Static Analysis for Syntax Objects

David Fisher
Georgia Institute of Technology
dfisher@cc.gatech.edu

Olin Shivers
Northeastern University
shivers@ccs.neu.edu

Abstract

We describe an s-expression based syntax-extension framework much like Scheme macros, with a key additional facility: the ability to define static semantics, such as type systems or program analysis, for the new, user-defined forms or embedded languages, thus allowing us to construct "towers" of language levels. In addition, the static semantics of the languages at two adjacent levels in the tower can be connected, allowing improved reasoning power at a higher (and perhaps more restricted) level to be reflected down to the static semantics of the language level below. We demonstrate our system by designing macros for an assembly language, together with some example static analyses (termination analysis, type inference and control-flow analysis).

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—extensible languages, macro and assembly languages; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—program analysis; D.3.3 [*Programming Languages*]: Language Constructs and Features—control structures, classes and objects, constraints, frameworks, inheritance, polymorphism

General Terms Languages, Design

Keywords Extensible programming languages, domain-specific languages, macros, language towers, static analysis, type inference, flow analysis, lazy delegation

1. Scheme macros: strengths and limitations

The most successful and widely employed extensible-syntax system in use today, by a large margin, is the LISP family's macro facility, including the "hygienic" macros developed for Scheme. It is a standard part of the everyday programming task for Lisp and Scheme programmers to define custom notations that permit their domain of programs to be expressed clearly and concisely, whether that task be data-base queries, string searching, Unix scripts, text parsing, VLSI design or airline reservation queries.

Once one has become accustomed to such a powerful tool, it is hard to give it up. When we find ourselves writing programs in languages such as Java, SML, or C—languages, that is, that lack Scheme's syntax-extension ability—we find that we miss it greatly. We'd love to be able to "peel" Scheme's macro system away from the rest of the language and apply it as a general-purpose syntactic front end to other languages.

Unfortunately, this is not a straightforward task. The are two key issues that bind the design of the Scheme macro system tightly to Scheme:

- **Focus on expressions**
  Scheme's system only allows the programmer to invoke macros in expression contexts. In the context of Scheme, this is not much of a hindrance, because Scheme has such a spare syntax: there is very little text in a Scheme program that is *not* an expression. However, if, for example, one wanted to define a macro that would be invoked in the parameter list of a λ form, or could be used in place of a `(`*var exp*`)` binding clause in a `let` form, one could not use Scheme's expression macros.
- **Static semantics**
  A Scheme macro is essentially "specification by compiler:" we define the meaning of a piece of syntax by providing a translation to a base language (that is, to core Scheme). As semanticists, we might quibble with this, but it works well as an engineering solution to define dynamic semantics in this way. However, Scheme macros do not provide a way to define static semantics *directly in terms of* the new form. Instead, the static semantics gets "dragged along" with the dynamic semantics by virtue of the translation.

  This is not a problem in Scheme because it has such a spare static semantics: with no type system, all Scheme really provides by way of static semantics is a lexical-scoping discipline for resolving names. So it is no accident that the "hygiene" facility of Scheme macros provide exactly the ability to statically (that is, at macro-execution time) manipulate name resolution—and no more.

The latter issue is the more serious barrier. Imagine that we did, somehow, manage to construct a Scheme-like macro system for programming in C. That is, we would define an s-expression concrete syntax for C, where one might write "`(if` *exp stmt stmt*`)`" instead of the usual "`if (`*exp*`) then` *stmt* `else` *stmt*" form. We would add to this language the ability to define macros, tagged with source-to-source expanders written in Scheme. The C compiler would be altered to read in the new syntax, and it would contain a Scheme interpreter to use in executing macro expansions at compile time.

How would we type-check a program written in such a language? We would have to first macro-expand the program into base C and then check that code. This would work, in the sense of being correct, but it would be a disaster in practice. When a programmer made a type error, the error would be reported, not in terms of the program originally written, but in terms of its expansion. For sophisticated macros that provide complete domain-specific languages, such as SQL queries, regular expressions or LALR parser tools, such error reporting would be next to useless, as the con-

ICFP '06 September 16–21, 2006, Portland, Oregon, USA.

nection between the original form and the translated form is quite complex.

Again, this has not traditionally been a problem for Scheme programs, because they essentially have no static semantics to be checked.

2. Adding static semantics with Ziggurat

Our task in this paper is to describe a system, which we call Ziggurat, that addresses these problems. Ziggurat allows the programmer to define syntactic extensions to a language and define both dynamic and static semantics for the new forms. Thus programmers can develop task- or domain-specific languages that have their own static semantics. The static semantics can be checked at compile time *in terms of* the original source.

As a second benefit, the static semantics of a term written in a macro-embedded higher-level language can be used to define the static semantics of its realisation at the next layer down in the language tower. This is a critical benefit, since frequently the point of a specialised notation is that it has restricted dynamic semantics—which corresponds to richer static semantics, or more precise compile-time analysis.

For example, if we describe a computation using a regular expression, it is easy to reason about its static properties (*e.g.*, it is trivially true that the computation terminates on finite input). If we translate the regular expression into the equivalent finite-state automaton *rendered in C*, then we need more powerful analyses to reason about the behaviour of the program. Thus, if we can connect static semantics *across* the layers of our language tower, we can potentially improve the quality of the reasoning we can perform on the source code.

Ziggurat directly addresses our desire to "peel away" Scheme's macro system and employ it for languages with significant amounts of static semantics: Although it is written in Scheme, it a general-purpose tool. To make this explicit, the example applications we will consider in this paper will be extensions to assembly language, along with associated static analyses, which is about as un-Scheme-like as we can manage.

3. Lazy delegation

Ziggurat uses a specialized object-oriented system we call *lazy delegation.* An object in this system is created dynamically, and interacts with other objects by sending messages. Unhandled messages are passed off to an object's super object. However, unlike other systems of this kind, such as Self [24], super objects are not instantiated at object-creation time, but rather, are calculated on demand.

All objects have a class, similar to most object-oriented systems. (Unlike other object-oriented systems, however, classes are not required to have a hierarchical structure, and thus cannot be considered subtypes.) A class is defined not only by its methods, but also by a special delegation function, that calculates the super object for objects of that class.

Figure 1 shows an example of creating classes and objects. This code creates two classes to represent numbers: a class to hold real numbers, and a class to hold integers. These two classes have very different internal representations: an integer is represented by a single Scheme integer, while a real number is represented by a pair of the mantissa and exponent. `real-class` is defined as a "top" class, meaning that an object of class `real-class` has no super object. `int-class` is defined with a rewrite function that takes an integer, and returns an object of class `real-class`. Ziggurat objects do not have multiple elements of internal state.[1] Instead, an object has a single piece of internal data, its "internal representation." This is what is passed to the lazy-delegation creation function when a super object must be instantiated. Object methods access this datum with the `view` form. For example, note that the `int-class` rewrite function does not take an object of class `int-class`, but rather, the internal representation of that object.

```
; Real number objects are described
; by a pair of integers (m . e), where
; the value x is determined by
;     x = m * 10^e
(define real-class (make-top-class))

; Integer objects are described by a
; single integer; to instantiate as
; real number, use an exponent of 1.
(define int-class
  (make-class
   (λ (x) (make-object real-class
                       (cons x 0)))))

(define int-20 (make-object int-class 20))
```

Figure 1. Creating classes and objects

```
(declare-method (num->string n))

(method real-class num->string
 (λ (n)
   (let* ((data (view real-class n))
          (mant (car data))
          (exp  (cdr data))
          (m (number->string mant))
          (e (number->string exp))
     (string-append m "E" e))))

(method int-class num->string
 (λ (n)
   (let ((snum (view int-class n)))
      (number->string n))))
```

Figure 2. Creating methods

Figure 2 shows an example of method definition in Ziggurat. The `declare-method` form defines a method `num->string`, invoked in the style of a generic function. The `method` forms define how the different classes handle the `num->string` message. Objects of the `real-class` class first extract their internal representation using the `view` function, and then render their mantissa and exponent in scientific notation, while the specialised `int-class` objects simply render themselves as a simple integer.

The laziness of lazy delegation comes in handy in cases where we want to do some computation before fixing an object to delegate to. This functionality is not commonly needed, but is useful in building a macro system, as we shall see.

4. A macro system using lazy delegation

Lazy delegation objects present a natural way to represent syntax and macros. We represent each piece of syntax as an object, and use the rewrite function to hold a compilation function for that object. Thus a new syntax node—say, a multi-way conditional form—can delegate messages to the syntax node to which it expands (presumably a tree of nested if/then/else nodes). However, it can also intercept any messages it wishes to override and directly handle them

[1] At least, not in our initial prototype design. We will likely add multiple instance variables in a later revision.

with methods of its own; we will exploit this to implement extended static semantics, later.

A macro definition contains two functional parts: a parsing function, and a rewrite function. In Ziggurat, each macro describes a class of syntax object, where the rewrite function of the macro is the rewrite function of the class.

Consider a Scheme macro `(apply-to-self` *e*`)` that takes an expression *e*, and rewrites it to an application of that expression to itself of the form `(`*e e*`)`. The equivalent macro in Ziggurat would define a class `apply-to-self`. The data that each object of this class contains would be represented as a hash table. This table would have a single field, with the key `fun`. That field would contain a Scheme expression corresponding to the function (and argument) sub-expression of `apply-to-self`. The rewrite function would create a new syntax object that represents the translated expression presented above. Then, a parse function that takes a Scheme list starting with the symbol `apply-to-self` and returns an object of class `apply-to-self` would be inserted into the top-level Scheme environment.

In Ziggurat, this can be done by having the language designer manually specify these functions, roughly corresponding to what Scheme implementations often call "low-level" macros. Since the method of specifying pattern-based "high-level" macros is often dependent on the language being extended, we do not commit to a single high-level macro language, but instead provide a number of macros for building a high-level macro language. We will have more to say about these macros in section 9.

We use the laziness of lazy delegation to break a potential cyclic dependency. If analysis is based on methods on objects, and these methods may require delegation, then analysis may depend on the rewrite function. However, in many stages of compilation, the rewrite function will require analysis. We use the laziness of lazy delegation to ensure that the rewrite function is run after the necessary analysis has taken place.

5. Example: assembly language

Let's start by defining a simple assembly language as our base language, encoded with an s-expression concrete syntax.

```
r ∈ Reg   ::= a | b | c | ...
l ∈ Label ::= *a | *b | *c | ...
c ∈ Const ::= 0 | 1 | 2 | ...
e ∈ Expr  ::= r | l | c
s ∈ Stmt  ::= (mv r e)
            | (add r e e)
            | (ld r e)
            | (st e e)
            | (bez e e)
            | (jmp e)
            | (let ((l s)...) s)
            | (letrec ((l s)...) s)
```

At this language level, values are untyped; although constants are written as integers, their interpretation depends on their use.

There are two kinds of names at this language level: registers and labels. Register names have a flat, global scope, and we assume an unbounded number of them (perhaps this language is to be used as input to a global register allocator). Code labels, however, have hierarchical scope, and are bound to program locations with `let` and `letrec` statements. Note that the body of `let` and `letrec` statements is a single statement, not a sequence of statements. The sets of register names and labels are disjoint; labels begin with a `*` character, while register names do not.

- **add** and **mv** manipulate values in registers. When they are done, they branch to the label `*next`.
- **ld** and **st** perform loads and stores, transferring data between the memory system and the register set. Both statements branch to the label `*next` on completion.
- **bez** and **jmp** perform branching: `jmp` is an unconditional jump, while `bez` branches to its target (the second expression) if the test value (the first expression) is zero. If the test value is non-zero, then `bez` branches to the label `*next`. For both statements, the target address can be a register, allowing "computed" branches (and making control-flow analysis, in general, undecidable). We will later exploit the ability to do computed branches for functions.
- **let** and **letrec** bind labels, with hierarchical, lexical scope. We can establish sequences and DAGs of code with `let`, and construct circular control structure with `letrec`.[2]

5.1 Assembly macros

New syntax at this language level is defined with the special form `define-asm-syntax`. This is a macro, itself defined in the meta-language, that introduces a new parser function to the top-level parse environment for assembly statements. A declaration takes the form:

```
(define-asm-syntax keyword transformer)
```

Additionally, this defines a class named *keyword* with a rewrite method defined by the transformer. The parse function that is inserted by `define-asm-syntax` into the top-level environment is also defined by the transformer.

Syntax transformers are built by the `syntax-rules` macro. The form of a `syntax-rules` transformer should be familiar to anyone who has used Scheme macros, although there are two notable differences: the syntax for defining patterns is different, as are the hygiene rules.

A `syntax-rules` macro takes the form

```
(syntax-rules (captured-name ...)
  (pattern template)
  (pattern template)
  ...)
```

The *pattern* part of a macro defines the parser for that syntax object. Unlike Scheme, assembly language has a number of syntactic types (including statements, expressions and all subtypes of expressions), and it is possible to define new types of syntax. A *pattern* takes the form:

```
(keyword (name syntax-type) ...)
```

The `(`*name syntax-type*`)` part parses an object of the type *syntax-type*, and binds it to *name* in the template. For example, the pattern

```
(kons (n asm-register)
      (x asm-exp)
      (y asm-exp))
```

applied to the form

```
(kons x 3 4)
```

would parse `x` as a register name, `3` as an expression, and `4` as an expression. Then, it would build a syntax object of class `kons`, and bind the subexpressions to the fields `n`, `x` and `y` in it.

We allow another kind of clause in patterns, using the keyword "`...`" for pattern repetition. For example, the pattern

```
(seq (fst asm-stm) (rest asm-stm ...))
```

[2] We have adopted this notation for describing low-level control structure from a previous design we made for a loop package [21].

would match

- `(seq (jmp *next))`
- `(seq (mv rb ra) (jmp *next))`
- `(seq (mv rb ra) (st rc *x) (jmp *next))`

The *template* parts of the syntax rule define the rewrite function for the syntax object. In essence, the syntax object is rewritten to look like the template, with bound variables replaced by their values. For example, if we define the `seq` keyword by

```
(define-asm-syntax seq
 (syntax-rules (*next)
   ((seq (x asm-stm)) x)
   ((seq (x asm-stm) (more asm-stm ...))
    (let ((*next (seq . more)))
      x))))
```

then, `(seq (mv x 5) (add x x 10))` would be rewritten

```
(let ((*next (seq (add x x 10))))
  (mv x 5))
```

This is actually a bit of a simplification. The rewrite function for an object must return another syntax object, not source code. We solve this by parsing the template, using a specialized method. The details of how this works are covered in section 9.

There is one final problem that must be dealt with: that of *hygiene*. We don't want temporary register and label names used in a macro definition to capture, or be captured by, names introduced in the use of that macro. We use the common method of renaming in order to solve this, but there is a complication: in assembly language, certain names have special meaning, such as the `*next` label. We therefore allow the language designer to specify *captured names* in the `syntax-rules` form, which explains to the system that those names are special, and not to be rewritten.

6. Termination analysis

In Ziggurat, local analysis is easy: we simply perform a recursive descent on the parse tree. To demonstrate this, we present a simple termination analysis: a method `halts?` on assembly statements that returns true if it determines that control will definitely leave the statement under all starting conditions, and false otherwise. Naturally, this analysis will frequently fail, returning false to indicate that it simply does not know.

6.1 Termination analysis for assembly language

The analysis we perform on base assembly objects is trivial: if a statement contains a `letrec` statement, or a `jmp` or a `bez` to a register, it returns false, otherwise it returns true. This is fairly easy to express in Ziggurat: we define a `halts?` method for each kind of statement. For example, the analysis method for a `let` statement is:

```
(method asm-let halts?
  (λ (s)
    (let ((tbl (view asm-let s)))
      (and (halts? (hash-table-get tbl 'stm))
           (andmap (λ (x) (halts? (cadr x)))
                   (hash-table-get tbl
                                   'bindings))))))
```

If the statement part of the `let` and the statement part of each binding all terminate, then the entire statement does as well. If any of them are not known to terminate, then the result will not necessarily terminate, either.

This termination analysis is easily implemented, but is very imprecise: it will return false for most instances of complex code.

The advantage of using lazy delegation comes in here. Consider our `seq` macro. There is no termination analysis rule built in for `seq`, but by delegating, one can be derived: Ziggurat first translates the `seq` statement to lower-level assembly language, and then applies termination analysis to that. For example, consider the code:

```
(seq (mv x 5) (add x x 5))
```

This translates into

```
(let ((*next (add x x 5)))
  (mv x 5))
```

Our implemented termination analysis tells us that this terminates, and thus this is the result of running `halts?`.

6.2 The run-n macro: dynamic and static semantics

Suppose we now define a `run-n` macro of the form

`(run-n` *n s*`)`

so that we may write loops that execute a statement *s* a constant number *n* of iterations. The macro's definition is:

```
(define-asm-syntax run-n
  (syntax-rules (*next)
    ((run-n (n asm-const) (s asm-stm))
     (letrec ((*loop (seq (bez loop-var *escape)
                          s
                          (add loop-var
                               loop-var -1)
                          (jmp *loop)))
              (*escape (jmp *next)))
       (seq (mv loop-var n)
            (jmp *loop))))))
```

This macro simply produces code that executes `s`, `n` times. Note that the form will terminate if its argument `s` does, and if `n` $\geq$ 0. However, since a use of the `run-n` macro expands into a `letrec` statement, the default `halts?` analysis will return false. So, we merely override the `halts?` method:

```
(method run-n halts?
  (λ (x) (let* ((tbl (view run-n x))
                (loop-num (hash-table-get tbl 'n))
                (inner-s  (hash-table-get tbl 's)))
           (and (halts? inner-s)
                (>= loop-num 0)))))
```

This produces a more precise analysis. (Note, however, that we've now introduced the possibility that a macro is now able to override a presumably *correct* base-level static analysis with one that contains errors—either accidentally or maliciously. We'll discuss this issue later.)

7. Type inference

In the case of termination analysis, our base analysis was so simple it could be implemented by recursive descent. With most analyses, we cannot do this directly. In order to do non-local analysis in Ziggurat, we pass around an analysis object, that describes some computation to be run at some later time. Each syntax node, then, rather than running the analysis on itself, describes how to change the analysis object.

For example, type inference is non-local. In languages with type inference, such as Haskell, the type of a variable reference might be dependent on the context of that use, or even on the context of another use of the same variable. Like many type-inference algorithms, we represent this analysis as a system of constraints to be solved.

$$\frac{\begin{array}{c} E \vdash e : t' \text{ in } t \\ E \vdash r : t' \text{ in } \mathrm{inst}(E[\texttt{*next}]) \\ \mathrm{inst}(E[\texttt{*next}])/r = t/r \end{array}}{E \vdash [\![\texttt{(mv } r\ e\texttt{)}]\!] : t}\ \textsc{MvType}$$

$$\frac{\begin{array}{c} E \vdash e_l : \textbf{word} \text{ in } t \\ E \vdash e_r : \textbf{word} \text{ in } t \\ E \vdash r : \textbf{word} \text{ in } \mathrm{inst}(E[\texttt{*next}]) \\ \mathrm{inst}(E[\texttt{*next}])/r = t/r \end{array}}{E \vdash [\![\texttt{(add } r\ e_l\ e_r\texttt{)}]\!] : t}\ \textsc{AddType}$$

$$\frac{\begin{array}{c} E \vdash e : \textbf{word} \text{ in } t \\ E \vdash r : t' \text{ in } \mathrm{inst}(E[\texttt{*next}]) \\ \mathrm{inst}(E[\texttt{*next}])/r = t/r \end{array}}{E \vdash [\![\texttt{(ld } r\ e\texttt{)}]\!] : t}\ \textsc{LdType}$$

$$\frac{\begin{array}{c} E \vdash e' : t' \text{ in } t \\ E \vdash e : \textbf{word} \text{ in } t \\ \mathrm{inst}(E[\texttt{*next}]) = t \end{array}}{E \vdash [\![\texttt{(st } e\ e'\texttt{)}]\!] : t}\ \textsc{StType}$$

$$\frac{E \vdash e : t \text{ in } t}{E \vdash [\![\texttt{(jmp } e\texttt{)}]\!] : t}\ \textsc{JmpType}$$

$$\frac{\begin{array}{c} E \vdash e' : t \text{ in } t \\ \mathrm{inst}(E[\texttt{*next}]) = t \\ E \vdash e : \textbf{word} \text{ in } t \end{array}}{E \vdash [\![\texttt{(bez } e\ e'\texttt{)}]\!] : t}\ \textsc{BezType}$$

$$\frac{\begin{array}{c} E \vdash s_1 : t_1, \ldots, E \vdash s_j : t_j \\ E[l_1 \mapsto \mathrm{gen}(t_1), \ldots, l_j \mapsto \mathrm{gen}(t_j)] \vdash s : t \end{array}}{E \vdash [\![\texttt{(let ((}l_1\ s_1\texttt{)}\ldots\texttt{(}l_j\ s_j\texttt{))}\ s\texttt{)}]\!] : t}\ \textsc{LetType}$$

$$\frac{E' \vdash s : t \quad E' \vdash s_1 : t_1, \ldots, E' \vdash s_j : t_j}{\begin{array}{c} E \vdash [\![\texttt{(letrec ((}l_1\ s_1\texttt{)}\ldots\texttt{(}l_j\ s_j\texttt{))}\ s\texttt{)}]\!] : t \\ \text{where } E' = E[l_1 \mapsto \mathrm{gen}(t_1), \ldots, l_2 \mapsto \mathrm{gen}(t_2)] \end{array}}\ \textsc{LetrecType}$$

$$\frac{}{E \vdash c : \textbf{word} \text{ in } t}\ \textsc{ConstType} \qquad \frac{}{E \vdash l : \mathrm{inst}(E[l]) \text{ in } t}\ \textsc{LabelType} \qquad \frac{}{E \vdash r : t[r] \text{ in } t}\ \textsc{RegisterType}$$

Figure 3. Polymorphic type rules for assembly language statements (top) and expressions (bottom).

7.1 A type system for assembly language

For the assembly language, both statements and expressions are typed. The type of an expression represents the kind of value it represents, while the type of a statement represents requirements on registers on entry to that statement. In order for this to work, we need parametric polymorphism: although `(add z x y)` requires `x` and `y` to be numbers, it imposes no such requirement on `z`. We refer to the types of statements as *code types.* Since labels are bound to statements, and labels can be used as expressions, expressions can have code types: this reflects the fact that registers can hold code pointers.

At the lowest language level, presented above, we have only one base data type: **word**. A **word** can be an integer, a character, or a pointer to structured data. This part of the type system is deliberately kept open, with the expectation that higher language levels will elaborate on it.

A monomorphic code type is a mapping from all possible registers to types. Since this would be highly impractical to represent, and also far too restrictive, we employ polymorphism by means of the following type system:

$$\begin{array}{lll} \tau \in \text{TypeSchema} & ::= & \forall \overline{v}.t \\ v \in \text{TypeVar} & ::= & \alpha, \beta, \gamma, \ldots \\ t \in \text{Type} & ::= & \textbf{word} \mid ct \mid v \\ ct \in \text{CodeType} & ::= & (l : t; ct) \\ & & \mid v \\ r \in \text{Reg} & ::= & \texttt{a}, \texttt{b}, \texttt{c}, \ldots \end{array}$$

A code type is represented by the triple $(l : t; ct)$, where ct is another code type. For example, the code type that maps register `x` to **num**, `y` to **num**, and is quantified over the rest would be represented $(\texttt{x} : \textbf{num}; (\texttt{y} : \textbf{num}; \alpha))$. We additionally require that a code type does not contain the same register twice. (This is enforced, in the actual implementation, with a parametric kinding discipline. We have excised kinds from our presentation for simplicity.)

This type system is based directly on Wand's row polymorphism [26], as defined in HM(X) [19]. It introduces *type schemas,* which are types quantified over variables. (Note that, in our simple unkinded system, we use the same set of type variables for both row and scalar types.)

We define two relations: a relation that assigns types to statements, and one that assigns types to expressions. $E \vdash s : t$ holds when statement s has type t in environment E, where an environment maps labels to type schemas. $E \vdash e : t \text{ in } t'$ holds when expression e has type t in an environment E, and e appears in a statement with type t'. The definition of these relations appears in Figure 3.

The relations inst and gen refer to type-schema instantiation and generalization relations. E is a mapping from labels to type schemas. $E[l]$ is label-environment lookup. Similarly, $t[r]$ is code-type lookup; it is undefined to look up a register that a code type does not define.

7.2 Constraint-based type inference

The analysis we implement assigns type schemas to statements. We do this via a constraint solver: we declare a method

```
(make-stm-type-goal s t e)
```

that returns the constraints required to cause statement `s` to have type `t` in label environment `e`.

To represent and solve constraints, we use a variant of the Mini-Kanren logic-programming system [7]. The basic unit of computation here is the *goal,* which is either satisfiable or unsatisfiable. A goal can contain logic variables, and in order for a goal to be satisfiable, there must be at least one mapping from logic variables to types that satisfies it. Running a goal allows us to compute the most general mapping that satisfies the goal.

The most basic goal is unification. In basic Mini-Kanren, two values are unifiable if they are equal; but here, this is not enough. For example, if t is the type $(\texttt{x} : \textbf{word}; (\texttt{y} : \textbf{word}; \alpha))$, and t' is the type $(\texttt{y} : \textbf{word}; (\texttt{x} : \textbf{word}; \alpha))$, then we want to be able to unify t and t'.

We accomplish this through object-oriented methods. We declare `unify` to be a method, and require every object used in unification to implement this method. It is up to the language designer to make the `unify` method symmetric, so that `(unify x y)` returns the same thing as `(unify y x)`. Evaluating `(unify x y)` returns a subgoal which must be satisfied in order for x to unify with y. For example, to unify $(\mathtt{x} : \alpha; \beta)$ and $(\mathtt{x} : \gamma; \delta)$, we must unify, as subgoals, α with γ, and β with δ. To unify $(\mathtt{x} : \alpha; \beta)$ and $(\mathtt{y} : \gamma; \delta)$, the process is only slightly more complex: we generate a fresh logic variable ϵ, and unify β with $(\mathtt{y}:\gamma;\epsilon)$, and δ with $(\mathtt{x}:\alpha;\epsilon)$.

Once type unification is defined, `make-stm-type-goal` can be written as a direct implementation of the type rules. For example, the rule for `add` is simply that its two arguments be of type `word` on entry, that its result be of type `word` on exit, and that all other registers remain unchanged. Figure 4 shows the code for the rule. This system of constraints is very extensible, as we shall see.

7.3 Structured data

This type system has a problem. It is incomplete, and in fact, unsound: storing to and loading from memory causes data to lose all type information, allowing a program to access uninitialized memory, execute arbitrary code, and so forth.

The problem lies in the fact that the only memory operations we have to work with are too low-level: it's difficult to come up with a type system that is both safe and allows for pointer arithmetic. Most languages take the more reasonable approach of having a type system for a higher-level language, and translating the guaranteed safe code to unsafe assembly language. The Ziggurat approach is more incremental: we write, as a language extension, structured data-type operations, and then extend the incomplete assembly language type system with type rules for this structured data.

At the next level above assembly language, we add only three kinds of structured data: sum types, product types, and named recursive types. The definitions of these macros are fairly straightforward, similar to the compilation of structured data operations in any language.

$$
\begin{array}{lcl}
s \in \text{Stmt} & ::= & \ldots \\
 & | & (\mathtt{kons}\ r\ e\ e) \\
 & | & (\mathtt{kar}\ r\ e) \\
 & | & (\mathtt{kdr}\ r\ e) \\
 & | & (\mathtt{left}\ r\ e) \\
 & | & (\mathtt{right}\ r\ e) \\
 & | & (\mathtt{branch}\ r\ l\ l)
\end{array}
$$

A sampling of the macros used to define this new structure can be found in Figure 5. The basic constructors and destructors are similar to those found in higher-level languages.

- **kons**, **kar** and **kdr** define product types. The `(kons` x y`)` form builds a two-place data structure (or *pair*) containing the values of x and y, and places the result in r. A `(kar` r x`)` form extracts the first value of x and places the result in r; likewise, `(kdr` r x`)` extracts the second value.

  The macro for `kons` expands into code that performs a subroutine call to a `malloc` procedure to allocate a new two-word block of storage; this keeps the macro simple and the extraneous details of memory management off-stage so that we may focus on our static semantics.

- **left**, **right** and **branch** define sum types. A `(left` r e`)` form builds a two-place sum object with the value of e for the first place; likewise, `(right` r e`)` builds a two-place sum object with the value of e for the second place. A `(branch` r l_1 l_2`)` performs a conditional branch based on the value of r: if it was constructed with `left`, then `branch` jumps to l_1, otherwise, it jumps to l_2.

```
(define-asm-syntax kons
 (syntax-rules
  ((kons (n asm-var) (kar asm-exp)
         (kdr asm-exp))
   (let ((*k (seq (mv n rv)
                  (st n kar)
                  (add tmp n 1)
                  (st tmp kdr))))
     (seq (mv arg1 2)
          (mv rp *k)
          (jmp *malloc))))))

(define-asm-syntax kar
 (syntax-rules
  ((kar (x asm-var) (k asm-exp))
   (ld x k))))

(define-asm-syntax kdr
 (syntax-rules
  ((kdr (x asm-var) (k asm-exp))
   (seq (add tmp k -1)
        (ld x tmp)))))
```

Figure 5. Macros for structured data

This, in and of itself, is not adequate to build an advanced type system. Despite the fact that we have built macros for structured data, the type system is not smart enough to detect what is a proper use of the basic constructs, and what is not. Code such as

```
(seq (kons x 1 2) (kdr y x) (kdr z y))
```

will not signal an error, as it still delegates handling for type checking to the underlying expanded code. However, we are now in a position to add extension-specific static semantics to the new forms.

7.4 Extending type inference

The problem with our type system is that it has no sum types, product types or named types: all of these are represented by the universal word type. So, we begin by defining these, as lazy-delegation object classes. There is very little functionality in the basic definition of the classes: at a lower language level, structured data simply looks like a pointer, and the rewrite function reflects that.

```
(define pair-type
  (make-class (λ (x) reference-word-type)))
```

The basic functionality of a structured data type is in unification. As in lower language levels, we define a unify method.

```
(method pair-type unify
  (λ (u v) (let* ((dat (view pair-type u))
                  (kar (car dat))
                  (kdr (cdr dat)))
            (unify-kar-kdr v kar kdr))))
(declare-method-default (unify-kar-kdr v kar kdr)
  (λ (v kar kdr)
    (type-error "not a pair type" v)))

(method pair-type unify-kar-kdr
 (λ (v kkar kkdr)
   (let* ((dat (view pair-type v))
          (kar (car dat))
          (kdr (cdr dat)))
     (all (==-check kar kkar)
          (==-check kdr kkdr)))))
```

```
(method asm-add make-stm-type-goal
  (λ (s t e)
    (let* ((tbl  (view asm-add s))
           (dst  (hash-table-get tbl 'dst))
           (srcl (hash-table-get tbl 'srcl))
           (srcr (hash-table-get tbl 'srcr))
           (next-type (instantiate-type (type-env-lookup e '*next))))
      (fresh (dst-type mirror-dst-type rest-type)
             (make-exp-type-goal dst  reference-word-type e next-type)
             (make-exp-type-goal srcl reference-word-type e t)
             (make-exp-type-goal srcr reference-word-type e t)
             (==-check t (make-object label-type
                                      (make-row-type (asm-var-name dst)
                                                     mirror-dst-type
                                                     rest-type)))
             (==-check next-type (make-object label-type
                                              (make-row-type (asm-var-name dst)
                                                             dst-type
                                                             rest-type)))))))
```

Figure 4. The ADDTYPE rule, procedurally encoded.

$$\frac{E \vdash e_l : t_l \text{ in } t \quad E \vdash e_r : t_r \text{ in } t \quad E \vdash e : \mathbf{pair}(t_l, t_r) \text{ in inst}(E[\texttt{*next}]) \quad \text{inst}(E[\texttt{*next}])/r = t/r}{E \vdash [\![\texttt{(kons } r\ e_l\ e_r\texttt{)}]\!] : t}\ \textsc{KonsType}$$

$$\frac{E \vdash e : \mathbf{pair}(t_l, t_r) \text{ in } t \quad E \vdash r : t_l \text{ in } \texttt{inst}(E[\texttt{*next}]) \quad \text{inst}(E[\texttt{*next}])/r = t/r}{E \vdash [\![\texttt{(kar } d\ s\texttt{)}]\!] : t}\ \textsc{KarType}$$

$$\frac{E \vdash e : \mathbf{pair}(t_l, t_r) \text{ in } t \quad E \vdash r : t_r \text{ in } \texttt{inst}(E[\texttt{*next}]) \quad \text{inst}(E[\texttt{*next}])/r = t/r}{E \vdash [\![\texttt{(kar } d\ s\texttt{)}]\!] : t}\ \textsc{KdrType}$$

$$\frac{E \vdash e : t_l \text{ in } t \quad E \vdash r : \mathbf{sum}(t_l, t_r) \text{ in inst}(E[\texttt{*next}]) \quad \text{inst}(E[\texttt{*next}])/r = t/r}{E \vdash [\![\texttt{(left } d\ s\texttt{)}]\!] : t}\ \textsc{LeftType}$$

$$\frac{E \vdash e : t_r \text{ in } t \quad E \vdash r : \mathbf{sum}(t_l, t_r) \text{ in inst}(E[\texttt{*next}]) \quad \text{inst}(E[\texttt{*next}])/r = t/r}{E \vdash [\![\texttt{(left } d\ s\texttt{)}]\!] : t}\ \textsc{RightType}$$

$$\frac{E \vdash r : \mathbf{sum}(t_l\ t_r) \text{ in } t \quad t/r = t' \quad E \vdash l_l : [r \mapsto t_l]t' \text{ in } t \quad E \vdash l_r : [r \mapsto t_r]t' \text{ in } t}{E \vdash [\![\texttt{(branch } r\ l_l\ l_r\texttt{)}]\!] : t}\ \textsc{BranchType}$$

Figure 6. Type rules for structured types

We want product types to unify with other product types, and not with any other structured types. Product type s unifies with product type t iff the kar of s unifies with the kar of t, and the kdr of s unifies with the kdr of t. This is directly represented in the type rule.

Finally, we define goals for our structured type macros. With the rules presented in Figure 6, unsafe code will signal an error.

7.5 Introducing named types

Named types present an additional complication. In order to associate a named type with a form, we must create an additional class of syntax node to represent a structured type, and insert them into the language as extensions. This is reflected in the fact that the grammar now requires new syntax categories. The new additions are:

$n \in$ TypeName ::= `a` | `b` | `c` | `...`

$t \in$ Type ::= . . .
    | n ; Named-type reference
    | `(x` t t`)`   ; Product type
    | `(+` t t`)`   ; Sum type

$s \in$ Stmt ::= . . .
    | `(typedef ((`n t`)...)` s`)`
    | `(name` r n e`)`
    | `(unname` r n e`)`

- **typedef** introduces named types. The `(typedef ((`n t`))` s`)` form introduces one named type, n, in the statement s. The type t describes the internal format of objects of type n. The name n is visible in t; this allows the language user to define a recursive type.
- **name** and **unname** construct and deconstruct named types. the `(name` r n e`)` form introduces into register r data of form e, with the named type n. Logically, then, e must have the same type as the form of n. Likewise, `(unname` r n e`)` takes data of named type n from e, strips off the name, and puts the result in r.

This is easily implemented in Ziggurat. First, we must have a class for named types. A named type compiles into its internal form. Thus, if `inflist` is declared to be `(x word inflist)`,

$$\frac{E, F' \vdash s : t}{E, F \vdash [\![\texttt{(typedef ((}n_{11}\texttt{) ...) } s\texttt{)}]\!] : t}\ \textsc{TypedefType}$$
where $F' = F[n_1 \mapsto s_1, ...]$

$$\frac{\begin{array}{c} E \vdash r : n \text{ in } t \\ E \vdash e : F[n] \text{ in inst}(E[\texttt{*next}]) \\ \text{inst}(E[\texttt{*next}])/r = t/r \end{array}}{E, F \vdash [\![\texttt{(name } r\ n\ e\texttt{)}]\!] : t}\ \textsc{NameType} \qquad \frac{\begin{array}{c} E \vdash r : F[n] \text{ in } t \\ E \vdash e : n \text{ in inst}(E[\text{*next}]) \\ \text{inst}(E[\text{*next}])/r = t/r \end{array}}{E, F \vdash [\![\texttt{(name } r\ n\ e\texttt{)}]\!] : t}\ \textsc{UnnameType}$$

Figure 7. Type rules for named types

then the object representing the type `inflist` rewrites to a pair-type object containing a `word` and an `inflist`. This reflects the fact the type name is invisible in the data; data of a named type is indistinguishable from data of that type name's form.

```
(define named-type
  (make-class (λ (x) (hash-table-get x 'form))))
```

Unification for a `named-type` is simple: a named type only unifies with itself.

Once we have done that, we still need to define type goals in this augmented type system. We introduce the notion of a *type name environment*. A type name environment is a mapping from type names to types. A type goal is now of the form $E, F \vdash s : t$, meaning that in variable environment E and type name environment F, statement s has type t. Implementing this would seem to present a complication: the object-oriented methods we use to create type goals take only one environment as an argument. This is not a problem, though: the environment we use in implementing type goals is a lazy delegation object, and so we simply add the type name environment functionality to it, as needed.

With this addition, defining type goals becomes a simple matter. Notably, there are no type goals generated by the `typedef` syntax; it merely uses the type goals from its inner statement, after augmenting the type name environment. The `name` and `unname` statements merely enforce that uses of a named type match the form of a named type, found in the type name environment.

Parsing becomes a bit more complicated. We now have a new kind of syntax for type forms, which means we need new syntax classes, syntax environments, and parse methods. Fortunately, by defining new parse environments, we can still use high-level macros.

8. Control-flow analysis

We will develop a flow analysis as our final example of an extensible static analysis. Flow analysis is an important class of static reasoning, enabling a host of dependent analyses and optimisations. As it is non-local, we implement it using a monadic solution, similar to our constraint-based solution for type inference.

The question asked by a control-flow analysis is simple: for a particular statement, where might control proceed after completing that statement? In the case of gotos to explicit labels, this is trivial to answer: control always passes to the code attached to the given label. But this is not sufficient: calculated gotos are essential for compiling code in higher-order languages. Likewise, though, higher-order languages often know very well where control will go, and have specialized flow-analysis algorithms.

8.1 Flow analysis for assembly language

For the basic assembly language level, we again start with a naive flow-analysis algorithm, though we define it in a flexible way. The goal of the algorithm is to map assembly-language statements to sets of other statements, or a special symbol $\bot$, which represents an unknown control point.

The algorithm for assembly language is simple, and in fact, completely local: if the statement next branches to one or more labels, we return the statement that the labels are statically bound to, otherwise, we return $\bot$. This, on its own, would be trivial to implement, much like the `halts?` example above.

Once we add more complex control-flow statements, there will be opportunities for more precise control flow analysis. Most of these will be fixed-point algorithms, and so, we describe a framework for implementing flexible, modular fixed-point algorithms, and implement our base analysis in this framework.

This framework, which we call *Tsuriai,* is structured much like the Kanren-based logic programming system we use for type inference. Algorithms written in Tsuriai consist of a series of assignments of values to variables, which are run until there is no change in the assignments. For example, if we have two variables x and y, where x is defined by $x = y \cup \{3\}$, and y is defined by $y = x \cup \{4\}$, and we wish to find the least fixed point of these mutually recursive definitions, we would write

```
(fresh (x y)
  (<- x (U y (set 3)))
  (<- y (U x (set 4))))
```

This is a goal in Tsuriai. Running it with a `run` command tells us that $x = y = \{3, 4\}$.

To write a control-flow algorithm, we make a variable for each assembly statement, and the value that is assigned to it is the set of statements it may branch to. For each assembly-language statement, we have a method `make-stm-flow-goal`, that returns a goal. For example, this method on a `mv` statement is

```
(method asm-mv make-stm-flow-goal
  (λ (s e)
    (<- (! s) (label-flow-lookup e '*next))))
```

The function `!` returns a unique variable for each statement. The variable `e` is not a fixpoint variable, but instead is simply a mapping from labels to sets of statements.

Since our base-level analysis is so simple, it reaches a fixed point after a single iteration. However, it is extensible, and adding new goals for new kinds of statements may make the composite analysis take much longer.

8.2 Funclets

At higher code levels, we have complex code structures that frequently get compiled to computed gotos, such as function return or multi-way case statements rendered by means of jump tables. Although we might not know the targets of these gotos at a low level, it may be determinable at a higher level. The example we implement is control-flow analysis for function calls, specifically, 0CFA [20].

We implement a specialized version of function call, *funclets*, which model the control and data-flow substructure of function linkage. Funclets are a restricted control operator in two ways. First, they never return, making funclet programming continuation-passing style. Secondly, the arguments to a funclet can only be other funclets: other values are passed by side-effect in registers, in the manner of assembly-language control transfers. Funclets are intended to be an intermediate stage in the compilation of functions.

$c \in$ CVar ::= `a | b | c | ...`
$s \in$ Stmt ::= ...
  | `(fletrec (((`c c c`...)` s`) ...)` s`)`
  | `(fcall` c c c`...)`

Once again, we have a new syntactic class, in this case, cvars. A cvar is a variable that is bound to a funclet, and has lexical scope. A cvar may be compiled into a register or a label, depending on the context.

We have two new kinds of syntactic keywords.

- Funclets are introduced by the `fletrec` keyword. A statement of the form `(fletrec (((`f x`)` s_1`))` s_2`)` introduces one funclet, with the name f. Upon a call, the funclet binds the variable x to the actual parameter of the call, and runs the statement s_1. The cvar f is visible in both s_1 and s_2, while the cvar x is only visible in s_1. These cvars are disjoint from the labels and registers, and are thus only usable in `fcall` statements.
- Funclets are called with the `fcall` keyword. A statement of the form `(fcall` f x`)` calls the funclet that is referenced by the cvar f, with a single actual parameter x. Both cvars, f and x, can either be bound directly to funclets by an enclosing `fletrec` statement, or may be arguments to the current funclet. This makes funclets first-order data, in the manner of SML, Haskell or Lisp.

The evaluation of funclets is roughly equivalent to interpration of continuation-passing-style λ-calculus. Thus, we can directly adapt CPS-based flow-analysis algorithms for the language.

8.3 Flow analysis for funclets

Consider the assembly code that sets up an infinite loop by means of a tight funclet tail-recursion:

```
(fletrec (((f x) (fcall x x)))
  (fcall f f))
```

The `(fcall x x)` statement compiles into code that ends with `(jmp g6175)`, where `g6175` happens to be the register that Ziggurat compiles the cvar `x` into. Since this is a branch to a register, the base flow-analysis algorithm gives up, and tells us that the destination of the branch is $\perp$; in other words, it could be anywhere. But simple inspection of the code tells us that `x` will always be bound to `f`; there is only one possible destination of the `fcall`. What's more, there are algorithms that can tell us this.

We implement 0CFA in Tsuriai. The basis of 0CFA is abstract interpretation. 0CFA simulates running the program, while representing funclets by their code pointers, and merging all environment structures into a single value. We implement this by building Tsuriai goals for each cvar, in order to find all of the funclets that cvar may be bound to.

For example, the statement `(fcall x y)` generates a goal

```
(maps (! x)
  (λ (flet)
    (let ((formal (car (funclet-formals flet))))
      (<- (! formal) (U (! formal) (! y))))))
```

The function `maps` returns a goal that applies a function to each member of a set, and combines the goals returned by that function. In this case, `(! x)` is bound to the set of funclets that might be bound to the cvar `x`. So, for each of those funclets, we want to assert that the first formal cvar argument of that funclet may also be bound to the funclets assigned to the cvar `y`, currently represented by the variable `(! y)`.

9. Macros for writing macros

A macro defines two functions: a parse function and a rewrite function. For a low-level macro, these are specified directly, with a function for each, but this is highly inconvenient; what we would like would be a macro that would allow us to specify these functions by filling in templates. This is actually a somewhat complex proposition, since Ziggurat must be flexible enough to allow any number of language extensions that can change the syntax and semantics of the language in unpredictable ways. Fortunately, we can solve this through the use of lazy delegation: by layering environments and using a variety of parse methods, we can provide several "hooks" to the language developer.

Each parser takes the form of an environment. An environment is simply an object with a parse method. The super-object of an environment is its enclosing environment. For example, if a language designer wished to implement a `let-syntax` syntax class, he or she would make it so that

`(let-syntax ((`*keyword* *transformer*`) ...)`
  statement`)`

defines a new environment that uses *transformer* to parse statements with the keyword *keyword*, and then parses *statement* with that environment.

This structure helps us in the structure of high-level macros. To make a rewrite method, we build a specialized environment, and then parse the template. In order to do this, we actually define a number of new environments. Assembly-language high-level macros look like:

`(define-asm-syntax` *keyword*
  `(syntax-rules (`*captured-name* `...)`
    `((`*keyword* `(`*name* *syntax-type*`) ...)`
     template`)`
    `...))`

and rely on a number of layered environments.

- **base-asm-env**
  This defines the basic keywords: `let`, `mv`, and so forth.
- **top-level-asm-env**
  This environment allows us to define new keywords by side-effect. This is the environment level that `define-asm-syntax` actually manipulates.
- **syntax-rules-env**
  When the programmer uses `define-asm-syntax`, a new object of the `syntax-rules-env` class is created. This class only parses one type of clause: the `syntax-rules` clause. The result of this parse method is a function that parses the clause being defined. The `syntax-rules` clause also defines the class of syntax objects whose instances are returned by the newly-defined parse function. The rewrite function is encoded in the newly-defined class.
- **syntax-names-env**
  This environment is used to parse clauses in the pattern part of a `syntax-rules` declaration. It matches the names of syntax classes to the parse methods used to produce objects of those classes.
- **template-env**
  This is the environment used to parse templates. This does

most of the work of defining the rewrite method. The rewrite method for a `syntax-rules` form is actually merely the result of suspending the parsing of the template under the template environment.

The template environment is just the parser environment where the macro is defined, augmented with specialized parse methods for forms defined by the `syntax-rules` form. For example, the template environment will rename any label or register it comes across, save those defined in the *captured-names* part of the `syntax-rules` form. Likewise, it will replace any of the variables bound by the `syntax-rules` form with their corresponding value.

For example, consider the `seq` macro:

```
(define-asm-syntax seq
  (syntax-rules (*next)
    ((seq (x asm-stm)) x)
    ((seq (x asm-stm) (more asm-stm ...))
     (let ((*next (seq . more)))
       x))))
```

This inserts into the `top-level-asm-env` a parser for the keyword `seq`. It creates this parser by first building a syntax-rules-env, and then evaluating the `syntax-rules` statement in this environment. The `syntax-rules-env` parses each individual syntax rule, using the current `syntax-names-env` in order to parse the `asm-stm` keyword, and produces a new parsing function.

This parsing function must return an object of the `seq` class, which is also defined by parsing each clause in the `syntax-rules` form. It achieves this by first building a template-env, to parse the template. For example, the template-env for the first syntax rule would parse the label `*next` to the label `*next`, the keyword `x` to argument in the first place of the `seq` statement, and all other labels and registers to fresh names. It would then parse the statement `x`, which simply returns the first statement; thus, `(seq (mv x 5))` parses to `(mv x 5)`.

10. Related work

The idea of using objects to represent syntax is not new. The idea was first proposed by Dybvig, Hieb and Bruggeman [3]. Others have attempted to implement analysis on syntax objects, with varying degrees of success.

The PLT Scheme product DrScheme [4] comes with such a system. This system was originally called McMicMac [11], before it was folded into the DrScheme macro system. However, the analysis this does is limited to the static semantics of Scheme, and it is primarily a debugging tool.

Maddox [15] has developed a similar object-oriented system for defining macros that permit static analysis, and linking across levels of the language tower. Ziggurat builds off this work, introducing monadic operators to allow non-local analysis. For example, Maddox's system provide a means of doing type analysis with explicit types, while Ziggurat provides a means of doing type inference. In addition, Maddox's system has a fixed system of high-level macros, whereas Ziggurat allows the language designer to provide both the parsing function and the transformer as general Scheme functions. If the language designer wished to parse a form as either a constant or a label, but not a register, this is possible in Ziggurat.

Nanavati [17] also provides a system for extensible analysis. In order to implement significant syntactic extensions in Nanavati's system, the programmer was forced essentially to simulate an object-oriented architecture programmatically, which made for a fairly awkward, tortured coding style. Ziggurat captures this structure directly in the linguistic mechanisms of the meta-language.

The related, but orthogonal problem of analysis of multi-stage programs has been handled in depth by some, notably Walid Taha *et al.* [23]. Their system MetaML does not allow for true syntax extension, and while it allows variables to be sent across language levels, Ziggurat allows a syntax node to exist at several language levels simultaneously, permitting a high-level methods to override low-level methods. This is not necessary for the problems that MetaML is designed to solve; it is a more focussed technology.

Van Wyk et al [28] add functionality, called forwarding, to attribute grammars similar to Ziggurat's delegation. In a sense, Ziggurat can be seen as adding analysis to macros, while forwarding in attribute grammars can be seen as adding macros to a modular program analysis framework.

11. Conclusions and future work

Ziggurat provides a language-extension facility, similar to Scheme macros, that is flexible enough to work on assembly language, and powerful enough for advanced program analysis. There is much work to be done on this system. This includes:

- **Certification.** One of the disadvantages of the way Ziggurat is currently structured is that overloaded analyses can give wrong results, possibly leading to unsafe compilation. Since Ziggurat is intended as a powerful language extension toolkit, this is unavoidable in the general case. A possible solution, though, is to require that certain analyses provide a certificate that their result obeys certain properties, either that they give a correct answer, or that they do not violate safety rules. Certification in this scheme becomes just another analysis.
- **Analysis-driven translation.** Currently, the results of these analyses are not used in compilation, though lazy delegation allows for this. This would be an essential step in implementing code optimization in Ziggurat.
- **General-purpose parsers.** Specifying parsers extensibly is a very difficult problem. Currently, we solve this by only considering one kind of syntax: s-expressions. However, the monadic method of specifying analysis suggests a method to parse a more broad syntax, using recent work in monadic parser combinators [9]. Parsing in Ziggurat is currently done by an environment, which defines a parsing function. If the environment instead supplied a parsing monad, there is the potential to make a more general-purpose parsing algorithm.

References

[1] Alan Bawden. Reification without evaluation. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 342–349, New York, NY, USA, 1988. ACM Press.

[2] William Clinger. Macros that work. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 155–162, New York, NY, USA, 1991. ACM Press.

[3] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in scheme. *Lisp Symb. Comput.*, 5(4):295–326, 1992.

[4] Robert Bruce Findler, John Clements, Matthew Flatt Cormac Flanagan, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A progamming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002.

[5] Matthew Flatt. Composable and compilable macros: you want it when? In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 72–83, New York, NY, USA, 2002. ACM Press.

[6] Dan Friedman. Object-oriented style. Invited talk at International LISP Conference, October 2003.

[7] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. The MIT Press, Cambridge, MA, 2005.

[8] Kathryn E Gray and Matthew Flatt. Compiling Java to PLT Scheme. In *Proceedings of the 2004 Scheme Workshop*, September 2004.

[9] Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[10] R. Kelsey, W. Clinger, and J. Rees (eds.). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computations*, 11(1), August 1998.

[11] Shriram Krishnamurthi, Yan-David Erlich, and Matthias Felleisen. Expressing structural properties as language constructs. In *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 258–272, London, UK, 1999. Springer-Verlag.

[12] Shriram Krishnamurthi and Matthias Felleisen. Toward a formal theory of extensible software. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 88–98, New York, NY, USA, 1998. ACM Press.

[13] Shriram Krishnamurthi, Matthias Felleisen, and Bruce F. Duba. From macros to reusable generative programming. In *GCSE '99: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, pages 105–120, London, UK, 2000. Springer-Verlag.

[14] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[15] William Maddox. Semantically-sensitive macroprocessing. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1989.

[16] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

[17] Ravi A. Nanavati. Extensible syntax in the presence of static analysis. Master's thesis, Massachusetts Institute of Technology, September 2000.

[18] Benjamin Pierce, editor. *Advanced Topics in Types and Programming Languages*. The MIT Press, Cambridge, MA, 2005.

[19] Francois Pottier and Didier Remy. *The Essence of ML Type Inference*, pages 389–489. In Pierce [18], 2005.

[20] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages, or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[21] Olin Shivers. The anatomy of a loop: a story of scope and control. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 2–14, Tallinn, Estonia, September 2005. ACM Press.

[22] Randall B. Smith and David Ungar. Programming as an experience: The inspiration for Self. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 303–330, London, UK, 1995. Springer-Verlag.

[23] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *PEPM '97: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 203–217, New York, NY, USA, 1997. ACM Press.

[24] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, New York, NY, USA, 1987. ACM Press.

[25] Dale Vaillancourt. ACL2 in DrScheme. `http://www.ccs.neu.edu/home/dalev/acl2-drscheme/`.

[26] Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, June 1987.

[27] Daniel Weise and Roger Crew. Programmable syntax macros. *SIGPLAN Not.*, 28(6):156–165, 1993.

[28] Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. Forwarding in attribute grammars for modular language design. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 128–142, London, UK, 2002. Springer-Verlag.