x_u k) \dots) \tau_u), \text{ where } x \notin x_u \dots$$

$$\begin{aligned} & \mathcal{T}[(\forall ((x_u k) \dots) \tau_u)[x \mapsto \tau_x]] \\ &= \mathcal{T}[(\forall ((x_u k) \dots) \tau_u[x \mapsto \tau_x])] \\ &= (\text{shape}) \\ &= \mathcal{T}[(\forall ((x_u k) \dots) \tau_u)][x \mapsto \mathcal{T}[\tau_x]] \end{aligned}$$

DEPENDENT PRODUCT:

$$\tau = (\prod ((x_p \gamma) \dots) \tau_p)$$

$$\begin{aligned} & \mathcal{T}[(\prod ((x_p \gamma) \dots) \tau_p)[x \mapsto \tau_x]] \\ &= \mathcal{T}[(\prod ((x_p \gamma) \dots) \tau_p[x \mapsto \tau_x])] \\ &= (\text{shape}) \\ &= \mathcal{T}[(\prod ((x_p \gamma) \dots) \tau_p)][x \mapsto \mathcal{T}[\tau_x]] \end{aligned}$$

DEPENDENT SUM:

$$\tau = (\sum ((x_p \gamma) \dots) \tau_p)$$

$$\begin{aligned} & \mathcal{T}[(\sum ((x_p \gamma) \dots) \tau_p)[x \mapsto \tau_x]] \\ &= \mathcal{T}[(\sum ((x_p \gamma) \dots) \tau_p[x \mapsto \tau_x])] \\ &= (\text{shape}) \\ &= \mathcal{T}[(\sum ((x_p \gamma) \dots) \tau_p)][x \mapsto \mathcal{T}[\tau_x]] \end{aligned}$$

ARRAY:

$$\tau = (A \tau_a \iota)$$

$$\begin{aligned} & \mathcal{T}[(A \tau_a \iota)[x \mapsto \tau_x]] \\ &= \mathcal{T}[(A \tau_a[x \mapsto \tau_x] \iota_a)] \\ &= \iota_a \quad (\text{note: } x \text{ is a type variable and cannot appear in } \iota_a) \\ &= \mathcal{T}[(A \tau_a \iota)][x \mapsto \mathcal{T}[\tau_x]] \end{aligned}$$

□

**Lemma 10.2.4** (Substituting types into terms commutes with erasure).

$$\mathcal{R}[t[x \mapsto \tau_x]] = \mathcal{R}[t][x \mapsto \mathcal{T}[\tau_x]]$$

*Proof.* We use induction on  $t$ . We elide the cases where  $x$  does not appear free in  $t$ .

TERM ABSTRACTION:

$$t = (\lambda ((x_i \tau) \dots) e)$$

$$\begin{aligned}
& \mathcal{R}[(\lambda ((x_i \tau) \dots) e)[x \mapsto \tau_x]] \\
&= \mathcal{A}[(\lambda ((x_i \tau) \dots) e)[x \mapsto \tau_x]] \\
&= \mathcal{A}[(\lambda ((x_i \tau) \dots) e)[x \mapsto \tau_x]] \\
&= (\lambda (x_i \dots) \mathcal{E}[e[x \mapsto \mathcal{T}[\tau_x]]]) \\
&= (\lambda (x_i \dots) \mathcal{E}[e][x \mapsto \mathcal{T}[\tau_x]]) \\
&= (\lambda (x_i \dots) \mathcal{E}[e])[x \mapsto \mathcal{T}[\tau_x]] \\
&= \mathcal{A}[(\lambda ((x_i \tau) \dots) e)][x \mapsto \mathcal{T}[\tau_x]] \\
&= \mathcal{R}[(\lambda ((x_i \tau) \dots) e)][x \mapsto \mathcal{T}[\tau_x]]
\end{aligned}$$

TYPE ABSTRACTION:

$$t = (\top \lambda ((x_i k) \dots) v), \text{ where } x \notin x_i \dots$$

$$\begin{aligned}
& \mathcal{R}[(\top \lambda ((x_i k) \dots) v)[x \mapsto \tau_x]] \\
&= \mathcal{A}[(\top \lambda ((x_i k) \dots) v)[x \mapsto \tau_x]] \\
&= \mathcal{A}[(\top \lambda ((x_i k) \dots) v)[x \mapsto \tau_x]] \\
&= (\top \lambda (x_i \dots) \mathcal{E}[v[x \mapsto \tau_x]]) \\
&= (\top \lambda (x_i \dots) \mathcal{E}[v][x \mapsto \mathcal{T}[\tau_x]]) \\
&= (\top \lambda (x_i \dots) \mathcal{E}[v])[x \mapsto \mathcal{T}[\tau_x]] \\
&= \mathcal{A}[(\top \lambda ((x_i k) \dots) v)][x \mapsto \mathcal{T}[\tau_x]] \\
&= \mathcal{R}[(\top \lambda ((x_i k) \dots) v)][x \mapsto \mathcal{T}[\tau_x]]
\end{aligned}$$

INDEX ABSTRACTION:

$$t = (\text{I} \lambda ((x_i \gamma) \dots) v)$$

$$\begin{aligned}
& \mathcal{R}[(\text{I} \lambda ((x_i \gamma) \dots) v)[x \mapsto \tau_x]] \\
&= \mathcal{A}[(\text{I} \lambda ((x_i \gamma) \dots) v)[x \mapsto \tau_x]] \\
&= \mathcal{A}[(\text{I} \lambda ((x_i \gamma) \dots) v)[x \mapsto \tau_x]] \\
&= (\text{I} \lambda (x_i \dots) \mathcal{E}[v[x \mapsto \tau_x]]) \\
&= (\text{I} \lambda (x_i \dots) \mathcal{E}[v][x \mapsto \mathcal{T}[\tau_x]]) \\
&= (\text{I} \lambda (x_i \dots) \mathcal{E}[v])[x \mapsto \mathcal{T}[\tau_x]] \\
&= \mathcal{A}[(\text{I} \lambda ((x_i k) \dots) v)][x \mapsto \mathcal{T}[\tau_x]] \\
&= \mathcal{R}[(\text{I} \lambda ((x_i k) \dots) v)][x \mapsto \mathcal{T}[\tau_x]]
\end{aligned}$$

BOX:

$$t = (\text{box } \iota \dots e_s \tau)$$

$$\begin{aligned}
& \mathcal{R}[(\text{box } \iota \dots e_s \tau)[x \mapsto \tau_x]] \\
&= \mathcal{A}[(\text{box } \iota \dots e_s \tau)[x \mapsto \tau_x]] \\
&= \mathcal{A}[(\text{box } \iota \dots e_s [x \mapsto \tau_x] \tau [x \mapsto \tau_x])] \\
&= (\text{box } \iota \dots \mathcal{E}[e_s [x \mapsto \tau_x]]) \\
&= (\text{box } \iota \dots \mathcal{E}[e_s][x \mapsto \mathcal{T}[\tau_x]])
\end{aligned}$$



$$\begin{aligned}
&= (\text{box } \iota \dots \mathcal{E}[e_s]) [x \mapsto \mathcal{T}[\tau_x]] \\
&= \mathcal{A}[(\text{box } \iota \dots e_s \tau)] [x \mapsto \mathcal{T}[\tau_x]] \\
&= \mathcal{R}[(\text{box } \iota \dots e_s \tau)] [x \mapsto \mathcal{T}[\tau_x]]
\end{aligned}$$

ARRAY:

$$t = (\text{array } (n \dots) \mathfrak{a} \dots)$$

$$\begin{aligned}
&\mathcal{R}[(\text{array } (n \dots) \mathfrak{a} \dots) [x \mapsto \tau_x]] \\
&= \mathcal{E}[(\text{array } (n \dots) \mathfrak{a} \dots) [x \mapsto \tau_x]] \\
&= \mathcal{E}[(\text{array } (n \dots) \mathfrak{a} [x \mapsto \tau_x] \dots)] \\
&= (\text{array } (n \dots) \mathcal{A}[\mathfrak{a} [x \mapsto \tau_x]] \dots) \\
&= (\text{array } (n \dots) \mathcal{A}[\mathfrak{a}] [x \mapsto \mathcal{T}[\tau_x]] \dots) \\
&= (\text{array } (n \dots) \mathcal{A}[\mathfrak{a}] \dots) [x \mapsto \mathcal{T}[\tau_x]] \\
&= \mathcal{E}[(\text{array } (n \dots) \mathfrak{a} \dots)] [x \mapsto \mathcal{T}[\tau_x]] \\
&= \mathcal{R}[(\text{array } (n \dots) \mathfrak{a} \dots)] [x \mapsto \mathcal{T}[\tau_x]]
\end{aligned}$$

FRAME:

$$t = (\text{frame } (n \dots) e_c \dots)^{\tau_r}$$

$$\begin{aligned}
&\mathcal{R}[(\text{frame } (n \dots) e_c \dots)^{\tau_r} [x \mapsto \tau_x]] \\
&= \mathcal{E}[(\text{frame } (n \dots) e_c \dots)^{\tau_r} [x \mapsto \tau_x]] \\
&= \mathcal{E}[(\text{frame } (n \dots) e_c [x \mapsto \tau_x] \dots)^{\tau_r [x \mapsto \tau_x]}] \\
&= (\text{frame } (\mathcal{T}[\tau_r] [x \mapsto \tau_x]) \mathcal{E}[e_c [x \mapsto \tau_x]] \dots) \\
&= (\text{frame } (\mathcal{T}[\tau_r] [x \mapsto \tau_x]) \mathcal{E}[e_c] [x \mapsto \mathcal{T}[\tau_x]] \dots), \\
&\text{by the induction hypothesis} \\
&= (\text{frame } (\mathcal{T}[\tau_r] [x \mapsto \mathcal{T}[\tau_x]]) \mathcal{E}[e_c] [x \mapsto \mathcal{T}[\tau_x]] \dots), \\
&\text{by Lemma 10.2.3} \\
&= (\text{frame } (\mathcal{T}[\tau_r]) \mathcal{E}[e_c] \dots) [x \mapsto \mathcal{T}[\tau_x]] \\
&= \mathcal{E}[(\text{frame } (n \dots) e_c \dots)^{\tau_r}] [x \mapsto \mathcal{T}[\tau_x]] \\
&= \mathcal{R}[(\text{frame } (n \dots) e_c \dots)^{\tau_r}] [x \mapsto \mathcal{T}[\tau_x]]
\end{aligned}$$

APPLICATION:

$$t = (e_f^{(A \rightarrow (\tau_i \dots) \tau_o) \iota_f} e_a \dots)^{\tau_r}$$

$$\begin{aligned}
&\mathcal{R}[(e_f^{(A \rightarrow (\tau_i \dots) \tau_o) \iota_f} e_a \dots)^{\tau_r} [x \mapsto \tau_x]] \\
&= \mathcal{E}[(e_f^{(A \rightarrow (\tau_i \dots) \tau_o) \iota_f} e_a \dots)^{\tau_r} [x \mapsto \tau_x]] \\
&= \mathcal{E}[(e_f [x \mapsto \tau_x]^{(A \rightarrow (\tau_i \dots) \tau_o) \iota_f} e_a [x \mapsto \tau_x] \dots)^{\tau_r [x \mapsto \tau_x]}] \\
&= \mathcal{E}[(e_f [x \mapsto \tau_x]^{(A \rightarrow (\tau_i [x \mapsto \tau_x] \dots) \tau_o [x \mapsto \tau_x]) \iota_f} e_a [x \mapsto \tau_x] \dots)^{\tau_r [x \mapsto \tau_x]}]
\end{aligned}$$

$$\begin{aligned}
& e_a[x \mapsto \tau_x] \dots \tau_a^{[x \mapsto \tau_x]} \\
= & (\mathcal{E}[e_f[x \mapsto \tau_x]] (\mathcal{E}[e_a[x \mapsto \tau_x]] T[\tau_i[x \mapsto \tau_x]]) \dots T[\tau_r[x \mapsto \tau_x]]) \\
= & (\mathcal{E}[e_f][x \mapsto T[\tau_x]] \\
& (\mathcal{E}[e_a][x \mapsto T[\tau_x]] T[\tau_i[x \mapsto \tau_x]]) \dots \\
& T[\tau_r[x \mapsto \tau_x]]), \text{ by the induction hypothesis} \\
= & (\mathcal{E}[e_f][x \mapsto T[\tau_x]] \\
& (\mathcal{E}[e_a][x \mapsto T[\tau_x]] T[\tau_i[x \mapsto \tau_x]]) \dots \\
& \mathcal{E}[\tau_r][x \mapsto T[\tau_x]]), \text{ by Lemma 10.2.3} \\
= & (\mathcal{E}[e_f] (\mathcal{E}[e_a] T[\tau_i]) \dots \mathcal{E}[\tau_r])[x \mapsto T[\tau_x]] \\
= & \mathcal{E}[(e_f^{(A \rightarrow (\tau_i \dots) \tau_o) \iota_f} e_a \dots)^{\tau_r}][x \mapsto T[\tau_x]] \\
= & \mathcal{R}[(e_f^{(A \rightarrow (\tau_i \dots) \tau_o) \iota_f} e_a \dots)^{\tau_r}][x \mapsto T[\tau_x]]
\end{aligned}$$

TYPE APPLICATION:

$$\begin{aligned}
t &= (t\text{-app } e_f \tau_a \dots)^{\tau_r} \\
\mathcal{R}[(t\text{-app } e_f \tau_a \dots)^{\tau_r}[x \mapsto \tau_x]] \\
= & \mathcal{E}[(t\text{-app } e_f \tau_a \dots)^{\tau_r}[x \mapsto \tau_x]] \\
= & \mathcal{E}[(t\text{-app } e_f[x \mapsto \tau_x] \tau_a[x \mapsto \tau_x] \dots)^{\tau_r[x \mapsto \tau_x]}] \\
= & (i\text{-app } \mathcal{E}[e_f[x \mapsto \tau_x]] T[\tau_a[x \mapsto \tau_x]] \dots T[\tau_r[x \mapsto \tau_x]]) \\
= & (i\text{-app } \mathcal{E}[e_f][x \mapsto T[\tau_x]] T[\tau_a[x \mapsto \tau_x]] \dots T[\tau_r[x \mapsto \tau_x]]), \\
& \text{by the induction hypothesis} \\
= & (i\text{-app } \mathcal{E}[e_f][x \mapsto T[\tau_x]] T[\tau_a][x \mapsto T[\tau_x]] \dots \\
& T[\tau_r][x \mapsto T[\tau_x]]), \\
& \text{by Lemma 10.2.3} \\
= & (i\text{-app } \mathcal{E}[e_f] T[\tau_a] \dots T[\tau_r])[x \mapsto T[\tau_x]] \\
= & \mathcal{E}[(t\text{-app } e_f \tau_a \dots)^{\tau_r}][x \mapsto T[\tau_x]] \\
= & \mathcal{R}[(t\text{-app } e_f \tau_a \dots)^{\tau_r}][x \mapsto T[\tau_x]]
\end{aligned}$$

INDEX APPLICATION:

$$\begin{aligned}
t &= (i\text{-app } e_f \iota_a \dots)^{\tau_r} \\
\mathcal{R}[(i\text{-app } e_f \iota_a \dots)^{\tau_r}[x \mapsto \tau_x]] \\
= & \mathcal{E}[(i\text{-app } e_f \iota_a \dots)^{\tau_r}[x \mapsto \tau_x]] \\
= & \mathcal{E}[(i\text{-app } e_f[x \mapsto \tau_x] \iota_a \dots)^{\tau_r[x \mapsto \tau_x]}] \\
= & (i\text{-app } \mathcal{E}[e_f[x \mapsto \tau_x]] \iota_a \dots T[\tau_r[x \mapsto \tau_x]]) \\
= & (i\text{-app } \mathcal{E}[e_f][x \mapsto T[\tau_x]] \iota_a \dots T[\tau_r[x \mapsto \tau_x]]), \\
& \text{by the induction hypothesis} \\
= & (i\text{-app } \mathcal{E}[e_f][x \mapsto T[\tau_x]] \iota_a \dots T[\tau_r][x \mapsto T[\tau_x]]), \\
& \text{by Lemma 10.2.3} \\
= & (i\text{-app } \mathcal{E}[e_f] \iota_a \dots T[\tau_r])[x \mapsto T[\tau_x]]
\end{aligned}$$

$$\begin{aligned}
&= \mathcal{E}[(i\text{-app } e_f \iota_a \dots)^{\tau_r}][x \mapsto \mathcal{T}[\tau_x]] \\
&= \mathcal{R}[(i\text{-app } e_f \iota_a \dots)^{\tau_r}][x \mapsto \mathcal{T}[\tau_x]]
\end{aligned}$$

UNBOXING:

$$t = (\text{unbox } (x_i \dots x_e e_s) e_b^{\tau_b})$$

$$\begin{aligned}
&\mathcal{R}[(\text{unbox } (x_i \dots x_e e_s) e_b^{\tau_b})[x \mapsto \tau_x]] \\
&= \mathcal{E}[(\text{unbox } (x_i \dots x_e e_s) e_b^{\tau_b})[x \mapsto \tau_x]] \\
&= \mathcal{E}[(\text{unbox } (x_i \dots x_e e_s[x \mapsto \tau_x]) e_b[x \mapsto \tau_x]^{\tau_b[x \mapsto \tau_x]})] \\
&= (\text{unbox } (x_i \dots x_e \mathcal{E}[e_s[x \mapsto \tau_x]]) \mathcal{E}[e_b[x \mapsto \tau_x]] \mathcal{T}[\tau_b[x \mapsto \tau_x]]) \\
&= (\text{unbox } (x_i \dots x_e \mathcal{E}[e_s[x \mapsto \tau_x]]) \mathcal{E}[e_b[x \mapsto \tau_x]] \mathcal{T}[\tau_b][x \mapsto \mathcal{T}[\tau_x]]) \\
&= (\text{unbox } (x_i \dots x_e \mathcal{E}[e_s][x \mapsto \mathcal{T}[\tau_x]]) \\
&\quad \mathcal{E}[e_b][x \mapsto \mathcal{T}[\tau_x]] \mathcal{T}[\tau_b][x \mapsto \mathcal{T}[\tau_x]]) \\
&= (\text{unbox } (x_i \dots x_e \mathcal{E}[e_s]) \mathcal{E}[e_b] \mathcal{T}[\tau_b])[x \mapsto \mathcal{T}[\tau_x]] \\
&= \mathcal{E}[(\text{unbox } (x_i \dots x_e e_s) e_b)][x \mapsto \mathcal{T}[\tau_x]] \\
&= \mathcal{R}[(\text{unbox } (x_i \dots x_e e_s) e_b)][x \mapsto \mathcal{T}[\tau_x]]
\end{aligned}$$

□

**Lemma 10.2.5** (Substituting indices into types commutes with erasure).

$$\mathcal{T}[\tau[x \mapsto \iota_x]] = \mathcal{T}[\tau][x \mapsto \iota_x]$$

*Proof.* We use induction on  $\tau$ . We elide the cases where  $x$  does not appear free in  $\tau$ .

FUNCTION:

$$\tau = (-> (\tau_i \dots) \tau_o)$$

$$\begin{aligned}
&\mathcal{T}[(-> (\tau_i \dots) \tau_o)[x \mapsto \iota_x]] \\
&= \mathcal{T}[(-> (\tau_i[x \mapsto \iota_x] \dots) \tau_o[x \mapsto \iota_x])] \\
&= (\text{shape}) \\
&= (\text{shape})[x \mapsto \iota_x] \\
&= \mathcal{T}[(-> (\tau_i \dots) \tau_o)][x \mapsto \iota_x]
\end{aligned}$$

UNIVERSAL:

$$\tau = (\forall ((x_u k) \dots) \tau_u)$$

$$\begin{aligned}
&\mathcal{T}[(\forall ((x_u k) \dots) \tau_u)[x \mapsto \iota_x]] \\
&= \mathcal{T}[(\forall ((x_u k) \dots) \tau_u[x \mapsto \iota_x])] \\
&= (\text{shape}) \\
&= (\text{shape})[x \mapsto \iota_x] \\
&= \mathcal{T}[(\forall ((x_u k) \dots) \tau_u)][x \mapsto \iota_x]
\end{aligned}$$

DEPENDENT PRODUCT:

$$\begin{aligned}
\tau &= (\Pi ((x_p \gamma) \dots) \tau_p), \text{ where } x \notin x_p \dots \\
\mathcal{T}[(\Pi ((x_p \gamma) \dots) \tau_p)[x \mapsto \iota_x]] & \\
&= \mathcal{T}[(\Pi ((x_p \gamma) \dots) \tau_p[x \mapsto \iota_x])] \\
&= (\text{shape}) \\
&= (\text{shape})[x \mapsto \iota_x] \\
&= \mathcal{T}[(\Pi ((x_p \gamma) \dots) \tau_p)][x \mapsto \iota_x]
\end{aligned}$$

DEPENDENT SUM:

$$\begin{aligned}
\tau &= (\Sigma ((x_p \gamma) \dots) \tau_p), \text{ where } x \notin x_p \dots \\
\mathcal{T}[(\Sigma ((x_p \gamma) \dots) \tau_p)[x \mapsto \iota_x]] & \\
&= \mathcal{T}[(\Sigma ((x_p \gamma) \dots) \tau_p[x \mapsto \iota_x])] \\
&= (\text{shape}) \\
&= (\text{shape})[x \mapsto \iota_x] \\
&= \mathcal{T}[(\Sigma ((x_p \gamma) \dots) \tau_p)][x \mapsto \iota_x]
\end{aligned}$$

ARRAY:

$$\begin{aligned}
\tau &= (A \tau_a \iota) \\
\mathcal{T}[(A \tau_a \iota)[x \mapsto \iota_x]] & \\
&= \mathcal{T}[(A \tau_a[x \mapsto \iota_x] \iota[x \mapsto \iota_x])] \\
&= \iota[x \mapsto \iota_x] \\
&= \mathcal{T}[(A \tau_a \iota)][x \mapsto \iota_x]
\end{aligned}$$

□

**Lemma 10.2.6** (Substituting indices into terms commutes with erasure).  
 $\mathcal{R}[t[x \mapsto \iota_x]] = \mathcal{R}[t][x \mapsto \iota_x]$

*Proof.* We use induction on  $t$ . We elide the cases where  $x$  does not appear free in  $t$ .

TERM ABSTRACTION:

$$\begin{aligned}
t &= (\lambda ((x_i \tau) \dots) e) \\
\mathcal{R}[(\lambda ((x_i \tau) \dots) e)[x \mapsto \iota_x]] & \\
&= \mathcal{A}[(\lambda ((x_i \tau) \dots) e)[x \mapsto \iota_x]] \\
&= \mathcal{A}[(\lambda ((x_i \tau) \dots) e[x \mapsto \iota_x])] \\
&= (\lambda (x_i \dots) \mathcal{E}[e[x \mapsto \iota_x]]) \\
&= (\lambda (x_i \dots) \mathcal{E}[e][x \mapsto \iota_x]) \\
&= (\lambda (x_i \dots) \mathcal{E}[e])[x \mapsto \iota_x] \\
&= \mathcal{A}[(\lambda (x_i \dots) e)][x \mapsto \iota_x] \\
&= \mathcal{R}[(\lambda (x_i \dots) e)][x \mapsto \iota_x]
\end{aligned}$$

TYPE ABSTRACTION:

$$t = (\text{T}\lambda ((x_i k) \dots) v)$$

Note:  $x$  is an index variable, while the  $x_i \dots$  are type variables, so they do not shadow.  $\mathcal{R}[(\text{T}\lambda ((x_i k) \dots) v)[x \mapsto \iota_x]]$

$$\begin{aligned} &= \mathcal{A}[(\text{T}\lambda ((x_i k) \dots) v)[x \mapsto \iota_x]] \\ &= \mathcal{A}[(\text{T}\lambda ((x_i k) \dots) v[x \mapsto \iota_x])] \\ &= (\text{I}\lambda (x_i \dots) \mathcal{E}[v[x \mapsto \iota_x]]) \\ &= (\text{I}\lambda (x_i \dots) \mathcal{E}[v][x \mapsto \iota_x]) \\ &= (\text{I}\lambda (x_i \dots) \mathcal{E}[v])[x \mapsto \iota_x] \\ &= \mathcal{A}[(\text{T}\lambda ((x_i k) \dots) v)][x \mapsto \iota_x] \\ &= \mathcal{R}[(\text{T}\lambda ((x_i k) \dots) v)][x \mapsto \iota_x] \end{aligned}$$

INDEX ABSTRACTION:

$$t = (\text{I}\lambda ((x_i \gamma) \dots) v), \text{ where } x \notin x_i \dots$$

$$\begin{aligned} &\mathcal{R}[(\text{I}\lambda ((x_i \gamma) \dots) v)[x \mapsto \iota_x]] \\ &= \mathcal{A}[(\text{I}\lambda ((x_i \gamma) \dots) v)[x \mapsto \iota_x]] \\ &= \mathcal{A}[(\text{I}\lambda ((x_i \gamma) \dots) v[x \mapsto \iota_x])] \\ &= (\text{I}\lambda (x_i \dots) \mathcal{E}[v[x \mapsto \iota_x]]) \\ &= (\text{I}\lambda (x_i \dots) \mathcal{E}[v][x \mapsto \iota_x]) \\ &= (\text{I}\lambda (x_i \dots) \mathcal{E}[v])[x \mapsto \iota_x] \\ &= \mathcal{A}[(\text{I}\lambda ((x_i \gamma) \dots) v)][x \mapsto \iota_x] \\ &= \mathcal{R}[(\text{I}\lambda ((x_i \gamma) \dots) v)][x \mapsto \iota_x] \end{aligned}$$

BOX:

$$t = (\text{box } \iota \dots e_s \tau)$$

$$\begin{aligned} &\mathcal{R}[(\text{box } \iota \dots e_s \tau)[x \mapsto \iota_x]] \\ &= \mathcal{A}[(\text{box } \iota \dots e_s \tau)[x \mapsto \iota_x]] \\ &= \mathcal{A}[(\text{box } \iota[x \mapsto \iota_x] \dots e_s[x \mapsto \iota_x] \tau[x \mapsto \iota_x])] \\ &= (\text{box } \iota[x \mapsto \iota_x] \dots \mathcal{E}[e_s[x \mapsto \iota_x]]) \\ &= (\text{box } \iota[x \mapsto \iota_x] \dots \mathcal{E}[e_s][x \mapsto \iota_x]) \\ &= (\text{box } \iota \dots \mathcal{E}[e_s])[x \mapsto \iota_x] \\ &= \mathcal{A}[(\text{box } \iota \dots e_s \tau)][x \mapsto \iota_x] \\ &= \mathcal{R}[(\text{box } \iota \dots e_s \tau)][x \mapsto \iota_x] \end{aligned}$$

ARRAY:

$$t = (\text{array } (n \dots) \mathbf{a} \dots)$$

$$\begin{aligned} &\mathcal{R}[(\text{array } (n \dots) \mathbf{a} \dots)[x \mapsto \iota_x]] \\ &= \mathcal{E}[(\text{array } (n \dots) \mathbf{a} \dots)[x \mapsto \iota_x]] \end{aligned}$$

$$\begin{aligned}
&= \mathcal{E}[(\text{array } (n \dots) \mathfrak{a}[x \mapsto \iota_x] \dots)] \\
&= (\text{array } (n \dots) \mathcal{A}[\mathfrak{a}[x \mapsto \iota_x]] \dots) \\
&= (\text{array } (n \dots) \mathcal{A}[\mathfrak{a}][x \mapsto \iota_x] \dots) \\
&= (\text{array } (n \dots) \mathcal{A}[\mathfrak{a}] \dots)[x \mapsto \iota_x] \\
&= \mathcal{E}[(\text{array } (n \dots) \mathfrak{a} \dots)][x \mapsto \iota_x] \\
&= \mathcal{R}[(\text{array } (n \dots) \mathfrak{a} \dots)][x \mapsto \iota_x]
\end{aligned}$$

FRAME:

$$\begin{aligned}
t &= (\text{frame } (n \dots) e_c \dots)^{\tau_r} \\
\mathcal{R}[(\text{frame } (n \dots) e_c \dots)^{\tau_r}[x \mapsto \iota_x]] \\
&= \mathcal{E}[(\text{frame } (n \dots) e_c \dots)^{\tau_r}[x \mapsto \iota_x]] \\
&= \mathcal{E}[(\text{frame } (n \dots) e_c[x \mapsto \iota_x] \dots)^{\tau_r[x \mapsto \iota_x]}] \\
&= (\text{frame } (\mathcal{T}[\tau_r[x \mapsto \iota_x]]) \mathcal{E}[e_c[x \mapsto \iota_x]] \dots) \\
&= (\text{frame } (\mathcal{T}[\tau_r[x \mapsto \iota_x]]) \mathcal{E}[e_c][x \mapsto \iota_x] \dots), \text{ by the induction hypothesis} \\
&\text{esis} \\
&= (\text{frame } (\mathcal{T}[\tau_r][x \mapsto \iota_x]) \mathcal{E}[e_c][x \mapsto \iota_x] \dots), \text{ by Lemma 10.2.5} \\
&= (\text{frame } (\mathcal{T}[\tau_r]) \mathcal{E}[e_c] \dots)[x \mapsto \iota_x] \\
&= \mathcal{E}[(\text{frame } (n \dots) e_c \dots)^{\tau_r}][x \mapsto \iota_x] \\
&= \mathcal{R}[(\text{frame } (n \dots) e_c \dots)^{\tau_r}][x \mapsto \iota_x]
\end{aligned}$$

APPLICATION:

$$\begin{aligned}
t &= (e_f^{(A \rightarrow (\tau_i \dots) \tau_o) \iota_f} e_a \dots)^{\tau_r} \\
\mathcal{R}[(e_f^{(A \rightarrow (\tau_i \dots) \tau_o) \iota_f} e_a \dots)^{\tau_r}[x \mapsto \iota_x]] \\
&= \mathcal{E}[(e_f^{(A \rightarrow (\tau_i \dots) \tau_o) \iota_f} e_a \dots)^{\tau_r}[x \mapsto \iota_x]] \\
&= \mathcal{E}[(e_f[x \mapsto \iota_x]^{(A \rightarrow (\tau_i[x \mapsto \iota_x] \dots) \tau_o[x \mapsto \iota_x]) \iota_f[x \mapsto \iota_x]}) \\
&\quad e_a[x \mapsto \iota_x] \dots)^{\tau_r[x \mapsto \iota_x]}] \\
&= (\mathcal{E}[e_f[x \mapsto \iota_x]] (\mathcal{E}[e_a[x \mapsto \iota_x]] \mathcal{T}[\tau_i[x \mapsto \iota_x]] \dots \mathcal{T}[\tau_r[x \mapsto \iota_x]])) \\
&= (\mathcal{E}[e_f][x \mapsto \iota_x] (\mathcal{E}[e_a][x \mapsto \iota_x] \mathcal{T}[\tau_i[x \mapsto \iota_x]] \dots \mathcal{T}[\tau_r[x \mapsto \iota_x]])), \\
&\text{by the induction hypothesis} \\
&= (\mathcal{E}[e_f][x \mapsto \iota_x] (\mathcal{E}[e_a][x \mapsto \iota_x] \mathcal{T}[\tau_i][x \mapsto \iota_x] \dots \mathcal{T}[\tau_r][x \mapsto \iota_x])), \\
&\text{by Lemma 10.2.3} \\
&= (\mathcal{E}[e_f] (\mathcal{E}[e_a] \mathcal{T}[\tau_i]) \dots \mathcal{T}[\tau_r])[x \mapsto \iota_x] \\
&= \mathcal{E}[(e_f^{(A \rightarrow (\tau_i \dots) \tau_o) \iota_f} e_a \dots)^{\tau_r}][x \mapsto \iota_x] \\
&= \mathcal{R}[(e_f^{(A \rightarrow (\tau_i \dots) \tau_o) \iota_f} e_a \dots)^{\tau_r}][x \mapsto \iota_x]
\end{aligned}$$

TYPE APPLICATION:

$$t = (\text{t-app } e_f \tau_a \dots)^{\tau_r}$$

$$\begin{aligned}
& \mathcal{R}[(\text{t-app } e_f \tau_a \dots)^{\tau_r}[x \mapsto \iota_x]] \\
&= \mathcal{E}[(\text{t-app } e_f \tau_a \dots)^{\tau_r}[x \mapsto \iota_x]] \\
&= \mathcal{E}[(\text{t-app } e_f[x \mapsto \iota_x] \tau_a[x \mapsto \iota_x] \dots)^{\tau_r[x \mapsto \iota_x]}] \\
&= (\text{i-app } \mathcal{E}[e_f[x \mapsto \iota_x]] \ T[\tau_a[x \mapsto \iota_x]] \ \dots \ T[\tau_r[x \mapsto \iota_x]]) \\
&= (\text{i-app } \mathcal{E}[e_f][x \mapsto \iota_x] \ T[\tau_a[x \mapsto \iota_x]] \ \dots \ T[\tau_r[x \mapsto \iota_x]]), \\
&\text{by the induction hypothesis} \\
&= (\text{i-app } \mathcal{E}[e_f][x \mapsto \iota_x] \ T[\tau_a][x \mapsto \iota_x] \ \dots \ T[\tau_r][x \mapsto \iota_x]), \\
&\text{by Lemma 10.2.3} \\
&= (\text{i-app } \mathcal{E}[e_f] \ T[\tau_a] \ \dots \ T[\tau_r])[x \mapsto \iota_x] \\
&= \mathcal{E}[(\text{t-app } e_f \tau_a \dots)^{\tau_r}[x \mapsto \iota_x] \\
&= \mathcal{R}[(\text{t-app } e_f \tau_a \dots)^{\tau_r}[x \mapsto \iota_x]
\end{aligned}$$

INDEX APPLICATION:

$$t = (\text{i-app } e_f \iota_a \dots)^{\tau_r}$$

$$\begin{aligned}
& \mathcal{R}[(\text{i-app } e_f \iota_a \dots)^{\tau_r}[x \mapsto \iota_x]] \\
&= \mathcal{E}[(\text{i-app } e_f \iota_a \dots)^{\tau_r}[x \mapsto \iota_x]] \\
&= \mathcal{E}[(\text{i-app } e_f[x \mapsto \iota_x] \ \iota_a[x \mapsto \iota_x] \ \dots)^{\tau_r[x \mapsto \iota_x]}] \\
&= (\text{i-app } \mathcal{E}[e_f[x \mapsto \iota_x]] \ \iota_a[x \mapsto \iota_x] \ \dots \ T[\tau_r[x \mapsto \iota_x]]) \\
&= (\text{i-app } \mathcal{E}[e_f][x \mapsto \iota_x] \ \iota_a[x \mapsto \iota_x] \ \dots \ T[\tau_r[x \mapsto \iota_x]]), \\
&\text{by the induction hypothesis} \\
&= (\text{i-app } \mathcal{E}[e_f][x \mapsto \iota_x] \ \iota_a[x \mapsto \iota_x] \ \dots \ T[\tau_r][x \mapsto \iota_x]), \\
&\text{by Lemma 10.2.3} \\
&= (\text{i-app } \mathcal{E}[e_f] \ \iota_a \ \dots \ T[\tau_r])[x \mapsto \iota_x] \\
&= \mathcal{E}[(\text{i-app } e_f \iota_a \dots)^{\tau_r}[x \mapsto \iota_x] \\
&= \mathcal{R}[(\text{i-app } e_f \iota_a \dots)^{\tau_r}[x \mapsto \iota_x]
\end{aligned}$$

UNBOXING:

$$t = (\text{unbox } (x_i \dots x_e e_s) e_b^{\tau_b}), \text{ where } x \notin x_i \dots$$

$$\begin{aligned}
& \mathcal{R}[(\text{unbox } (x_i \dots x_e e_s) e_b^{\tau_b})[x \mapsto \iota_x]] \\
&= \mathcal{E}[(\text{unbox } (x_i \dots x_e e_s) e_b^{\tau_b})[x \mapsto \iota_x]] \\
&= \mathcal{E}[(\text{unbox } (x_i \dots x_e e_s[x \mapsto \iota_x]) \ e_b[x \mapsto \iota_x]^{\tau_b[x \mapsto \iota_x]})] \\
&= (\text{unbox } (x_i \dots x_e \ \mathcal{E}[e_s[x \mapsto \iota_x]]) \ \mathcal{E}[e_b[x \mapsto \iota_x]] \ T[\tau_b[x \mapsto \iota_x]]) \\
&= (\text{unbox } (x_i \dots x_e \ \mathcal{E}[e_s][x \mapsto \iota_x]) \ \mathcal{E}[e_b[x \mapsto \iota_x]] \ T[\tau_b][x \mapsto \iota_x]) \\
&= (\text{unbox } (x_i \dots x_e \ \mathcal{E}[e_s][x \mapsto \iota_x]) \ \mathcal{E}[e_b][x \mapsto \iota_x] \ T[\tau_b][x \mapsto \iota_x]) \\
&= (\text{unbox } (x_i \dots x_e \ \mathcal{E}[e_s]) \ \mathcal{E}[e_b] \ T[\tau_b])[x \mapsto \iota_x] \\
&= \mathcal{E}[(\text{unbox } (x_i \dots x_e e_s) e_b^{\tau_b})][x \mapsto \iota_x] \\
&= \mathcal{R}[(\text{unbox } (x_i \dots x_e e_s) e_b^{\tau_b})][x \mapsto \iota_x]
\end{aligned}$$

UNBOXING, WITH SHADOWED VARIABLE:

$$\begin{aligned}
t &= (\text{unbox } (x_i \dots x_e e_s) e_b^{\tau_b}), \text{ where } x \in x_i \dots \\
\mathcal{R}[(\text{unbox } (x_i \dots x_e e_s) e_b^{\tau_b})[x \mapsto \iota_x]] & \\
&= \mathcal{E}[(\text{unbox } (x_i \dots x_e e_s) e_b^{\tau_b})[x \mapsto \iota_x]] \\
&= \mathcal{E}[(\text{unbox } (x_i \dots x_e e_s[x \mapsto \iota_x]) e_b^{\tau_b})] \\
&= (\text{unbox } (x_i \dots x_e \mathcal{E}[e_s[x \mapsto \iota_x]]) \mathcal{E}[e_b] \mathcal{T}[\tau_b]) \\
&= (\text{unbox } (x_i \dots x_e \mathcal{E}[e_s][x \mapsto \iota_x]) \mathcal{E}[e_b] \mathcal{T}[\tau_b]) \\
&= (\text{unbox } (x_i \dots x_e \mathcal{E}[e_s]) \mathcal{E}[e_b] \mathcal{T}[\tau_b])[x \mapsto \iota_x] \\
&= \mathcal{E}[(\text{unbox } (x_i \dots x_e e_s) e_b^{\tau_b})][x \mapsto \iota_x] \\
&= \mathcal{R}[(\text{unbox } (x_i \dots x_e e_s) e_b^{\tau_b})][x \mapsto \iota_x]
\end{aligned}$$

□

**Lemma 10.2.7** (Values erase to values). *For any well-typed term  $t$ ,*

- *If  $t$  has the form  $v$ , then  $\mathcal{R}[t]$  has the form  $\hat{v}$*
- *If  $t$  has the form  $v$ , then  $\mathcal{R}[t]$  has the form  $\hat{v}$*

*Proof.* We use induction on  $t$ . The result is trivial for all atom cases except for boxes, so we have only two cases left to consider.

BOX:

$$t = (\text{box } \iota \dots v \tau)$$

According to the grammar of Remora,  $v$  must have the form (array ( $n \dots$ )  $v \dots$ ). Then  $\mathcal{R}[t] = \mathcal{A}[(\text{box } \iota \dots v \tau)] = (\text{box } \iota \dots \mathcal{E}[v])$ . By the induction hypothesis,  $\mathcal{E}[v]$  has the form  $\hat{v}$ , so  $\mathcal{R}[t]$  has the form  $(\text{box } \iota \dots \hat{v})$ , which is a valid  $\hat{v}$  form.

ARRAY LITERAL:

$$t = (\text{array } (n \dots) v \dots)$$

The induction hypothesis implies that for each  $v_i \in v \dots$ ,  $\mathcal{A}[v_i]$  produces an erased atomic value  $\hat{v}_i$ . Therefore the erased term  $\mathcal{R}[t] = \mathcal{E}[(\text{array } (n \dots) v \dots)] = (\text{array } (n \dots) \mathcal{A}[v] \dots)$  must have the form  $(\text{array } (n \dots) \hat{v} \dots)$ . □

**Lemma 10.2.8** (Lockstep). *For any well-typed  $e$ , one of the following holds:*

- *$e$  has the form  $v$ , and  $\mathcal{E}[e]$  has the form  $\hat{v}$*
- *$e \mapsto e'$ , and  $\mathcal{E}[e] \mapsto \mathcal{E}[e']$*
- *$e \not\mapsto$ , and  $\mathcal{E}[e] \not\mapsto$*

*Proof.* We prove this by induction on  $e$ . Note that if  $e$  is not itself a redex or a value form, then the progress lemma implies that it must be an evaluation context filled with a redex.



VALUE: This case is exactly the expression case of Lemma 10.2.7.

REDEX WITHIN NONTRIVIAL EVALUATION CONTEXT:

$$e = \mathbb{V}[e_r], \text{ where } e_r \mapsto e'_r$$

Then  $e \mapsto e' = \mathbb{V}[e'_r]$ . By Lemma 10.2.1 (erasure in context), we have  $\mathcal{E}[\mathbb{V}[e_r]] = \mathcal{C}[\mathbb{V}[\mathcal{E}[e_r]]]$ . The induction hypothesis implies that  $\mathcal{E}[e_r] \mapsto \mathcal{E}[e'_r]$ , so the full erased expression  $\mathcal{C}[\mathbb{V}[\mathcal{E}[e_r]]] \mapsto \mathcal{C}[\mathbb{V}[\mathcal{E}[e'_r]]]$ . Erasure in context gives us  $\mathcal{C}[\mathbb{V}[\mathcal{E}[e'_r]]] = \mathcal{E}[\mathbb{V}[e'_r]]$ . Therefore  $\mathcal{E}[\mathbb{V}[e_r]] \mapsto \mathcal{E}[\mathbb{V}[e'_r]]$ .

LIFT REDEX:

$$e = ((\text{array } (n_f \dots) \mathbf{v}_f \dots)^{(A \tau_f (\text{shape } n_f \dots))} \\ (\text{array } (n_a \dots n_i \dots) \mathbf{v}_a \dots)^{(A \tau_i (\text{shape } n_a \dots n_i \dots))} \\ \dots)^{(A \tau_o (\text{shape } n_p \dots n_o \dots))}$$

$\mapsto \text{lift}$

$$e' = ((\text{array } (n_p \dots) \\ \text{Concat} \left[ \text{Rep}_{n_{fe}} \left[ \text{Split}_1 \left[ \mathbf{v}_f \dots \right] \right] \right])^{(A \tau_f (\text{shape } n_p \dots))} \\ (\text{array } (n_p \dots n_i \dots) \\ \text{Concat} \left[ \text{Rep}_{n_{ae}} \left[ \text{Split}_{n_{ac}} \left[ \mathbf{v}_a \dots \right] \right] \right])^{(A \tau_i (\text{shape } n_p \dots n_i \dots))} \\ \dots)^{(A \tau_o (\text{shape } n_p \dots n_o \dots))}$$

where  $\tau_f = (-> ((A \tau_i (\text{shape } n_i \dots)) \dots) (A \tau_o t_o))$ . Then

$$\mathcal{E}[e] = ((\text{array } (n_f \dots) \mathcal{A}[\mathbf{v}_f] \dots) \\ ((\text{array } (n_a \dots n_i \dots) \mathcal{A}[\mathbf{v}_a] \dots) (\text{shape } n_i \dots)) \\ \dots (\text{shape } n_p \dots n_o \dots))$$

This is a *lift* redex in Erased Remora, and it steps to

$$\mathcal{E}[e'] = ((\text{array } (n_p \dots) \text{Concat} \left[ \text{Rep}_{n_{fe}} \left[ \text{Split}_1 \left[ \mathcal{A}[\mathbf{v}_f] \dots \right] \right] \right]) \\ ((\text{array } (n_p \dots n_i \dots) \\ (\text{Concat} \left[ \text{Rep}_{n_{ae}} \left[ \text{Split}_{n_{ac}} \left[ \mathcal{A}[\mathbf{v}_a] \dots \right] \right] \right]) \\ (\text{shape } n_i \dots)) \\ \dots (\text{shape } n_p \dots n_o \dots))$$

That is,  $\mathcal{E}[e] \mapsto_{\text{lift}} \mathcal{E}[e']$ .

MAP REDEX:

$$e = ((\text{array } (n_f \dots) \mathbf{v}_f \dots)^{(A \tau_f (\text{shape } n_f \dots))} \\ (\text{array } (n_f \dots n_i \dots) \mathbf{v}_a \dots)^{(A \tau_i (\text{shape } n_f \dots n_i \dots))} \\ \dots)^{(A \tau_o (\text{shape } n_f \dots n_o \dots))}$$

$\mapsto_{map}$

$$e' = (\text{frame } (n_f \dots) \\ ((\text{array } () \mathbf{v}_f)^{(A \ (-> \ ((A \ \tau_i \ (\text{shape } n_i \dots)) \dots) \ (A \ \tau_o \ \iota_o)) \ (\text{shape}))}) \\ (\text{array } () \mathbf{v}_c \dots)^{(A \ \tau_i \ (\text{shape } n_i \dots))} \\ \dots)^{(A \ \tau_o \ (\text{shape } n_o \dots))} \\ \dots)^{(A \ \tau_o \ (\text{shape } n_f \dots n_o \dots))})$$

The argument cells' atoms  $((\mathbf{v}_c \dots) \dots)$  are given by

$$\text{Transpose} \left[ \left[ \text{Split}_{n_c} \left[ \left[ \mathbf{v}_a \dots \right] \dots \right] \right]$$

where each  $n_c$  is computed as the product of the corresponding position's expected argument dimensions  $n_i \dots$ , as in Figure 4.11. We then consider the erased form of  $e$ :

$$\mathcal{E}[e] = ((\text{array } (n_f \dots) \mathcal{A}[\mathbf{v}_f] \dots) \\ ((\text{array } (n_f \dots n_i \dots) \mathcal{A}[\mathbf{v}_a] \dots) \ (\text{shape } n_i \dots)) \dots \\ (\text{shape } n_p \dots n_o \dots))$$

This is a *map* redex in Erased Remora. The argument atoms

$$((\mathcal{A}[\mathbf{v}_a] \dots) \dots)$$

are split up into cells in the same way with

$$((\widehat{\mathbf{v}}_c \dots) \dots) = ((\mathcal{A}[\mathbf{v}_c] \dots) \dots)$$

as the result of  $\text{Transpose} \left[ \left[ \text{Split}_{n_c} \left[ \left[ \mathcal{A}[\mathbf{v}_a] \dots \right] \dots \right] \right]$ . So  $\mathcal{E}[e]$  steps to

$$\mathcal{E}[e'] = (\text{frame } (n_f \dots) \\ ((\text{array } () \mathcal{A}[\mathbf{v}_f]) \\ ((\text{array } () \mathcal{A}[\mathbf{v}_c] \dots) \ (\text{shape } n_i \dots)) \\ \dots) \\ \dots)$$

**BETA REDEX:** Note that  $\tau_I$  here must be  $(A \ \tau_i \ (\text{shape } n_a \dots))$ .

$$e = ((\text{array } () \ (\lambda \ ((x \ \tau_i) \dots) \ e_b))^{(A \ (-> \ (\tau_I \dots) \ \tau_o) \ (\text{shape}))}) \\ (\text{array } (n_a \dots) \ \mathbf{v}_a \dots)^{\tau_I} \\ \dots)^{\tau_o}$$

$\mapsto_{\beta}$

$$e' = e_b[x \mapsto (\text{array } (n_a \dots) \ \mathbf{v}_a \dots)^{\tau_I}, \dots]$$

Then

$$\mathcal{E}[e] = ((\text{array } () (\lambda (x \dots) \mathcal{E}[e_b])) \\ ((\text{array } (n_a \dots) \mathcal{A}[\mathbf{v}_a] \dots) (\text{shape } n_a \dots)) \\ \dots)^{\tau_0}$$

$$\mapsto_{\beta} \mathcal{E}[e_b][x \mapsto (\text{array } (n_a \dots) \mathcal{A}[\mathbf{v}_a] \dots), \dots]$$

By Lemma 10.2.2 (substitution commutes with erasure), this result term is equal to  $\mathcal{E}[e']$ .

I-BETA REDEX:

$$e = (\text{i-app} \\ (\text{array } (n_f \dots) \\ (\text{I}\lambda ((x \gamma)) e_b) \dots)^{(\Lambda (\Pi ((x \gamma) \dots) \tau_b) (\text{shape } n_f \dots))} \\ \iota_a \dots)^{\tau_R}$$

$$\mapsto_{i\beta}$$

$$e' = (\text{frame } (n_f \dots) e_b[x \mapsto \iota_a, \dots] \dots)^{\tau_R}$$

Then

$$\mathcal{E}[e] = (\text{i-app } (\text{array } (n_f \dots) (\text{I}\lambda (x) \mathcal{E}[e_b]) \dots) \iota_a \dots \mathcal{T}[\tau_R])$$

$$\mapsto_{i\beta} (\text{frame } (n_f \dots) \mathcal{E}[e_b][x \mapsto \iota_a, \dots] \dots)^{\tau_R} = \mathcal{E}[e']$$

T-BETA REDEX:

$$e = (\text{t-app } (\text{array } (n_f \dots) \\ (\text{T}\lambda ((x k)) e_b) \dots)^{(\Lambda (\forall ((x k) \dots) \tau_b) (\text{shape } n_f \dots))} \\ \tau_a \dots)^{\tau_R}$$

$$\mapsto_{t\beta}$$

$$e' = (\text{frame } (n_f \dots) e_b[x \mapsto \tau_a, \dots] \dots)^{\tau_R}$$

Recall that type abstraction and application erase to index abstraction and application. So in the erased language, we have:

$$\mathcal{E}[e] = (\text{i-app } (\text{array } (n_f \dots) (\text{I}\lambda (x) \mathcal{E}[e_b]) \dots) \tau_a \dots \mathcal{T}[\tau_R])$$

$$\mapsto_{i\beta} (\text{frame } (n_f \dots) \mathcal{E}[e_b][x \mapsto \tau_a, \dots] \dots)^{\tau_R} = \mathcal{E}[e']$$

UNBOX REDEX:

$$e = (\text{unbox } (x_i \dots x_e (\text{array } (n_s \dots) (\text{box } \iota_s \dots v_s \tau_s) \dots)) \\ e_b^{\tau_B})^{\tau_R}$$

$$\mapsto_{\text{unbox}}$$

$$e' = (\text{frame } (n_s \dots) e_b[x_i \mapsto v_s, \dots, x_e \mapsto v_s] \dots)^{\tau_R}$$

Then relying on our earlier result that erasure commutes with substitution, we can take a reduction step on the erased version of  $e$  to get the erased version of  $e'$ :

$$\mathcal{E}[e] = (\text{unbox } (x_i \dots x_e (\text{array } (n_s \dots) (\text{box } v_s \dots \mathcal{E}[v_s]) \dots)) \mathcal{E}[e_b] \mathcal{T}[\tau_B])$$

$\mapsto_{\text{unbox}}$

$$\begin{aligned} & (\text{frame } ((++ (\text{shape } n_s \dots) \mathcal{T}[\tau_B])) \\ & \quad \mathcal{E}[e_b][x_i \mapsto v_s, \dots, x_e \mapsto \mathcal{E}[v_s]] \dots) \\ & = (\text{frame } ((++ (\text{shape } n_s) \mathcal{T}[\tau_B])) \\ & \quad \mathcal{E}[e_b[x_i \mapsto v_s, \dots, x_e \mapsto v_s]] \dots) \\ & = \mathcal{E}[e'] \end{aligned}$$

**MIS-APPLIED PRIMITIVE OPERATOR:** The remaining case is that  $e$  is not reducible (*i.e.*,  $e \not\mapsto$ ), but  $e$  is still not a value. Since  $e$  is well-typed, Lemma 4.4.1 (Progress) implies that  $e$  must be a mis-application of a primitive operator. That is,  $e$  has the form  $\mathbb{V}[(\text{array } () \mathfrak{o}) v \dots]$ , where  $\mathcal{S}[\mathfrak{o}]$  is  $(\rightarrow (\tau_I \dots) \tau_O)$ , and the types of  $v \dots$  are  $\tau_i \dots$ . Using Lemma 10.2.1 (Erasure in context),  $\mathcal{E}[e]$  is  $\mathcal{C}[\mathbb{V}][(\mathfrak{o} (\mathcal{E}[v] \mathcal{T}[\tau_I]) \dots \mathcal{T}[\tau_r])]$ . Since each  $v$  has shape  $\mathcal{T}[\tau_i]$ , we still have function application with a scalar frame, and the values given as arguments are still out-of-domain for  $\mathfrak{o}$ .  $\square$