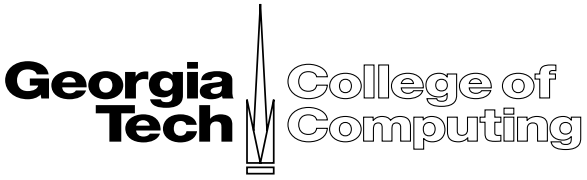

Technical Report GIT-CC-02-48
ftp://ftp.cc.gatech.edu/pub/coc/tech_reports/2002/GIT-CC-02-48

Scheme 2002

Proceedings of the Third Workshop on Scheme and Functional Programming

October 3, 2002
Pittsburgh, Pennsylvania

Olin Shivers, editor

Sponsored by the Association for Computing Machinery's Special Interest Group on Programming Languages (ACM/SIGPLAN)

Preface

This report contains the papers presented at the Third Workshop on Scheme and Functional Programming, on October 3, 2002, in Pittsburgh, Pennsylvania.

Sixteen papers were submitted in response to the workshop's call for papers. Every paper was reviewed by every member of the program committee. This unusually thorough level of reviewing ensured that the review process, which was conducted over the course of two weeks by electronic mail, was both an informed and spirited one. In addition to personally reviewing each paper, the members of the program committee also solicited outside experts for supporting reviews. We are grateful to Will Clinger, Paul Graunke, Shriram Krishnamurthi, Philippe Meunier and Mitch Wand for their service as outside reviewers.

The purpose of the workshop is to discuss experience with, and future developments of, the Scheme programming language, as well as general aspects of computer science loosely centered on the general theme of Scheme. The intention of the steering committee is that the workshop provide an annual focal point where the Scheme community can gather and share ideas: researchers, educators, implementors, programmers, hobbyists, and enthusiasts of all stripes—all welcome.

Paul Graunke, of Northeastern University, capably and expeditiously managed the server infrastructure supporting the reviewing process. (His task was faciliated, in turn, by the fact that this server infrastructure was written in Scheme, as were the scripts used in the production of this workshop proceedings.) Publicity for the workshop was managed by Shriram Krishnamurthi, of Brown University.

I would personally like to thank the workshop steering committee, and particularly Matthias Felleisen, for advice and general counsel during the planning of the workshop.

Olin Shivers
Workshop Chairman
For the program committee

Contents

A library for quizzes

Christian Queinnec
Université Paris 6 — Pierre et Marie Curie
LIP6, 4 place Jussieu, 75252 Paris Cedex — France
Christian.Queinnec@lip6.fr

ABSTRACT

Programming web dialogs is already known to be well served by continuations; this paper presents a continuation-based library for a particular class of web dialogs: quizzes for students. The library is made of *objects* representing the individual questions and of *functional* combinators hiding the *imperative* aspects of page shipping over HTTP and management of continuations. Mixing these three styles provide an elegant framework that fulfills our initial goal. The description of that library is hoped to be helpful for quizzes designers.

1. INTRODUCTION

Last year, we designed a CD-ROM in order to support a college-level course named "Evaluation process" strongly based on the Scheme programming language [1]. This is the first computer science (CS) course delivered to young scientists (eighteen-year old) who still have to choose whether to specialize in maths, CS, mechanics or physics. The goal of the course is to introduce students to recursion, trees, grammars and language interpretation.

The CD-ROM was given to a special group of 45 computer-equipped students who were then able to work at home comfortably with the same means they have access to at the university. Therefore, besides our course material, the CD-ROM also contains copies of the DrScheme programming environment [3] along with some add-ons providing *exercises* and *quizzes*.

An exercise is an assignment that should be performed with the help of the programming environment. A student chooses an exercise (with an additional menu), reads the question (an HTML page displayed by the inner browser of DrScheme), writes the required function(s) (as well as the required testing function(s)), tests them then hit the "check" button which synthetizes a new HTML page with some comments and a mark ranking the provided solution (see Figure 1). Above a given threshold, teachers' solutions are displayed and the student may proceed to the next question.

Third Workshop on Scheme and Functional Programming. October 3, 2002, Pittsburgh, Pennsylvania, USA.

Quizzes are tightly bound to the written course. The course is chopped into a number of HTML pages, each centered on a single topic. For ease of use, these pages are accessed through a mainstream browser such as Explorer or Communicator (the inner browser of DrScheme 103 was not able to handle forms). After every topic, the system proposes various quizzes (as HTML links) checking various levels of understanding. We distinguish level-1 quizzes that are simple applications of the course: they mainly correspond to very simple Scheme questions that do not require the whole power of the DrScheme environment (see Figure 2). Level-2 quizzes strive the student to verbalize its understanding; these questions are not checked but links to appropriate answers are given back. Finally, level-3 quizzes help to understand how the topic contributes to the overall goal of the whole course.

Technically, links to quizzes are served by a web server running as a thread inside DrScheme. A quiz (and the average ten questions it contains) is entirely held in a single file that is simply evaluated by the web server. Continuations [6] are used

- to suspend the server after shipping a page to the student
- and to resume the server with student's answers to the displayed questions.

In order to give a uniform look for the quizzes and to minimize code for the definition of the individual questions of quizzes, quizzes were defined with the help of a library of functions and macros. A question is represented by an object, a quiz is a combination of questions, and combinators embed (and hide) the imperative aspects of page shipping and continuations management.

The rest of the paper presents that library and some elements of the rationale behind it. Section 2 will describe the "question" object, Section 3 will present how questions are composed via appropriate combinators to form quizzes. Section 4 will detail the imperative implementation of combinators and their use of continuations. Finally, Section 5 will conclude.

2. QUESTIONS

A quiz is made of a succession of pages, each of them contains one or more questions. When a question is asked, its terms are generated into HTML. Answers are graded; this grading triggers the synthesis of a good or a bad answer (both in HTML). The grade is a number – positive if the answer is correct, negative otherwise. The HTML produced by a question is limited to the terms or to the good or bad answer without

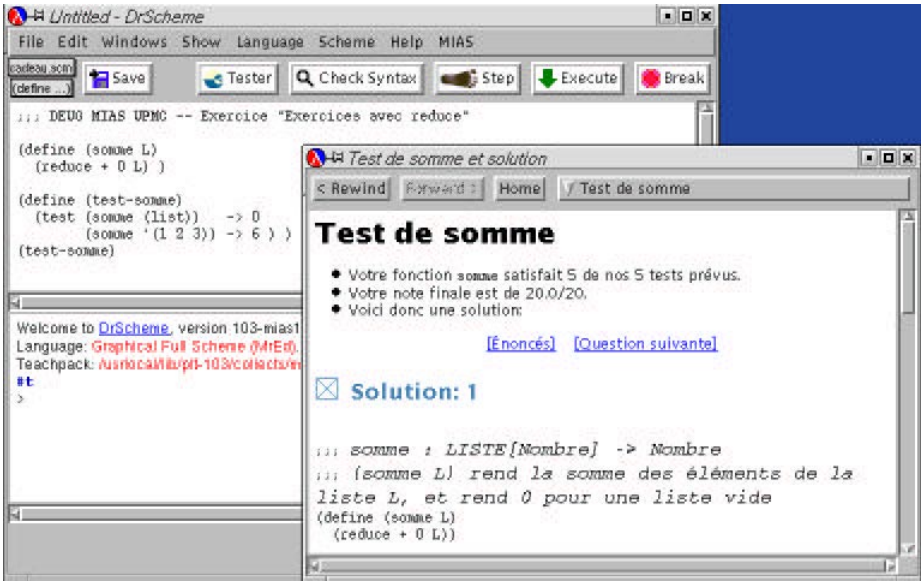

Figure 1: Screen capture of an exercise – The student hit the "*Tester*" (check) button and got a mark good enough to let him see a solution (more than one solution may appear).

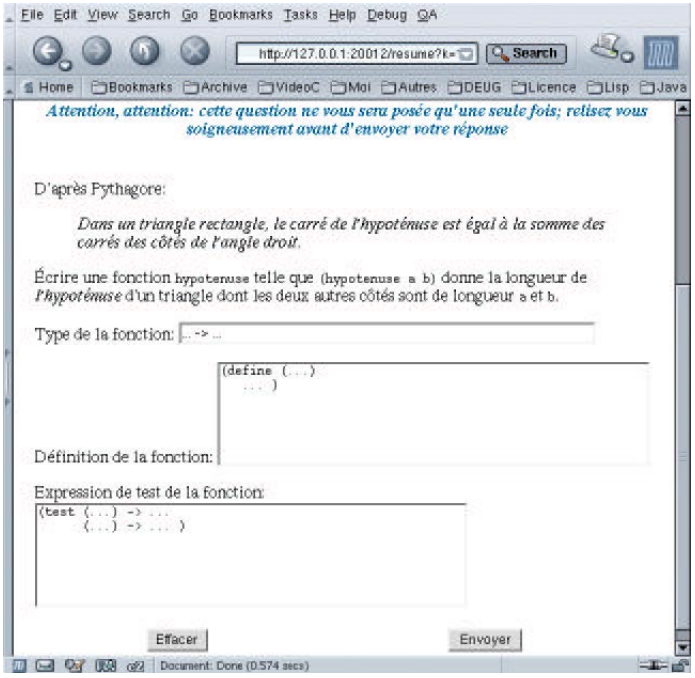

Figure 2: Screen capture of a quiz – This quiz corresponds to a compound question where the student has to write a function that is, its type, its definition and some associated tests.

any adornment. The resulting HTML is one (or more) paragraph(s), not a complete HTML page.

Besides these characteristics, a question also knows how to be logged (when displayed, answered or graded): this is necessary in order to assess students' progress. Of course, all questions are identified with a unique identifier. Logging is done via an HTTP Post request to a centralized logging service feeding a database from which students' progress is deduced.

When a question is displayed again (for instance after a student answers incorrectly), the HTML fields are pre-filled with their former content. It is also possible to blank those fields if required.

We chose to represent questions as objects with fields and methods. Here are the signatures of the methods on the question objects. They are given in a Meroon [5] style (although the current implementation uses message passing rather than Meroon itself).

```
(define-method (id (question))
  returns the identifier (a string) )
(define-method (author (question))
  returns the author (a string) )
(define-method (reset (question))
  erases all already-filled fields )

(define-method
     (html-question (question) interactive?)
  returns an HTML string: the question stem.
  if interactive? is true then generate also the
  HTML INPUT tags (textfield, textarea, checkbox, etc.) )

(define-method
     (report-question (question) nextUrl)
  logs that the question was asked )

(define-method
     (report-answer (question) request)
  logs the answer )

(define-method (verify (question) request)
  grades student's answer (encoded in the HTTP
  request) and returns a number coding the
  grade )

(define-method
     (html-good-answer (question) request a)
  generates a positive comment (an HTML string)
  based on a grade a )
(define-method
     (html-bad-answer (question) request a)
  generates a negative comment (an HTML string)
  based on grade a )
```

We adopt objects to structure behavior sharing. The hierarchy of questions is sketched on Figure 3 where indentation denotes the subclass relationship. The first two classes generate questions offering single or multiple choices. The terms of a question of the third class always display a box where the student types in his answer. The `predicate` field of the question analyzes this answer that is, a string. Some subclasses exist for instance, the `question-regexp` which imposes students' answers to satisfy a given regexp.

Another, more important, subclass is the `question-scheme` that expects students' answers to be legal Scheme expressions. The associated `predicate` then receives that Scheme expression instead of a string (of course, the Scheme expression might be a Scheme string). Among questions expecting a

```
question-qcu with radio buttons
question-qcm with check boxes for multiple choice
question-simple with a box for the answer: a string
  question-regexp the answer must satisfy a regexp
  question-scheme the answer must be a legal S-expression
    question-evaluation
      question-reverse-evaluation
    question-context
question-function with multiple specialized S-expression
boxes
```

Figure 3: Fragment of the class hierarchy for questions

Scheme answer, we have the `question-evaluation` that says "What is the value of *some expression* ?". The `predicate` checks that the students' answer is indeed that *some expression*: the quiz writer just has to mention the *some expression*. Similarly, the `question-reverse-evaluation` says "Give an expression whose value is *some value*". Finally, the `question-context` says "Give an expression using *some expression* and its expected value". There again, the quiz writer just mentions the fragment *some expression* to be used (for instance `(list +)`). Questions of this last class display two boxes related by one predicate.

The last mentioned class, `question-function`, see Figure 2, displays a number of boxes to help a student define a function, its type, its definition, some invocations of this function and their expected values. The quiz writer just has to mention his own version of the specified function.

To sum up, we have a number of questions constructors for various types:

- question without answer
- question with an unchecked textual answer
- question with a regexp-checked textual answer
- question with a checked Scheme answer
- question with unique choice (radio buttons or menus). For instance, what is the arity of *some function* ?
- question with multiple choices For instance, which arity are correct for *some function* ?

We also have a number of refinements for questions with checked answers. Their appearance may differ as well as the grading process. Here are some of our scheme-based questions:

- what is the Scheme encoding of . . . ?
- what is the value of . . . ?
- give a program whose value is . . .
- give a valid program containing . . . , what will be its value ?
- define a function whose specification is . . . , give *n* examples of invocations and the expected values.

For the moment, they cover all our needs for our CD-ROM. We even use a quiz for the registration procedure (when students install the CD-ROM on their home machine in order to log in our databases the sole students we want to assess). We also write a little quiz to define simple quizzes.

3. COMBINATORS

Questions form the basic building blocks for HTML pages, therefore, they should be freely re-usable in various contexts. For instance, when building a quiz, one may want a simple question to be iterated until the student answers it correctly, repeat another question at most twice if badly answered and so on. Questions must be combined in order to form quizzes.

A quiz is a Scheme file that, when evaluated, builds pages with questions and ships them to the student. When the student answers (with an HTTP request), the quiz is resumed at the point where the page was shipped. This is the essence of web continuations [6]. When resumed, the quiz dispatches the request towards the asked questions, gathers the positive/negative comments along with some new or previous questions, packs these all in a new page and ships it to the student. Reaching the end of the file ends the quiz.

In order to be able to re-use questions in various contexts, we separate questions' content from the way questions are asked. In a given context, a question may be mandatory while in another context, the same question may be grouped (and displayed) with three others among which two good answers may be sufficient to proceed past this group of four questions. We must be able to precisely state how the student is led through the quiz depending on his previous good or bad answers.

Here are our current combinators:

```
(ask-only-once question)
(loop-until-verified question)
(loop-at-most n question exhaustion)

(ask-multiple-questions-once questions...)
(ask-multiple-questions n questions...)

(mute-ask-only-once question)
(mute-ask-multiple-questions-once questions...)
```

We group them into three families. The first family just confers a behavior to questions that is, — ask a question only once and proceed to the rest of the quiz even if the answer is incorrect — ask a question until obtaining a correct answer (students complain against this behavior, even though we scarcely used it) — ask a question until obtaining a correct answer or at most *n* times. After *n* failures, the student may proceed to the next question but is given a notice generated by `(exhaustion n)`.

The second family just gathers questions to make them appear as a single one. This is not an easy point since the meaning of the correctness of a group of questions immediately occurs. There is no such problem with the `ask-multiple-questions-once`, it just gathers the comments for the group of questions. The second combinator generalizes the `loop-at-most` combinator with the following behavior: the group of questions is asked again and again but correctly answered questions are removed from the group until the maximal number of iterations is reached or all questions are correctly answered.

The last family corresponds to examination performed on computers. They are similar to the combinators with the same name less the `mute-` prefix. The differences are

- positive/negative comments are not displayed
- students are not allowed to submit more than one answer to any questions (more on that point later).

Here is a contorted example of a quiz that asks a question over and over until the student clicks the "Yes" button. A confirmation is asked for (only once) immediately after. The first two questions are roughly the same but they are defined with alternate means: the first uses a macro while the second uses a function instead. The macro makes available finer details and adopts a uniform keyword-value look and feel.

The third question asks for a Scheme expression returning a number (but at most 2 times). The question generator, named `7-77` (a *local* value) generates a question asking for a program whose value is a number between 7 and 77. If the answer is correct, the quiz ends with a final `cul-de-sac` combinator that displays a specific page telling the student that the quiz is over (this allows us to override the implicit call to `cul-de-sac` with a default message). If the answer is not correct (this is notified with an assignment to the *local* variable named `success?`) the same question generator exactly is called to create a new question that will be asked *ad libitum*.

```
;;;         parameterless question generator
(define-question-generator (understood?)
        type: qcu           ;question with unique choice
          id: "q-qnc-understood1"
     choices: '(yes no) ;rendered as radio-buttons
     correct: 'yes
      author:  "Christian.Queinnec@lip6.fr"
  bad-answer: "Please think harder!"
        text: "This is a quiz, i.e., a dialog
where you get questions that you must answer."
              (p "Do you understand ?") )

(h1 "Welcome to a regular quiz")        ;inter-title

(loop-until-verified                    ;combinator
  (understood?) )                       ;question

(ask-only-once                          ;combinator
 (one-choice-question                   ;question
  "q-qnc-understood2"
  '(yes no)
  'yes
  (div "Do you really understand ?") ) )

(h1 "Welcome to a less simplistic quiz");inter-title

(let again ((success? #t))

  ;;another (hand-made) question generator:
  (define (7-77)
    (reverse-evaluation-question        ;question
     "q-qnc-7-77"
     (+ 7 (random 70)) ) )

  (loop-at-most                         ;combinator
   2
   (7-77)
   (lambda (n)
     (set! success? #f)
     "Alas!" ) )

  (if success?
     (cul-de-sac                        ;combinator
      "The quiz ends here!" )
     ;;otherwise:
     (again #t) ) )
```

So far we have a library of combinators over objects to define quizzes. Regular quizzes writers do not need further details, they just have to pick the right question generator, the

appropriate arguments and the right combinators (the first two questions of the example are examples of regular quizzes). Some of our colleagues even told us that they have the impression of writing Scheme data rather than Scheme code.

4. IMPERATIVE ASPECTS OF COMBINATORS

The combinators hide two very different aspects: they hide continuation management and HTML generation details. Since they manage continuations and HTTP, they require a deeper understanding to be written.

4.1 HTML generation details

Questions only generate fragments of HTML. Between combinator-expressions, there may be other HTML-generating expressions in the quiz (see, for instance, the `h1` function generating a `H1` tag in the previous quiz example; this tag will appear before the HTML stem of the next question). All these HTML fragments are sequentially (imperatively) accumulated in the *communication channel*.

All combinators force an interaction with the student. They gather all HTML fragments so far accumulated, wrap them in a `FORM` tag with a fresh URL bound to the continuation of the quiz (materialized as a "Submit" button), wraps again this form into a complete HTML page (then introducing standard headers, footers, logos, titles, styles, CSS, etc.) and ship it to the student.

Observe that it is up to the final wrapper (a mutable property of the communication channel) to decide how to arrange all these HTML fragments. This isolates questions from their appearance on students' browsers. This also allows us to have a uniform presentation for all pages.

The combinators also solve another problem on the ergonomic side. To consider the quiz as made of a series of question/answer is rather abstract since the quiz has to deal with HTTP where server answers are only displayed when the user requests something. This is the usual inversion of control [4] which we name question/answer (from the view point of the server) or reply/request (from the point of view of the client's browser) where the question is the reply while the answer is the request.

When the server receives an answer, there are various dialogical strategies, see Figure 4:

1. it may reply with a negative comment and a link directing the student back to the old question,
2. it may reply with a negative comment and the old question again (with pre-filled fields),
3. it may reply with a positive comment and a link to the new question,
4. it may reply with a positive comment and the new question,

After some experiments, we chose options 2 and 4 since they minimize the number of clicks. Some of our colleagues do not like option 4 when the comment is too big since it refers to the previous question whose terms are gone and therefore pollutes the terms of the new following question. There again, combinators isolate questions from the way the dialog is chopped into pages.

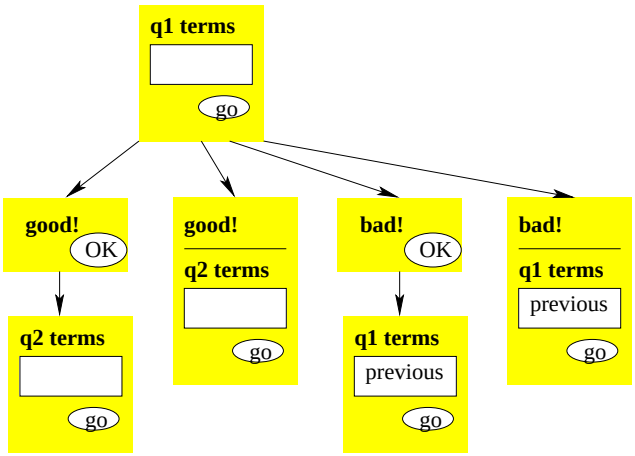

Figure 4: Dialogical split

The imperative side of the communication channel allows pages to share some information: the communication channel plays the role of a sort of shared "session object", but limited to the quiz (as for servlets or ASP dynamic pages). For instance, to be less uniform, messages, button labels and titles are varied. Questions may also put some hints in the communication channel to suggest a title (recall the title of the page is chosen by the HTML wrapper that may pack more than one question on a single page).

For combinators that iterate over a question, the suggested title displays the current trial number and the maximal number of allowed trials.

4.2 Continuation management

Following previous work [6], continuations are mainly put to use via the `show` function that receives an HTML page generator, captures the current continuation, binds it with a fresh URL, feeds the HTML page generator with that URL, ships the obtained HTML page and waits for an answer, that is, an HTTP request that will become the value of the invocation of the `show` function.

Combinators wrap a call to the `show` function with specific management of continuations. These continuations are obtained through the regular `call/cc` however some hackery specific to DrScheme was required since continuations cannot be called out of their birth thread.

On Figure 5 left, the student hits the "submit" button (labeled `go`), resumes the quiz server that decides whether to reply with a positive comment and the new question or to reply with a negative comment and the old question. This latter page is not the same as the first one since the second one contains, in addition, the negative comment. However the continuation of the "submit" button is the same.

This situation must be contrasted with the `mute-` combinators that prevent students from re-submitting to an already answered question. On Figure 5 right, the student answers question 1 then answers question 2 and obtains the terms of question 3, the student then instructs the browser to go back and back to question 1 and tries to change the answer. The combinator detects that and forces the student back to the last

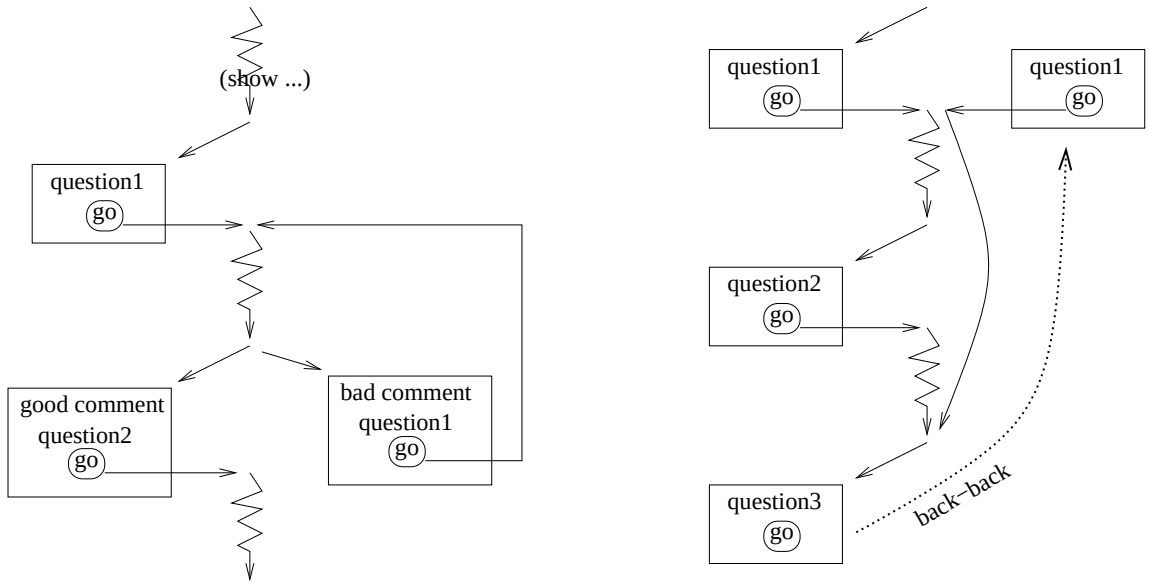

Figure 5: Continuations and dialogs

unanswered question that is, question 3. A fine point is that the invoked continuation leads to the point right before `show`-ing question 3 and not the continuation bound to the "submit" button of question 2 (since the answer of question 2 is already graded).

Still playing with continuations we also introduce a mode where a teacher may see a quiz at once that is in a single page. This is, of course, only possible if the quiz is static enough and linear. The trick is to transform the `show` operator to simply accumulate HTML fragments rather than shipping them. The concatenation of all these fragments is performed at the end of the quiz file.

These various modes are well served by the separation of methods on questions. Answers may be not graded (when the teacher wants to have a global look to the entire quiz or wants a paper copy to circulate), answers may be graded without emitting any comment (this is the examination mode).

5. CONCLUSIONS AND PERSPECTIVES

Concerning web continuations, the paper does not present new results. It only shows how they may be put to work for quizzes. Only the trick concerning the continuations just before or after the shipping of a page in the implementation of the `mute-` combinators is new.

Therefore, the paper is centered on the main features of the quiz library that had several goals:

1. **separation of concerns:** A question writer just has to understand how to build questions. These questions may then be put in a big database (correctly indexed to let them be easily retrieved); this is future work!

   A quiz writer just has to understand combinators in order to assemble questions into dialogs. A quiz programmer may dynamically builds thematic quizzes extracted from the previous database. A special quiz may be designed to build quizzes interactively.

   An HTML designer just has to change the HTML generation part of questions and combinators to alter the look.

   A web-dialog designer (just) has to understand continuations to implement other kinds of dialog. For instance, students asked us in the examination mode (the `mute-` combinators) to be able to see all questions in advance that is, to only prevent submitting more than once to any given question.

2. **nice multi-paradigmatic fit:** Programmation requires mastering various programming styles making some tasks easier. Refining questions is well served by objects and classes. Combinators are nice means to assemble questions to form dialogs. The sequentiality of web interactions via HTTP forces an imperative view for continuations and HTML fragments accumulation.

This is the third version of that library, each version has improved the separation of concerns and adopted the most appropriate framework to deal with the new concerns. The current library has been stable for the last year. Quizzes may have very reactive behaviors and are far more easier to define and manage compared to the very static tools of generic authoring systems. In such systems, a quiz is usually defined with a number of boxes, radio-buttons, menus to fill, click or unroll. The resulting quizzes are, most of the time, sequential and made of independent questions that are syntactically graded (syntactically since there is no relationship between the label of a radio-button and the fact that this radio-button should be pressed for a correct answer).

In our system and since we are teaching a language with an easy to use `eval`uator, questions may be specified in a more semantical way. Since the quiz is a program, it may use the full

power of the underlying language and use conditional or recursion as shown in the previous quiz example where students with good answers may terminate quickly whereas others are provided fresh exercises until they got one right.

6. ACKNOWLEDGMENTS

Many thanks to the numerous (and anonymous as well) reviewers whose comments terrificly improves the paper.

7. REFERENCES

[1] A. Brygoo, T. Durand, P. Manoury, C. Queinnec, and M. Soria. Experiment around a training engine. Complete version of [2], Oct. 2002.

[2] A. Brygoo, T. Durand, P. Manoury, C. Queinnec, and M. Soria. Experiment around a training engine. In *IFIP WCC 2002 – World Computer Congress*, Montreal (Canada), Aug. 2002. IFIP.

[3] R. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. Drscheme: A programming environment for scheme. *Journal of Functional Programming*, 2001.

[4] M. A. Jackson. *Principles of Program Design*. Academic Press, 1975.

[5] C. Queinnec. Designing MEROON v3. In C. Rathke, J. Kopp, H. Hohl, and H. Bretthauer, editors, *Object-Oriented Programming in Lisp: Languages and Applications. A Report on the ECOOP'93 Workshop*, number 788, Sankt Augustin (Germany), Sept. 1993.

[6] C. Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *ICFP '2000 – International Conference on Functional Programming*, pages 23–33, Montreal (Canada), Sept. 2000.

Incorporating Scheme-based Web Programming in Computer Literacy Courses

Timothy J. Hickey *
Department of Computer Science
Brandeis University
Waltham, MA 02254, USA

Abstract

We describe an approach to introducing non-science majors to computers and computation in part by teaching them to write applets, servlets, and groupware applications using a dialect of Scheme implemented in Java. The declarative nature of our approach allows non-science majors with no programming background to develop surprisingly complex web applications in about half a semester. This level of programming provides a context for a deeper understanding of computation than is usually feasible in a Computer Literacy course. The course does not require the students to download any software as all programming can be done with Scheme applets. The instructor however must provide a Scheme server which will run the students' servlets.

1 Introduction

There are two general approaches to teaching a Computer Literacy class. The most common approach is a broad overview of Computer Science including hardware, software, history, ethics, and an exposure to industry standard office and internet software. On the other end of the spectrum is the class that focuses on programming in some particular general purpose language, (e.g. Javascript [12], Scheme[5], MiniJava[11]).

The primary disadvantage of the breadth-first approach is that it tends to offer a superficial view of computing. The depth-first programming approach on the other hand often requires a substantial effort just to learn the syntax of the language and the semantics of the underlying abstract model of computation, leaving little time to look at other aspects of computing such as internet technology or computer architecture.

Several authors have recently proposed merging these two approaches by using a simpler programming language (e.g. Scheme[5], [6], [7]) or by using an internet-based language (e.g. Javascript[12], MiniJava[11]).

In this paper we describe a five year experiment in combining these two approaches in a Computer Literacy course at Brandeis University (CS2a: Introduction to Computers). We deviate from many Computer Literacy courses in that we spend very little time discussing the standard application programs (e.g. word processors, spreadsheets, email, instant messaging, file sharing, image processing etc.) It has been our experience that students are able to learn how to use most of these programs on their own and that use of these applications does not generally require a deep understanding of computation. In a phrase, we don't teach them what they are going to learn by themselves anyway.

The CS2a:Introduction to Computers course teaches programming concepts and uses a small (but powerful) subset of Jscheme[2] – a Java-based dialect of Scheme. The tight integration of Java with Jscheme allows it to be easily embedded in Java programs and hence makes it easy for students to implement servlets, applets, and other web-deliverable applications. Jscheme is an implementation of Scheme in Java (meeting almost all of the requirements of the R4RS [4] Scheme standard). It also includes two simple syntactic extensions:

- **javadot notation**: this provides full access to Java classes, methods, and fields
- **quasi-string notation**: this simplifies the process of generating HTML.

The javadot notation provides a transparent access to Java and the quasi-string notation provides a gentle

*This work was supported by the National Science Foundation under Grant No. EIA-0082393.

path from HTML to Scheme for novices. It also provides a convenient syntax for generating complex strings of other sorts (such as SQL queries). These two extensions will be discussed at length below.

Jscheme can be accessed as an interpreter applet (running on all Java-enabled browsers) or as a Java Network Launching Protocol (JNLP) application. Both of these provide one click access to the Jscheme IDE from standard browsers. It can also be downloaded as a jar file and run from the command line as a standard read-eval-print-loop program.

Jscheme has been built into a Jakarta Tomcat webserver as a webapp which allows students to write servlets and JNLP applications directly in Jscheme. This webserver typically runs on the instructor's machine, but students can easily download and install the server on their home/dorm PCs as well.

In the sequel, we explain, in detail, how Jscheme can be used to teach non-science majors in a large lecture class how to build servlets and applets in a six week section of a Computer Literacy course. The approach described here is very similar to the approach used in the Autumn 2001, "Introduction to Computers" course at Brandeis University, but it reflects changes that will be incorporated in the next year's version of the course. The course and the underlying language have been evolving steadily over the past five years and will likely continue to do so.

This approach to teaching Computer Literacy is feasible because of the declarative style of programming that is possible in Scheme, together with the extremely simple syntax and semantics of Scheme.

We posit that this web-programming based approach would work with other declarative languages (e.g. Haskell or Prolog), but would be infeasible with imperative languages such as Java or Perl. Scheme however is ideally suited to this application because of the relative simplicity of its syntax and semantics, both of which can be stumbling blocks for novice programmers.

Although the particular languages and techniques that we use may not be the best match at other institutions, we feel that the general approach could be easily replicated using other languages provided care is taken to make the syntax and semantics that must be learned as simple as possible.

2 Related Work

The need for a simple, but powerful, language for teaching introductory CS courses has been discussed recently by Roberts [11] who argues for a new language, Minijava, that provides both a simpler computing model (e.g. no inner classes, use of wrapper class for all scalar values, optional exception throwing) and a simpler runtime environment (e.g. a read-eval-print loop is provided).

Jscheme can be viewed as an even more radical simplification of Java in that it replaces the syntax of Java with the much simpler syntax of Scheme while maintaining access to all of the classes and objects of Java.

Another recent approach for introductory courses is to use Javascript to both teach programming concepts and to provide a vehicle for discussing other aspects of computing such as the internet and web technology. For example, David Reed proposes teaching a course [12] in which about 15% of class time is devoted to HTML, 50% to Javascript, and 35% to other topics in computer science. Our approach follows a similar breakdown but also allows the students to build servlets, applets, and GUI-based applications.

A third related approach is to teach Scheme directly as a first course. The MIT approach, pioneered by Abelson and Sussman [1], is not suitable for non-science majors as it requires a mathematically sophisticated audience. The approach being developed by the PLT group [5], [6],[7], on the other hand, provides a rigorous introduction to Scheme programming but is designed to be accessible to students from all disciplines.

In our aproach, we provide an introduction to only a subset of the language (for example, introducing lists only toward the end). We start by introducing some high-level declarative libraries for teaching an event-driven model of GUI construction. The Scheme section of the course requires only about 6 weeks. This leaves half of the course for standard Computer Literacy topics.

3 Goals, Syllabus, and Rationale

Our main goal in teaching a Computer Literacy course is to help the students gain a broad understanding of digital computation. It is our feeling that Computer Literacy courses are most effective if they focus on the fundamental mechanisms of computing at all levels and if they ground this theoretical material by requiring the students to build programs using these fundamental concepts.

The syllabus covers the mechanisms underlying CMOS gates and VLSI, the structure and interpretation of assembly language, the design of simple GUI-based applications, the mechanisms underlying servlets (including counters, logs, and auto-generated email), the basic design and structure of the internet, and the limits of computers (e.g. the Halting problem and the Turing test).

We test their understanding of this material using weekly quizzes, biweekly homework assignments, and a final exam in which they must write and/or trace programs at these various levels (from semiconductors to servlets). Before delving into a detailed description of the curriculum we first explain what we do not cover and provide some justification for these choices.

This course also does not delve very deep into the soft aspects of Computing. These topics are covered in a companion course (CS33b: Internet and Society), which is focused primarily on the social, ethical, legal, economic, political and aesthetic aspects of computers. It is our opinion that these issues are best taught in an interdisciplinary context. Indeed, the CS33b course is currently taught by a dozen instructors from half a dozen different departments.

The course does not teach algorithms and data structures. Although the students do learn to trace through the execution of fast-exponential procedures, gcd calculators, and the "map" function, we do not teach them to use computers for problem solving. Thus we do not ask them to write sorting procedures or programs to find average grade scores, etc.

We **do** teach "reactive" programming in this course, i.e. programs that interact with the user (through GUIs or HTML forms) and use the user-supplied information to generate responses and perform simple actions (logging, sending email, updating counters, performing simple calculations and tests). We also teach the students to understand how to trace recursive programs which is a far easier task than learning how to write recursive programs. More precisely,the students are required to be able to write applets and servlets in three languages (HTML, CSS, Scheme) and to trace programs in two additional "languages" (pcode assembly language, and CMOS circuit diagrams).

The goal in teaching them to write "reactive" programs and to trace recursive programs is to help them understand the deeper issues of computation more clearly. For instance, one of the applet programs we present is a simple "Psychiatrist" simulator which they are encouraged to modify. This provides a context for a deeper discussion of artificial intelligence, ethics, and the Turing problem. For another example, when we discuss the substitution model of Scheme the students are required to trace recursive programs with function parameters (e.g. map). This paves the way for a discussion of the Halting problem. We consider the consequences of extending the Scheme language by adding a primitive procedure `(halts? F X)` which returns true if `(F X)` eventually returns an answer and false if it throws an exception or does not return. In particular, we look at the following program:

```
(define (skeptic Q)
      (if (halts? Q Q) (skeptic Q) 'ha))
(skeptic skeptic)
```

The trace of `(skeptic skeptic)` yields the expected contradiction which then leads to a discussion of the limits of computation. It is true that the `skeptic` example only makes sense in the context of a Scheme which provides source code access to all procedures and closures, but the impossibility of adding a recursive "halts?" procedure still illustrates well the limits of computation. We usually couple this lecture with a classroom exercise in which the students must prove that the instructor can not tell the future. The proof consists of asking the instructor to predict the student's behavior using the same strategy as the "skeptic" procedure.

A rough outline of the syllabus, which shows the context of the web-programming part of the course is shown below.

- 1 week **HTTP** and the structure of the Internet: IP addresses, ports, sockets, services, routers, gateways. Use of telnet, dig, traceroute, ping, portscan to illustrate these issues.
- 2 weeks **HTML/CSS** – the thirty non-style HTML tags and 10 basic CSS properties. Copyright issues.
- 3 weeks **Scheme Servlets** – quasi-string notation, abstraction, conditional execution, lists, file I/O, email, database access. Security, privacy, cookies, ethics.
- 3 weeks **Scheme Applets/Groupware** – GUI components, layout, callbacks, animation, networking primitives, groupware components. Doctor applet, Turing Test. Halting problem. Substitution model. Software licenses.
- 1 week **Assembly Language/Pcode** - von Neumann architecture, memory-mapped peripherals, memory, speed, bandwidth, cacheing, super-scalar architectures. Operating Systems, file systems, time sharing, ...
- 1 week **CMOS/Logic Circuits** - semiconductors (P/N-type), gates, circuits, adders, latches and bits.

Observe that the course contains a signficant amount of non-Scheme material that would be found in most typical Computer Literacy courses (such as copyright issues and ethical questions dealing with servers), but with this programming-based approach these issues are more meaningful as the students are able to write servers that create logs and must deal with the resulting ethical questions.

4 Courseware

The main language used in the course is Jscheme[1] [2, 3, 8], an open source implementation of Scheme in Java.

[1] http://jscheme.sourceforge.net

```
SYNTACTIC CONSTRUCT          JAVA MEMBER        EXAMPLE
"." at the end               constructor        (Font. NAME STYLE SIZE)
"." at the beginning         instance method    (.setFont COMP FONT)
"." at beginning, "$" at end instance field     (.first$ '(1 2))
"." only in the middle       static method      (Math.round 123.456)
".class" suffix              Java class         Font.class
"$" at end, no "." at beg.   static field       Font.BOLD$
"$" in the middle            inner class        java.awt.geom.Point2D$Double.class
"$" at the beginning         packageless class  $ParseDemo.class
"#" at the end               access private data Symbol.#
```

Figure 1: Java reflectors in Jscheme

It is almost completely compliant with the R4RS standard [2] [4] and also provides full access to Java using the Java Reflector syntax shown in Figure 1. Jscheme also provides full access to Java thread and exception handling. The following example illustrates the ease with which one can access Java libraries in Jscheme. It implements a simple multi-threaded "echo service" on a specified port and catches/reports any errors that may arise in each thread:

```
(define (echoserver N)
 (let ((SS (java.net.ServerSocket. N)))
  (let loop ()
   (let ((S (.accept SS)))
    (.start
     (java.lang.Thread.
      (lambda()
       (tryCatch
        (let*
          ((in (java.io.BufferedReader.
                (java.io.InputStreamReader.
                 (.getInputStream S))))
           (out (java.io.PrintStream.
                 (.getOutputStream S))))
         (.println out (.readLine in))
         (.close S))
        (lambda(e)
          (.println java.lang.System.out$
                    (.toString e))))))))
   (loop))))
```

The course uses a small but powerful subset of Scheme and also relies on only a few selected Java reflectors and a small GUI-building library. For control flow and abstraction it uses `define, set!, lambda, if`,`cond`, `case, let*`. For primitives, it uses arithmetic operators and comparisons, a simple GUI-building library (providing declarative access to Swing components, events, and layout managers).

[2] strings are not mutable, and call/cc is only implemented for try/catch like applications

4.1 Scheme Servlets

Files which appear in the Jscheme webserver student directory with the extension ".servlet" are treated as Jscheme expressions which are evaluated to generate the html to send back to the client. After working with this model for a while, we found that the need to combine scheme and text resulted in programs containing large numbers of string-append's and quoted strings (with many quoted quotes). In response to this somewhat confusing syntax, we introduced a slight syntactic extension to Scheme which allows curly braces {} to be used in place of double quotes for strings. Moreover, inside a {} string, any scheme expressions appearing within square brackets [], are evaluated and appended into the string. These two devices make use of the unassigned outfix operators [] and {}, and allow for a more concise method for constructing strings in Scheme. We call this quasi-string notation[3]

For example, using quasi-string notation we can write

```
(define (my-li NAME IMAGEFILE COST)
 {<div style="background:rgb(0,150,150)">
   <table  width="100%">
     <tr><td>
      <a href="[IMAGEFILE]">
       <img src="[IMAGEFILE]"
         alt="[NAME]" width="150"></a><br>
    </td><td> <h1 style="background:lightgreen;
                        color:black">[NAME]</h1>
    </td><td style="text-align:right">
        Cost: $[COST] </td></tr></table>
  </div>  <br>   <br>   <br>
 })
```

which is equivalent to the following (less elegant) standard Scheme expression. Note in particular the confusion that arises from the need to quote double quotes. In the quasi-string syntax, it is much easier to verify the syntactic correctness of the resulting code.

[3] The quasi-string notation is a syntactic variant on Bruce R Lewis' Beautiful Report Language (BRL) Syntax. Our approach is based on the quasiquote/unquote approach for constructing lists in Scheme.

```
(define (my-li NAME IMAGEFILE COST)
 (string-append
  "<div style=\"background:rgb(0,150,150)\">
   <table  width=\"100%\">
     <tr><td>
      <a href=\""
 IMAGEFILE
      "\">
       <img src=\""
 IMAGEFILE
       "\"
          alt=\""
 NAME
       "\" width=\"150\"></a><br>
    </td><td> <h1 style=\"background:lightgreen;
                  color:black\">"
 NAME
      "</h1>
    </td><td style=\"text-align:right\">
      Cost: $"
 COST
     " </td></tr></table>
  </div>  <br>   <br>   <br>
 "))
```

The quasi-string notation is similar to the quasiquote syntax used to construct s-expressions in Scheme.

4.1.1 Dynamic content

The first non-trivial examples of servlets that we provide are servlets that include runtime generated data (such as the current date, or information from the HTML headers, like the client operating system). For example, by enclosing their HTML in curly braces, changing the extension from html to servlet, they can add this dynamic content to their page just by including the `[(java.util.Date.)]` expression into their HTML.

```
{<html>
   <head><title>Date/Time</title></head>
   <body>
     Current local time is
       [(java.util.Date.)]
   </body>
</html>}
```

Evaluating this expression yields

```
<html>
  <head><title>Date/Time</title></head>
  <body>
     Current local time is
     Fri Sep 07 09:33:30 EDT 2001
  </body>
</html>
```

These small syntactic changes provide a gentle introduction to servlets that, as we will show below, leads naturally to abstraction, conditional execution, and expression evaluation.

4.1.2 Introducing Abstraction

Once the idea of dynamic content is clearly established, we move on to abstraction and show how to use the "define" form to create "scheme tags." This simple and powerful idea only requires an understanding of the substitution model of scheme evaluation, and yet allows students to start writing and sharing new HTML tag libraries, written in Scheme. For example, Figure 2 shows a typical and simple library that includes a generic webpage procedure and a captioned image procedure.

```
;; loadmylib.servlet
   (define (cimg C I) ;; captioned images
    {<table border=5>
        <tr><td>
          <img src="[I]" alt="[C]">
       </td></tr>
        <tr><td>[C]
      </td></tr> </table>})

   (define (generic-page Title CSS Body)
     {<html>
         <head><title> [Title]</title>
          <style type="text/css" media="screen">
            <!-- [CSS] --></style></head>
       <body> [Body]</body>
     </html>})
```

Figure 2: An HTML abstraction library

An example of the use of this simple library is shown in Figure 3. The benefits of this sort of abstraction become even greater when the abstractions start using sophisticated inline-CSS style attributes to create a highly stylized HTML components.

```
(begin
  (generic-page    "Pets"
    "body {background:black;color:white}
     h1{border: thick solid red}"

    {<h1>Pets</h1>
      [(list
         (cimg "Snappy" "snappy.jpg")
         (cimg "Pepper" "pepper.jpg")
         (cimg "Missy"  "missy.jpg")
         (cimg "Kitty"  "kitty.jpg")
         (cimg "Tarzan" "dog17.jpg"))]
    })
```

Figure 3: Using HTML abstraction libraries

This technique for abstracting HTML is well-known is Lisp/Scheme web programming (e.g. LAML[10], BRL[4]) and is similar to Server-Side Includes in JSP[5] or the publishing model of the Zope environment[6].

4.1.3 Introducing User Interaction

The next pedagogical step is to introduce the notion of using HTML forms to send data from the user to the servlet.

To simplify the computational model for novice students, Jscheme provides easy access to form parameters using the `(servlet (p1 p2 ...) ....)` macro which binds the variables `p1,...` to the strings associated with the form parameters of the same names. This allows one to easily write servlets that process form data from webpages. This also proves to be a good time to introduce the notion of conditional execution (using if, cond, and case):

```
(servlet (password bg fg words)
 (case password
   ((#null)  ; first visit to page, make form
    (generic-page {color viewer form} {}
    {<h1>pw-protected color viewer</h1>
     <form method=post action="demo1.servlet">
      pw <input type=text name="pw"><p>
      bg <input type=text name="bg"><p>
      fg <input type=text name="fg"><p>
      text<textarea name="words">
      Enter text to view here</textarea>
      <input type=submit>
    </form>}))

  (("cool!") ;; correct pw, process data
   (generic-page "color viewer"
    "body {background:[bg];color:[fg]}"
     words))

 (else ;; incorrect password, complain!
   (generic-page "ERROR"
     " body {color:red;background:black}"
     {<h1>WRONG PASSWORD<h1>
       Go back and try again!}))))
```

Figure 4: A password protected page

For example, after a week of HTML instruction we have found that beginning students easily create HTML forms and it is then a small step to the servlet in Figure 4 which either generates a form or generates a response to the form, depending on whether the form parameter has been given a value by the browser.

[4] http://brl.sourceforge.net
[5] http://java.sun.com/products/jsp
[6] http://www.zope.org

4.1.4 Expression Evaluation

The next step is to introduce numerical computation into servlets. An example, of the type of program the students are able to construct at this level is shown in Figure 5 below.

```
(servlet (inches pounds)
 (if (equal? inches #null)
   ;; first visit to page, create form
    (generic-page {color viewer form} {}
    {<h1>BMI Calculator</h1>
     <form method=post action="bmi.servlet">
      height:
      <input type=text name="inches"> inches<br>
      weight:
      <input type=text name="weight">pounds<br>
      <input type=submit>
    </form>})
  ;; else  compute BMI, display results
   (let*(     (h-in-m (* inches 0.0254))
              (w-in-kg (/ pounds 2.2))
              (bmi  (/ w-in-kg (* h-in-m h-in-m))))
   (generic-page "Body Mass Index"
     " body {background:rgb(255,235,215)}"
     {<h1>Body Mass Index<h1>
      With a height of [inches] inches and
      a weight of [pounds] pounds, your
      Body Mass Index is [bmi] <br>
     Note: a BMI over 25 indicates you may be
     overweight, while a BMI over 30 indicates
     that your weight may cause significant health
     problems!}))))
```

Figure 5: A sample quasi-string servlet

This requires two new ideas:

- evaluation of arithmetic s-expressions[7]
- introduction of intermediate variables using `let*`

This is admittedly a big step. At this point we review the substitution model to explain how expression evaluation proceeds, and we introduce an environment model to explain the semantics of the `let*` expression.

For students to be able to write this type of servlet they need to learn to use prefix Scheme arithmetic expressions and to use the `servlet` and `case` macros.

4.1.5 System Interaction

We have also added a few additional primitives for writing or appending scheme terms to a file, and for reading

[7] The servlet macro automatically converts numerals to Java numbers, thus `pounds` and `inches` are numbers

a file either as a string or as a list of scheme terms. These allow students to easily write logs and counters as in Figure 6. This example also shows the `send-mail` procedure which allows the students to specify the "from", "to", "subject" fields and give a quasi-string for the body.

```
(servlet()
  (let* ((c (read-from-file "counter" 0))
         (d (list c (Date.)
                  (.getRemoteHost request))))
    (write-to-file "counter" (+ 1 c))
    (append-to-file "log" d)
    (send-mail
      "tjhickey@brandeis" "nobody@brandeis"
      "counter" {You got a hit: [d]!})
  {<html><body>
    This list has been visited by <xmp>
   [(read-string-from-file "log" "")</xmp>
  and you are visitor number [(+ 1 c)]
```

Figure 6: Logs and Counters in test.servlet

In order to simplify the problem of associating log and counter files to servlets, these primitives read and write from files whose prefix is the name of the servlet. Thus, for the log and counters example, the "log" file would be named "test.servlet_log" and the counter would be "test.servlet_counter". The students can also use library procedures that allow absolute addresses for files, but this is discouraged.

4.1.6 Data Structures and `map`

Students naturally want to handle list-style data (e.g. multiple checkboxes in form data). This leads naturally into a description of "map" and also to table abstractions. We find it useful to introduce map before car, cdr, cons, since it provides a powerful and intuitively clear operation and does not require an understanding of recursion. Moreover, as the examples in Figure 7 below illustrate, the `map` procedure gives the students most of what they need to handle lists of data values. There is also a `map*` procedure which uses a generalized map that converts Java collection objects into lists, and hence can be used with arrays, hashtables, etc.

```
(define (li x) {<li>[x]</li>})
(define (lis L) (map li L))
(define (ul L) {<ul>[(lis L)]</ul>})
(define (ol L) {<ol>[(lis L)]</ol>})
(define (td X) {<td>[X]</td>})
(define (tds Ts) (map td Ts))
(define (tr Ts) {<tr> [(tds Ts)] </tr>})
(define (trs Rs) (map tr Rs))
(define (table Rs) {<table> [(trs Rs)] </table>})
```

Figure 7: Generating lists and tables

4.2 Scheme Applets

After spending about three weeks studying servlets, we turn to client-side computing. The tomcat server has been configured so that any scheme program that ends with ".applet" is transformed into a Jscheme applet and runs on the client's browser. Likewise, Jscheme programs that end in "snlp" are converted into Java Network Protocol format which will be automatically downloaded and run in the Java Web Start plugin.[8].

```
"John Doe"
"http://www.johndoe.com"
"years->secs calculator"
"Convert age in years to age in seconds"
"http://www.johndoe.com/jd.gif"

(jlib.JLIB.load)
(define t (maketagger))
(define w (window "years->secs"
  (menubar
   (menu "File"
    (menuitem "quit"
     (action (lambda(e) (.hide w))))))
  (border
   (north (label "Years->Seconds Calculator"
                 (HelveticaBold 60)))
   (center
    (table 3 2
     (label "Years:")
     (t "years" (textfield "" 20))

     (label "Seconds:")
     (t "secs" (label ""))

     (button "Compute" (action(lambda(e)
       (let*
          ((y (readexpr (t "years")))
           (s  (* 365.25 24 60 60 y)))
         (writeexpr (t "secs") s)))))))))
(.pack w)
(.show w)
```

Figure 8: A sample SNLP program

Jscheme has also been extended to allow students to learn to implement simple programs with Graphical User Interfaces. We have written a library, JLIB, that provides declarative access to the AWT package (There is also a version for the Swing package). An example of a simple Scheme program using this library is shown below in Figure 8. The first five lines of the program listed above are strings that provide documentation about this program which is required by the Java Network Launching Protocol (JNLP).

[8] http://java.sun.com/products/javawebstart

4.2.1 JLIB

The JLIB model is based on five fundamental concepts:

- COMPONENTS – there are a small number of ways to construct basic components (buttons, windows, ...)
- LAYOUTS – there are a small number of ways to layout basic components (row, col, table, grid, ...)
- ACTIONS – there is a simple mechanisms for associating an action to a component
- PROPERTIES – there are easy ways for setting the font and color of components
- TAGS – this is a mechanism for giving names to components while they are being laid out.

Another key idea is that operations on all components should be as uniform as possible. For example, there are procedures "readstring" and "writestring" which allow one to read a "string" from a component, and write a string onto a component. Thus "writestring" can change the string on a label, a button, a textfield, a textarea. It can also change the title of a window or add an item to a choice component. Likewise, readstring returns the label of a button, the text in a textarea or textfield, the text of the currently selected item in a choice, the title of a window, and the text of a label. The readexpr and writeexpr procedures are similar, but they allow reading and writing of Scheme expressions on GUI components. For example, the following snippet of code defines a button which changes state when pushed:

```
(define (flip x)
  (case x
       (("on") "off")
       (("off") "on")))
(define B
  (button "off" (action (lambda(e)
      (writestring B (flip (readstring B)))))))
```

JLIB provides procedures for each of the main GUI widgets (window, button, menubar, label) and it also provides procedures for specifying layouts (e.g. border, center, row, col, table). The first few arguments of these procedures are mandatory (e.g. window must have a string argument, textfield requires a string and a integer number of columns). The remaining arguments are optional and can appear in any order. Examples are fonts, background colors, and actions.

The JLIB package provides a "tagger" procedure which allows one to give names to components *in situ*

- `(define t (maketagger))` creates a tagger,
- `(t NAME OBJ)` assigns the NAME to the OBJ and
- `(t NAME)` looks up the OBJ with that NAME.

This makes the code more declarative because the name for a textfield appears with its constructor in the expression that creates the GUI.

4.2.2 Graphics and Animation

We also provide a simple graphics library providing access to a canvas with an offscreen buffer. The drawing primitives are the Java instance methods of the java.awt.Graphics class. The "canvas" procedure is a JLIB procedure that creates a canvas with an offscreen buffer accessed by `(.bufferg$ c)` and which can be drawn to the screen using `(.repaint c)`. The program in Figure 9 shows a simple example drawing a red ball moving across a blue background.

```
(jlib.JLIB.load)
(define c (canvas 400 400))
(define w (window "graphics1"
 (border
   (center c)
   (south
      (button "draw"
        (action (lambda(e)
         (run-it drawballs))))))))

(define (run-it F) (.start (Thread. F)))

(define (drawballs) (drawball 200))

(define (drawball N)
  (define g (.bufferg$ c)) ;get graphics object
  (.setColor g blue)
  (.fillRect g 0 0 1000 1000) ;; clear background
  (.setColor g red)
  (.fillOval g N N 100 100)    ;draw red disk
  (.repaint c) ; copy buffer to screen
  (Thread.sleep 100L) ;; pause 0.1 sec
  (if (> N 0) (drawball (- N 1))) ;; loop
  )

(.resize w 400 400)
(.show w)
```

Figure 9: Graphics programming

The `run-it` procedure is used when the students write animations. They seem to understand the notion of multi-threaded programming in the context of having several animations each running in their own thread [9]

[9] We also have a version of run-it that looks for errors and reports them in a debugging window.

4.3 Networking Abstractions

After spending two weeks mastering the JLIB library we introduce network programming using a simple model where applets communicate by sending scheme terms to each other through a `group-server`. Since applets are only able to open sockets on their host server, we must run the `group-server` on the same machine that manages the students' applets. The students connect to this `group-server` using the `make-group-client` procedure:

```
(define S
  (make-group-client Name Group Host Port))
```

This creates an object, `S`, that can communicate with the group-server. To send the scheme terms `key b c ...` to the server, one evaluates the expression

```
(S 'send key b c ...)
```

The first term, `key`, is used as a filter. Indeed, the group-server bounces back every message it receives to all the members of the group. A member can specify how to handle a message using the `add-listener` method

```
(S 'add-listener key
    (lambda (key . restargs ) ...))
```

This method indicates that the indicated procedure should be called on each message that arrives from the server with the specified key.

This model builds on the student's experience with callbacks in GUI's and with reading/writing on GUI components. The analogy is that "send" is like writing to a component and "add-listener" is like adding an action.

An example of the kind of applet that is explained in class is the chat applet shown in Figure 10. In the most recent semester we did not require students to write an applet using networked communication, but several students chose to write such applets for their final project. The best example was a pictionary program which allowed any number of students to join in a game of pictionary using a shared whiteboard as well as private and group chats. This program was written by a student with no previous programming experience and made use of almost all of the examples we had given previously in the course.

In the coming year we plan on introducing networked communication using the notion of groupware components. These are textareas and canvases which are shared among several users on the network. This approach may provide an even simpler model of network programming that builds more directly on their understanding of GUI programs.

```
(jlib.JLIB.load)
(jlib.Networking.load)
(define (chatwin
          UserName ChatGroup Host Port)
    (define t (maketagger))
    (define S (make-group-client
         UserName ChatGroup Host Port))
    (define w (window "test"
      (col
        (button "quit" (action (lambda (e)
              (S 'logout) (.hide w))))
        (t "chatarea" (textarea 20 50))
        (t "chatline" (textfield "" 50
         (action (lambda(e)
          (S 'send "chat" (string-append
             UserName ": "
             (readstring (t "chatline"))))
          (writeexpr (t "chatline") "")
        )))))))
      (S 'add-listener "chat" (lambda R
       (appendlnexpr (t "chatarea") R)))
      (.pack w) (.show w)
      w)
(define (rand N)
       (Math.round (* N (Math.random))))
(chatwin
      (string-append "user-" (rand 1000))
      "chat"
      (.getHost (.getDocumentBase thisApplet))
      23456)
```

Figure 10: A multi-room chat program

5 Student Evaluation Strategies

We have used several techniques to accommodate the non-science students that are a majority in this class. The homework assignments allow students to exercise their creativity in creating a web artifact (webpage, servlet, applet, application) which must meet some general criteria. For example, in one assignment they are required to create a servlet that uses several specific form tags (in HTML) and generates a webpage in which some arithmetic computation is performed. This encourages a bricolage approach to learning programming concepts which seems to appeal to non-science majors.

The course features weekly quizzes which take an opposite approach. The students are shown a simple web artifact and asked to write the code for it during a twenty minute in-class quiz. This practice helps keep the students from falling behind in the class and also helps counterbalance the openness of the homework assignments.

The final exam is based on the weekly quizzes so the quizzes also prepare students for the exam. The course provides a high level of teaching assistant support and uses peers who have completed the course in a previous year. The students post their homework assignments on

the web and are thereby able to learn from each other, while the creativity requirement and the sheer joy of creating keeps copying to a minimum.

In the most recent class the three hour open-notes final exam required students to write a webpage, a Scheme servlet, a Scheme applet, and to trace through Scheme code, a logic circuit, and a CMOS circuit. The goal of the exam was to test their ability to synthesize solutions to problems using the tools they had learned.

5.1 Pitfalls

The course requires a substantial investment in TA resources and in class preparation time as there is no textbook for the course. Indeed the course has been heavily revised each year to include more web programming. We are currently working on a textbook which should lessen the class preparation time.

The fact that the course is taught as a large lecture course makes it difficult to keep track of the students who are doing poorly. This is partly ameliorated by weekly quizzes which help track student performance. Smaller class sizes or sectionals might make it easier to track students, but would require a greater commitment of staffing resources.

The current version of software tools used in the course (debuggers, help systems, etc.) are not as well-suited for novice programmers as are other more mature systems (e.g. DrScheme), but they are available as applets so there is a tradeoff between ease of access and ease of use. We are strongly considering porting the class to DrScheme and/or other Scheme systems.

Although the course covers a great deal of material and requires the students to demonstrate their mastery of it in timed quizzes and exams as well as substantial homework projects, the grades are always highly skewed toward the top. This suggests that the class should be taught in two or more sections as the very best students are clearly not being sufficiently challenged. For these students a modified version of the course which included more "algorithmic" computer science would be ideal. This would, again, require a greater commitment of department resources to the non-major course offerings.

6 Lessons learned

Overall the most surprising aspect of the course is that these non-science students have been able to learn how to write servlets, applets, and applications in Scheme, all within a 6 week unit of a 13 week semester. Although they have not delved deeply into "algorithmic" computer science, most of the students do thoroughly understand the mechanism by which a computer program can specify the appearance and functionality of simple applets and servlets. They also understand the notion of a formal semantics (the substitution model) for a computer language and the idea of the evolution of a process as a model of computation as in SICP [1].

The primary reasons for the success of this approach seems to be two-fold:

- **Scheme reduces cognitive overload.** By using a subset of Scheme we eliminate the problem of learning complicated syntax (as one must only match parens (of various sorts) and quotes and the Jscheme IDEs help one do this) and also minimize the problem of learning the underlying abstract machine due to the declarative nature of the language. They can understand the Scheme programs they write using a combination of the substitution model with an intuitive notion of objects (window, buttons, label, menus), events (button pushes, choice selections), and simple operations on these objects (reading/writing data from GUI components or HTML fields). If we were to use Java for this class they would be exposed to a much more complicated model with different kinds of methods (static/instance/constructor), variables (static/instance fields, local variables, parameters), types (classes, interfaces, scalars), and a dizzying array of packages. The use of Jscheme reduces all of the Java libraries to a set of primitive procedures and greatly reduces cognitive overload.
- **JScheme makes applets and servlets easily accessible to non-majors.** By using a Scheme implemented in Java we are able to maintain strong student interest by embedding Scheme in applets, servlets, and JNLP applications and thereby allowing the students to develop web artifacts that are usually only accessible to upper level Computer Science majors. Most of these types of applications could be made accessible through other Scheme implementations. Applets would require a plug-in, but students would probably be just as excited (if not more excited) about creating double-clickable GUI applications in Scheme, which would not require a plug-in.

Acknowledgment

I would like to acknowledge the support of the steadily growing Jscheme community, including my co-developers Ken Anderson and Peter Norvig. I would also like to thank the referees of Scheme2002 for their detailed comments as well as the referees from the ICFP02 conference, who provided some excellent suggestions for improving the paper, even though it was not accepted to ICFP02. Finally, I'd like to thank the 1000+ students who have explored the possibilities of Scheme applets

and servlets with me in various introductory classes over the past five years.

References

[1] H. Abelson and J. Sussman. Structure and Interpretation of Computer Programs MIT Press.

[2] Kenneth R. Anderson, Timothy J. Hickey, Peter Norvig "Silk: A Playful Combination of Scheme and Java" Proceedings of the Workshop on Scheme and Function Programming Rice University, CS Dept. Technical Report 00-368, September 2000.

[3] Ken Anderson and Timothy J. Hickey, "Reflecting Java into Scheme" Proceedings of Reflection 99, Springer-Verlag, Lecture Notes in Computer Science, v. 1616, 1999.

[4] William Clinger and Jonathan Rees, editors. "The revised^4 report on the algorithmic language Scheme." In ACM Lisp Pointers 4(3), pp. 1-55, 1991

[5] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, Matthias Felleisen. *DrScheme: a programming environment for Scheme.* Journal of Functional Programming 12(2): 159-182 (2002)

[6] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. *DrScheme: a pedagogic programming environment for Scheme.* Proc. 1997 Symposium on Programming Languages: Implementations, Logics, and Programs, 1997.

[7] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi. *How to Design Programs.* MIT Press, 2001.

[8] Timothy J. Hickey, Peter Norvig, and Ken Anderson "LISP - a Language for Internet Scripting and Programming", (.ps.gz 130K) in LUGM'98: Proceedings of Lisp in the Mainstream, Nov. 1998, Berkeley, CA.

[9] Timothy J. Hickey, Richard Alterman, John Langton. "TA Groupware" Tech. Rep. CS-02-222, CS Dept. Brandeis University, 2002.

[10] Kurt Normark, "Programming World Wide Web pages in Scheme" Sigplan Notices, vol. 34, no. 12, 1999.

[11] Eric Roberts. *An overview of MiniJava.* in SIGCSE'00 ACM Digital Library, 2000.

[12] David Reed. *Rethinking CS0 with Javascript.* in SIGCSE'00 ACM Digital Library, 2000.

SchemeUnit and SchemeQL: Two Little Languages

Noel Welsh
LShift
Burbage House
83-85 Curtain Road
London, EC2A 3BS, UK
noel@lshift.net

Francisco Solsona
Universidad Nacional
Autónoma de México
Mexico City, Mexico 04510
solsona@acm.org

Ian Glover
Cambridge Positioning
Systems
62-64 Hills Road
Cambridge, UK
ian@manicai.net

ABSTRACT

We present two little languages implemented in Scheme: SchemeUnit, a language for writing unit tests, and SchemeQL, a language for manipulating relational databases. We discuss their design and implementation and show how the features of functional languages in general, and Scheme in particular, contribute to the ease of use and implementation of our languages.

Keywords

Scheme, Little Language, SQL, Unit testing, SchemeQL

1. INTRODUCTION

The domain specific language, or little language, is a powerful technique for increasing programmer productivity. Much work in domain specific languages has been done in functional languages (e.g. [28, 13, 8]). Our work is no different in this regard. Our contribution is to focus on the interface of our languages and show how we can use the features of functional languages in general, and Scheme in particular, to improve the user experience. We describe little languages for unit testing and relational database manipulation. The two languages have been used by the authors and others in real applications, and the code is available from

http://schematics.sourceforge.net/

2. THE SCHEMEUNIT FRAMEWORK

Unit testing concerns testing individual elements of a program in isolation. SchemeUnit is a framework for defining, organizing, and executing unit tests written in the PLT dialect of Scheme[11]. We drawn inspiration from two strands of work: existing practice in interactive environments and the development of unit testing frameworks following the growth of Extreme Programming.

In an interactive environment it is natural to write in a "code a little, test a little" cycle: evaluating definitions and then immediately testing them in the read-eval-print loop (REPL). We take the simplicity and immediacy of this cycle as our model. By codifying these practices we preserve the test cases beyond the running time of the interpreter allowing the tests to be run again when code changes.

Third Workshop on Scheme and Functional Programming. October 3, 2002, Pittsburgh, Pennsylvania, USA.

Unit testing is one of the core practices of the Extreme Programming[3] software development methodology. Unit testing is not new to Extreme Programming but Extreme Programming's emphasis on unit testing has spurred the development of software frameworks for unit tests. The original unit testing framework (SUnit) is written in SmallTalk[2]. Since then unit testing frameworks have been written for many languages[18]. We draw inspiration from these frameworks and find it enlightening to compare the expressivity of these frameworks with SchemeUnit. In particular we will compare SchemeUnit to JUnit[4], an extremely popular unit testing framework for the Java language (it has been downloaded over 340,000 times at the time of writing).

We start our discussion by clarifying the goals of SchemeUnit. We then describe the framework's design and show how our goals have influenced the design. We follow with a comparison of SchemeUnit and JUnit that illustrates how the expressivity of Scheme leads to a cleaner implementation and better user experience. We finish with a discussion of related and future work.

2.1 Goals

We have three goals for SchemeUnit. Firstly we want to remain as close as possible to the "code a little, test a little" cycle we described above. Secondly we want to support the main testing patterns we encounter in practice. Finally we want to support user extensions to the testing framework.

Throughout this paper we shall use an example of simple interactive testing to illustrate our design. Suppose the user is testing the invariance of write and read. The code they may execute is given below:

```
(define data (list 1 2 3 4))

(with-output-to-file "test.dat"
  (lambda () (write data)))

(with-input-from-file "test.dat"
  (lambda () (equal? data (read))))

(delete-file "test.dat")
```

The programmer checks the test by inspecting the result of the (equal? data (read)) expression. If the result is *#t* the test has succeeded.

We shall show how this example is coded in our framework and take the simplicity of the above example as our goal.

2.2 Core Design

The *test* is the core type in our framework. A test is either a *test case*, which is a single action to test, or a *test suite*, which is a collection of tests.

test	$\longrightarrow$	test-case | test-suite
test-case	$\longrightarrow$	name $\times$ action
test-suite	$\longrightarrow$	name $\times$ tests
tests	$\longrightarrow$	listof test

The hierarchical arrangements of tests into suites helps the programmer organize and maintain their tests.

We represent a test action as a closure. Three ways spring to mind to signal test success or failure:

1. Indicate success by returning a non-*#f* value and failure by returning *#f*.
2. Return a datatype indicating success or failure and additional information
3. Throw an exception on failure and return normally for success

The first method has the advantage of simplicity but the disadvantage that we lose information about the cause of failure, so we discard it immediately. The other two methods are equivalent in terms of the information they can return (we can encode arbitrary information in the return value or the exception). We have several reasons for choosing the third option over the second. Firstly we wish to catch exceptions anyway to prevent an unexpected error (i.e. ones that we are not testing for) from halting the testing framework. Secondly when using the second method and testing a sequence of expressions it is necessary to use continuation passing style to propagate a test failure that occurs in an intermediate expression. In this case we are simulating exceptions! Therefore for simplicity of implementation and use we choose to throw an exception to signal an error. We also divide the types of exception we catch into those we catch as the result of a tested failure (which we call *failures*) and those we catch due to untested failures (which we call *errors*).

We provide a *run-test-case* function that takes a *test-case* and returns a *test-result*:

(run-test-case test-case)	$\Rightarrow$	test-result

test-result	$\longrightarrow$	test-failure test-case $\times$ failure-exn | test-error test-case $\times$ error-exn | test-success test-case $\times$ result

Finally, the two functions *fold-test* and *fold-test-results* make it easy to walk over tests.

(fold-test test-collector seed test)	$\Rightarrow$	seed
(fold-test-results result-collector seed test)	$\Rightarrow$	seed

seed	$\longrightarrow$	α
test-collector	$\longrightarrow$	(test α) $\longrightarrow \alpha$
result-collector	$\longrightarrow$	(test-result α) $\longrightarrow \alpha$

2.3 Testing Patterns

Our example in the core framework is:

```
(make-test-case
 (assert binary-predicate actual expected)
 "write/read invariance"
 (lambda ()
   (let ((data (list 1 2 3 4)))
     (dynamic-wind
      (lambda ()
        (with-output-to-file "test.dat"
          (lambda () (write data))))
      (lambda ()
        (with-input-from-file "test.dat"
          (lambda ()
            (let ((actual (read)))
              (if (not (equal? actual data))
                  (raise
                   (make-exn:test:assertion
                    (string-append
                     "write/read invariance failed with "
                     (format "actual ~a" actual)
                     " and "
                     (format "expected ~a" data))))
                  #t)))))
      (lambda () (delete-file "test.dat"))))))
```

Clearly we have lost the simplicity of the original REPL! By adding common testing patterns to SchemeUnit we show how we can regain this simplicity.

2.3.1 Assertions

Checking actual output against expected output is the most common test pattern. We borrow the idea of assertion functions from JUnit. An assert function tests a condition, raising a failure exception if the condition is false. The failure exception contains the location of the failed assertion, the actual and expected parameters, and an optional user specified message string.

The core functionality can be provided by a single function:

(assert binary-predicate actual expected [message])

We know from experience that it pays to provide assertions for the most common cases, so SchemeUnit provides a library of assertions:

- (assert binary-predicate actual expected [message])
- (assert-equal? actual expected [message])
- (assert-eqv? actual expected [message])
- (assert-eq? actual expected [message])
- (assert-true actual [message])

- (assert-false actual [message])
- (assert-pred unary-predicate actual [message])
- (assert-exn exn-predicate thunk [message])
- (fail [message])

Assertions are defined using the define-assertion macro:

(define-assertion (name param ...) expr ...)

The define-assertion macro expands into the definition of a macro and a function[1] that takes the given parameters and an optional message string. If the result of the expressions is *#f* the assertion raises a failure exception containing the all the information given above.

The define-assertion macro is exported so users can define their own domain-specific assertions on par with those already provided. We hope over time to accumulate libraries of specialized assertions.

2.3.2 State Management

Note that our example test uses state and hence requires initialization and cleanup code. This is fairly common and we would like to make is easier for the user to specify these actions. Borrowing again from JUnit we call this code *setup* and *teardown* actions and we augment test-case to optionally include them. So

test-case $\longrightarrow$ name $\times$ action [$\times$ setup] [$\times$ teardown]

2.3.3 Interface Enhancements

We use macros to add the repetitive lambda statements around the action, setup, and teardown expressions. We also wrap the call to action with calls to setup and teardown in the macro rather than requiring the test framework to preform this action.

Our example is now:

```
(let ((data (list 1 2 3 4)))
  (make-test-case "write/read invariance"
    (with-input-from-file "test.dat"
      (lambda ()
        (assert-equal? (read) data)))
    (with-output-to-file "test.dat"
      (lambda () (write data)))
    (delete-file "test.dat")))
```

This code is almost identical to the original example typed at the REPL. We have achieved our ease-of-use goal, and we have done so by supporting testing patterns and allowing user extensions to the testing framework.

[1]Only macros can get location information in PLT Scheme. We define the function variant as we have occasionally found uses for higher order assertions. The function variant has a $*$ appended to its name.

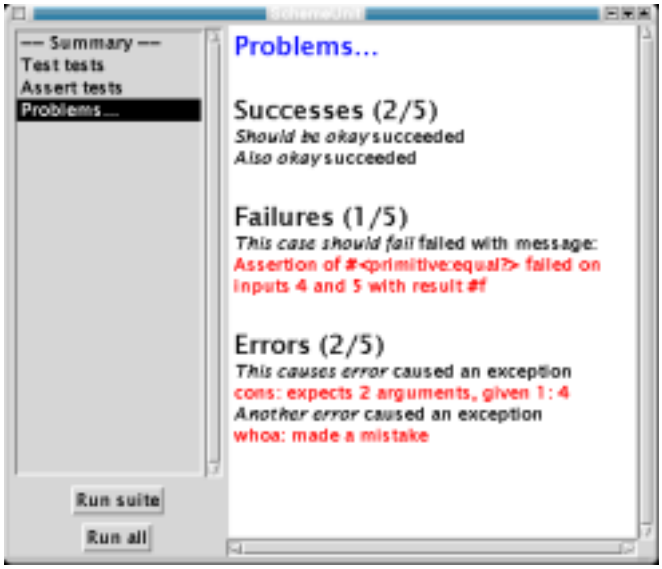

Figure 1: The SchemeUnit graphical interface

2.4 Interfaces

We provide textual and graphical interfaces to SchemeUnit. An example run shows the user interface in action. The following test suite

```
(test/text-ui
  (make-test-suite "Example suite"
    (make-test-case "Will succeed"
      (assert-equal? (+ 1 2) 3))
    (make-test-case "Will fail"
      (assert-equal? (+ 1 1) 3))
    (make-test-case "Will cause error"
      (assert-equal? (/ 1 0) 0))))
```

gives the output:

```
Error:
 Will cause error
 an error of type exn:application:divide-by-zero
 occurred with message: "/: division by zero"
Failure:
 Will fail
 assert-equal? failed at:  top-level 8:7
 Inputs: <2> <3>

1 success(es)  1 error(s)  1 failure(s)
```

The graphical interface is still in development. When complete it will provide source level highlighting and allow navigation to error location using DrScheme. An example of the current graphical interface is shown in Figure 1.

2.5 SchemeUnit versus JUnit

It is instructive to compare SchemeUnit with the popular JUnit test framework, as doing so serves to illustrate the expressive advantage of SchemeUnit. Our discussion centers on a basic example from [25] based on a telephone class. The Java code is:

```
public class TelephoneNumberTests extends TestCase {
  public static void main(String[] args) {
    junit.textui.TestRunner.run(suite());
  }
  public static TestSuite suite() {
    return new TestSuite(TelephoneNumberTests.class);
  }
  public TelephoneNumberTests(String testname) {
    super(testname);
  }
  public void testSimpleStringFormatting()
      throws Exception {
    // Build a complete phone number
    TelephoneNumber number ->
      new TelephoneNumber("612", "630",
                          "1063", "1623");
    assertEquals("Bad string",
                 "(612) 630-1063 x1623",
                 number.formatNumber());
  }
  public void testNullAreaCode()
      throws Exception {
    // Build a phone number without area code
    TelephoneNumber number ->
      new TelephoneNumber(null, "630",
                          "1063", "1623");
    assertEquals("Bad string",
                 "630-1063 x1623",
                 number.formatNumber());
  }}
```

A translation of this to the SchemeUnit syntax is

```
(require (lib "test.ss" "schemeunit")
         (lib "text-ui.ss" "schemeunit"))

(test/text-ui
  (make-test-suite "Telephone number tests"

    (make-test-case "Simple format"
      (assert-equal? "(612) 630-1063 x1623"
        (format-number
          (make-number 612 630 1063 1623))
        "Bad String"))

    (make-test-case "No area code"
      (assert-equal? "630-1063 x1623"
        (format-number
          (make-number (void) 630 1063 1623))
        "Bad string"))))
```

There are several points to note about this example. One is the amount of typing required for this short example. The Java code is far more verbose, most notably in the setup code. This is largely a result of the type declarations and noise keywords (like return and new) required by Java. To our eyes the Scheme code is much more elegant though we recognize this is a subjective judgment.

JUnit relies extensively on reflection. Test cases are defined by prefixing the method name with `test`. This is an elegant solution to the problem that Java has no first class representation of functions but can lead to problems: JUnit uses a custom class loader that can interact unpredictably with other Java code that makes extensive use of reflection (e.g. Java remote method calls). This makes testing difficult in these environments. There is no such problem in Scheme.

In JUnit setup and teardown methods are similarly identified by name and discovered by reflection. Again first class functions reduce the complexity of the SchemeUnit framework.

In general structuring the test suites by value rather than by name makes for a simpler and more flexible system. There are fewer new conventions for the user to remember and tests can be manipulated on the fly.

2.6 Related and Future Work

SUnit has spawned a large and increasing number of testing frameworks of which SchemeUnit is one. We shall briefly consider those that are particularly relevant to SchemeUnit.

HUnit[14], an implementation for the Haskell Language, is a recent addition to the family. There are broad similarities between HUnit and SchemeUnit. Both signal failure with exceptions and both provide a number of convenience assertion functions. HUnit recognizes the importance of interface and defines infix operators that make test specification easier. The combination of lazy evaluation and infix operators achieves a similar effect to our macros. We briefly illustrate HUnit below, along with the equivalent code in SchemeUnit:

```
test1 -> 3 ~->? (1 + 2)
tests -> TestList [TestLabel "Addition" test1]
```

```
(define tests
  (make-test-suite
    "All tests"
    (make-test-case "Addition" (assert -> 3 (+ 1 2)))))
```

LIFT[20], CLUnit[1] (Common Lisp) and CurlUnit[5] (Curl) are Lisp dialect implementations of the SUnit framework. All are broadly similar to SchemeUnit. Both LIFT and CLUnit have some stateful features to ease interactive development of tests. Defining a test in LIFT (with deftest) implicitly creates a test suite to which later tests (created with addtest) are automatically added. In CLUnit tests are categorized by name and stored in a global collection. Tests override existing tests with the same name and are removed with the remove-test function. CurlUnit is a direct translation of JUnit to Curl so most of our earlier comments about JUnit apply to CurlUnit.

The FORT[9] framework, implemented in O'Caml, takes a different approach to the SUnit family. Test results take one of seven values including unexpected success, expected failure, untested, and unresolved in addition to the more usual pass and fail. Test results and returned by the normal function return mechanism so we envisage some difficulty in constructing a single test case containing multiple test expressions. The multitude of test results is an interesting idea but we have yet to encounter a situation where they are necessary. Lacking a clear need we favor simplicity and stick with our three result types.

As the Extreme Programming community evolved from the design pattern community it is no surprise that testing patterns[23][10] have been developed. We intend to analyze these patterns and see how SchemeUnit can provide direct support for them.

A more advanced approach is to generate tests from specifications (e.g. [6]). This approach naturally leads to model checkers like ACL2[19] and SPIN[15] that prove correctness. This is a powerful approach, though quite a leap from our simple system.

SchemeUnit only targets unit tests. In future we wish to target functional (whole system) testing, and testing of non-functional requirements such as performance. We are also aim to extend SchemeUnit to support domain specific functionality such as web site testing.

3. THE SCHEMEQL QUERY LANGUAGE

The International Standard Database Language[17] (SQL 1992, SQL'92 or just SQL) is a declarative language for manipulating data in database manager systems (DBMS). SQL is the standard interface to relational databases and is implemented by all major (and most minor) DBMSs. SchemeQL integrates a database manipulation language into the Scheme language offering an alternative to raw SQL.

Nowadays most database programmers already know SQL, and SchemeQL is designed to offer a gentle slope[16] from existing SQL knowledge to the higher level abstractions offered by SchemeQL.

We start by discussing the limitation of embedded SQL and why an alternative is desirable. We then describe the design and implementation of SchemeQL. We follow with an extended example that shows how SchemeQL builds on SQL but provides extended functionality that makes programming in SchemeQL easier than SQL. We finish with a discussion of related and future work.

3.1 The Limitations of Embedded SQL

The traditional approach to mixing SQL with another language is to embed the SQL as text strings. Even supposedly modern languages like Java [12] continue this tradition. The disadvantages of this approach are:

- SQL statements are not checked until execution time. It is easy to make grammatical or type errors when embedding SQL. For example, forgetting to include a space when concatenating two strings is a common error. Similarly one can write a SQL statement that uses SQL constructs where they aren't allowed, or uses the wrong type for arguments to SQL functions and so on. All these errors will cause execution time exceptions that may affect end users, whereas compilation time exceptions would have been caught and dealt with by the programmer.
- SQL statements can not be manipulated like host language statements. Except by using crude text processing one cannot programaticaly compose, abstract, and refine SQL statements. Hence code quality and programmer productivity suffer when using embedded SQL

If SQL statements were first class members of the programming language we could use our existing tools and language constructs to work with them, avoiding the problems given above.

3.2 The SchemeQL Design: a better SQL

SchemeQL embeds in Scheme a little language for creating and manipulating SQL queries. SchemeQL allows complex structured statements to be treated as first class citizens, thus considerably raising the level of abstraction a programmer can use.

The SchemeQL grammar is very *schemish* while following closely, in spirit, the SQL grammar. This eases the implementation as SQL is a complex mix between the relational algebra and the relational calculus, but more importantly allows the programmer to use their existing knowledge of basic SQL constructions and programming in Scheme. Furthermore, by making SchemeQL a set of syntactic extensions and procedures we can concentrate on the design of our little language, while retaining the whole power of a real programming language, Scheme, following the steps of other little languages [28], and [8].

SQL statements are divided into three main groups:

- Selection (SELECT)
- Modification (INSERT, UPDATE, and DELETE)
- Data definition (CREATE TABLE)

Selection (aka projection) statements produce a result set. Modification statements return a natural number representing the number of rows affected by the execution of the statement. Data definition statements are only interesting for their side effects, such as creating a new table or view in the database.

SchemeQL has the same logical division, with the following differences: result sets are represented by *cursors*, a lazy stream of rows (which basically allows the programmer to work with one row at the time), and instead of having a one to one mapping from SQL statements to Scheme procedures, we have a set of procedures to mimic the work of a single SQL statement. This simplifies the construction, combination, and refinement of statements. For instance, the full power of the SQL SELECT statement is achieved by the appropriate combination of several SchemeQL forms. Basic selection in SchemeQL follows this grammar:

```
selection    ::= (query <exp>)
               |  (query ((LITERAL <exp>)))
               |  (query <col-spec> <table-spec>)
               |  (query <col-spec> <table-spec>
                         <pred-spec>)
<exp>        ::= string-or-symbol
```

```
                    (passed verbatim to the DBMS)
<col-spec>    ::= ALL | (<column> ...)
<column>      ::= string-or-symbol | Number
                | (<table> string-or-symbol)
                | (AS <column> string-or-symbol)
                | (LITERAL <exp>)
<table-spec>  ::= <table>
                | (<action> <table-spec> <table-spec>)
<table>       ::= string-or-symbol
<action>      ::= ALIAS | INNERJOIN | STRAIGHTJOIN
                | NATURALLEFTJOIN
<pred-spec>   ::= (<op> <col-spec> <col-spec-or-value>)
                | ([AND|OR|NOT] <pred-spec> ...)
<op>          ::= < | <= | > | >= | = | <>
                | Any DBMS defined binary operator
<col-spec-or-value> ::= <col-spec>
                | Any value suitable for comparison
```

It is important to note, that the subforms in query, and in most forms in SchemeQL for that matter, are implicitly backquoted. Thus, (query ALL ,(f x)) means "select everything from the table, or tables returned by the application of Scheme procedure *f*, to the Scheme variable *x*".

3.2.1 More on Selection, and the SchemeQL Times

The query procedure alone does not provide all the functionality a programmer may want when selecting data from a database, and for a good reason: it would be as complex as the SQL's SELECT statement. Instead of offering a much too complex form, SchemeQL provides a set of forms, and procedures to specialized, compose, and otherwise handle selections. These forms are: query, distinct!, group-by!, order-by!, having!, limit!, union, intersect, and difference.

```
<selection> ::= (distinct! <selection>)
              | (group-by! <selection> <limit-col>)
              | ... the other forms
<limit-col> ::= ([ASC|DESC] <col-spec>) | <col-spec>
```

The syntax of the rest of the forms is just minor variations of that given above.

The reader may wonder what a SchemeQL selection exactly does. A selection in SchemeQL is an internal Scheme structure, that holds the information provided thus far to *perform* the selection, and that is why you can continue specializing it.

(query param ...) ⇒ query-struct

This is what we called the *SchemeQL compilation time*, for it allows us to perform basic static checking, based only on the information already provided to perform the selection. Only when schemeql-execute is called is the selection is performed and a result set (also called a *cursor* in SchemeQL) is returned.

(schemeql-execute schemeql-struct [conn]) ⇒ cursor

This is the *SchemeQL execution time*. The same scenario repeats itself for the data modification and data definition forms in SchemeQL.

Since sometimes we want to immediately execute a form, SchemeQL provides some useful shorthands for some forms that combine the generation of the internal structures and their execution. Here are some such forms that we will use later:

(direct-query conn param ...) ⇒ cursor
(query-with-current-connection param ...) ⇒ cursor
(query/cc param ...) ⇒ cursor

where conn is an open connection to a DBMS, which is created by a call to the SchemeQL form connect-to-database, and param ... are exactly those parameters valid for query.

3.2.2 SchemeQL Cursors

SQL result sets can be seen as tuples that form a table. SchemeQL cursors are pairs of values, (row promise), where row is a list representing the first tuple in the result set, and promise is a cursor holding a promise (that has to be forced) to return the rest of the tuples in the result set.

cursor → row × promise
row → listof any

A library to work with cursors is provided as part of SchemeQL. Programmers most likely will use the following basic procedures to work with cursors:

- (cursor-car cursor): returns the first tuple in cursor.
- (cursor-cdr cursor): returns the rest of cursor, another cursor, similar to the original only that the *next* element, if any, is on the cursor-car position of the returned cursor.
- (cursor-null? cursor): $\#t$ iff cursor is the empty cursor.
- (cursor-map proc cursor): returns another cursor, whose first element is the application of proc to the first element in cursor, and whose second element holds the promise to apply proc to the rest of cursor.
- (cursor->list cursor N): returns a list containing the first N, or less if there are not enough, rows in cursor.
- (finite-cursor->list cursor): returns a list containing all the elements of cursor.

It is worth noting that cursors in SQL are a completely different concept, and are used to retrieve a small number of rows at a time out of a larger query. SchemeQL also provides support for them, through the procedures open-cursor, which receives a query and optional information to create different *kinds*[2] of cursors, the initial size of the set, and the starting row. Two other procedures work on the result of open-cursor: roll-cursor!, that changes the orientation of the given cursor, and close-cursor!, which closes the given cursor.

One important feature of this way of handling SQL cursors is that the resulting set of tuples is represented as an

[2]Kinds as those defined by Open Database Connectivity (ODBC) [27], which are: `FORWARD ONLY`, `STATIC`, `KEYSET DRIVEN`, and `DYNAMIC`.

SchemeQL cursor, and thus can be handled in the same way as the result of regular queries. We will not go into more detail here for space reasons.

3.2.3 The Rest of SQL

Most of the "usual" SQL functionality is already part of SchemeQL. Transactions, for instance, can be handled in two different ways. The first one, is by using the (transaction exp ...) form which executes all the expressions given in order, and if no exception occurs then it *commits* the block, otherwise, it sends a *rollback* to the DBMS, and pass along the exception. The transaction form tries to set the transaction isolation level to the highest possible, ideally to *serializable* level.

The second way allows the programmer to select the isolation level required and is represented by two procedures: begin-transaction and end-transaction. The begin-transaction form switches to manual commit mode, and sets the isolation level to the highest supported by the DBMS, or to the requested one if given. Then end-transaction either commits or rolls back the transaction block, depending on the argument supplied by the programmer. The transaction form is more scheme-like, since the other two can lead to the common error of opening a transaction, executing a block of expressions, and never closing the transaction again.

SchemeQL supports basic user and table management, and connection management that allows simultaneous connections to different databases. Even non-standard, yet very useful and regularly employed, SQL extensions such as CREATE DATABASE and USE DATABASE are supported, though no SQL standard procedure depends internally on these extensions.

3.3 The SchemeQL Implementation

SchemeQL is layered upon SrPersist[3] and takes full advantage of SrPersist's knowledge of the particular DBMS in use. SrPersist provides a safety check for every SQL statement sent to the DBMS, in addition to the SchemeQL's error detection, and thus we can offer a hierarchical approach to error handling.

SchemeQL together with SrPersist is a highly portable library since ODBC is the *de facto* standard for database connectivity and is widely supported (although it should be noted that many ODBC drivers have different levels of conformance[4]). In this regard SchemeQL offers two specific and crucial benefits. Firstly it hides the tedious and ugly details of the ODBC conformance levels from the Scheme programmer. Secondly, and more importantly, it removes the complexity of standard ODBC manipulation, which is probably the biggest drawback of ODBC when compared to other DBMS drivers.

[3]SrPersist is an ODBC library for PLT Scheme. More information on SrPersist can be found at:

http://www.plt-scheme.org/software/srpersist/

[4]At the time of writing there have been several major releases, from 1.0 through 3.51, and SrPersist supports them all.

Even though, for portability reasons, we use SrPersist, SchemeQL allows the use of different DBMS drivers. ODBC drivers are known to do extensive error checking, and so it is possible to have a database specific driver outperforming a generic ODBC driver. SQL support and basic error checking facilities are independent of the driver in use.

3.4 SchemeQL in action

All examples below are based around the following database structure. Suppose you own a software company, and the following tables are a snippet of your employees database.

personnel

id	name	lid
1	Noel	1
2	Ian	1
3	Francisco	1
4	Simon	2
5	James	3
6	Brian	4
7	Dennis	4

salaries

id	salary
1	30'000
2	30'000
3	30'000
4	30'000
5	45'000
6	45'000
7	45'000

languages

id	lang
1	Scheme
2	Haskell
3	Java
4	C

We start with the most common sort of query, which is a SELECT statement such as the following statement to get the names of all the programmers:

```
SELECT name FROM personnel
```

In SchemeQL this query has almost exactly the same structure as its SQL equivalent:

```
(query (name) personnel)
```

Now suppose we wish to get all the *id*s of those employees who program in Scheme. In SQL we'd write:

```
SELECT personnel.id
  FROM  personnel, languages
  WHERE personnel.lid = languages.id
  AND   languages.lang = 'Scheme'
```

In SchemeQL we write

```
(query ((personnel id))
       (personnel languages)
       ((= (personnel lid) (languages id))
        (= (languages lang) "'Scheme'")))
```

Again the two queries have a very similar structure. Now suppose we want to get all Java programmers. Immediately we see an opportunity for code reuse if we parameterize the above queries on the language. This is trivial in SchemeQL as we can use abstraction facilities provided by Scheme:

```
(define (programmers language)
  (query ((personnel id))
         (personnel languages)
         ((= (personnel lid) (languages id))
          (= (languages lang) ,language))))
```

Remember that most subforms in SchemeQL are backquoted.

There is no way to do this in standard SQL, though individual DBMSs may provide parameterized queries. To do this in embedded SQL we could append strings:

```
(define (programmers language)
  (string-append
    "SELECT id "
    "FROM personnel, languages"
    "WHERE personnel.lid = languages.lid "
    "AND   languages.lang = " language))
```

We note that this method is error-prone as it is easy, for example, to forget to include a space between strings as we have done above (between *languages*, and the keyword *WHERE*).

Now suppose you want to get the *id*s of all C programmers who are earning 45'000. This is the intersection of all C programmers, which we already know how to do, with all programmers who are earning 45'000. In SQL we can write:

```
SELECT id
  FROM personnel, languages
  WHERE personnel.lid = languages.lid
  AND   languages.lang = 'C'
  INTERSECT ( SELECT id
              FROM salaries
              WHERE salary = '45000' )
```

In SchemeQL we can form the two sets separately and then perform the intersection:

```
(let ((c-programmers (programmers "'C'"))
      (high-earners (query  (id) (salaries) (= salary "45000"))))
  (intersect c-programmers high-earners))
```

Notice how we have reused the programmers function defined above and then composed a query from parts. We cannot do this in SQL.

That does it! Impressed by the productivity of your functional programmers you decide to fire all the Java and C programmers and use the extra money to give a raise to your fine Scheme programmers (you find the Haskell programmers productive but inexplicably lazy). Coincidentally this also give us an opportunity to show further query composition and cursor handling in SchemeQL.

First we define the sets of interest: the Schemers, who are getting a raise, the Haskell programmers, who just stay as they are, and everyone else, who are getting the opportunity to explore other interests.

```
(define schemers (programmers "'Scheme'"))
(define haskellers (programmers "'Haskell'"))
```

```
(define fired
  (let ((all (query (id) personnel)))
    (schemeql-execute
      (difference all (union schemers haskellers)))))
```

Now all programmer who have been fired are removed from the salaries table:

```
(cursor-map
 (lambda (programmer)
   (let ((id (car programmer)))
     (delete/cc salaries (= id ,id))))
 (result-cursor fired))
```

Finally, to give the Scheme programmers a raise:

```
(cursor-map
 (lambda (id)
   (update/cc salaries
              ((salary (LITERAL "salary * 2")))
              (= id ,(car id))))
 (result-cursor (schemeql-execute schemers)))
```

The above operations cannot be performed in pure SQL as query results cannot be used as the input to modification statements. We give below equivalent statements to perform the above actions. Where an action requires repetition of a number of very similar statements (eg, when DELETEing the imperative programmers) we only give an example.

```
SELECT personnel.id
    FROM personnel
    EXCEPT (SELECT personnel.id
              FROM personnel,languages
              WHERE personnel.lid = languages.lid
                    AND languages.lang = 'Scheme'
            UNION
            SELECT personnel.id
             FROM personnel,languages
             WHERE personnel.lid = languages.lid
                   AND languages.lang = 'Haskell');
DELETE FROM salaries
    WHERE id = 4;
SELECT personnel.id
    FROM personnel
    INTERSECT (SELECT personnel.id
              FROM personnel,languages
              WHERE personnel.lid = languages.lid
                    AND languages.lang = 'Scheme');
UPDATE salaries
    SET salary = salary * 2
    WHERE id = 1;
```

3.5 Related and Future Work

Haskell/DB, a compiler embedded in Haskell that dynamically generates SQL queries, was developed as an instance of the more general design pattern for embedding client-server style services into Haskell detailed in[22]. Some of the benefits this technique offers are:

- Programmers need to know only one language,

- it allows language extensions in the form of libraries to be presented,
- it is possible to impose specific typing rules,
- integration with other domain specific libraries (e.g. CGI, mail) is possible, and finally
- this approach offers a strategic advantage, for it empowers programmers to use the language infrastructure, such as the module, and type systems.

SchemeQL has all of these benefits except for static typing.

The implementation of Haskell/DB, presented in[22] uses ActiveX Data Objects (ADO) to connect to the DBMS. In this regard Haskell/DB is limited to the Windows platform. SchemeQL does not share this limitation as it uses SrPersist, which can interact with any ODBC driver.

Our approach is to define a limited domain specific language that can be translated into SQL. Another approach is to expand the database query language into a full programming language[29]. This approach has many benefits but requires the underlying DBMS to change.

It is clear that *structured data* is taking over. The Extensible Markup Language [26] (XML) is now considered the universal format for structured documents and data on the Web. With XML arises the need for efficient query languages to exploit structured data. XML Query [24] is a working group aiming to create a set of query facilities to extract data from XML, or viewing XML files as databases. Unfortunately there is not yet any direct point of comparison between XML and current database technology. This will remain one of the most interesting topics of research in the years to come. Whether or not a language like SchemeQL will be able to enter the XML realm is a question we cannot answer yet.

In the immediate future we will be adding support for specific DBMS drivers and SQL dialects (e.g. Oracle, PostgreSQL, etc.). We will also attempt to standardize the SchemeQL syntax as a Scheme Request for Implementation[21] (SRFI).

4. SCHEMEUNIT AND SCHEMEQL

SchemeUnit and SchemeQL have both been designed with a 'gentle-slope' philosophy: start with an already familiar base and then build additional functionality as independent components on that base. In SchemeUnit this is evident in the way test code mimics the "code a little, test a little" cycle and adds facilities to organize and rerun tests. In SchemeQL the starting point is the SQL SELECT statement upon which the query macro is modeled. The combinators intersect, difference and so on are then introduced as ways of modifying the basic query.

SchemeUnit and SchemeQL both take advantage of Scheme's macro facilities to present a cleaner interface to the user. In both languages macros are used to avoid repetitious lambda statements. In SchemeUnit this is in the creation of test cases. In SchemeQL this is in cursor creation. Macros are also used for other purposes: in SchemeUnit to allow user-extensions via the define-assertion macro and in SchemeQL to provide implicit backquoting on forms. These simple uses of macros go a long way to improving the user experience.

SchemeUnit is used extensively to test itself and SchemeQL.

5. CONCLUSIONS

A language is a user interface just like a graphical interface and deserves as much attention from the language designer as a GUI would get from it's designer.

We have described SchemeUnit, a little language for writing tests in Scheme, and have illustrated how we have used the features of functional languages in general, and Scheme in particular, to simplify the interface. Via comparison with the "code a little, test a little" cycle and the JUnit framework we have shown that SchemeUnit achieves an admirable level of simplicity without sacrificing expressive power.

SchemeQL, our little language for database interaction, has been shown to be a feasible alternative to embedded SQL. By building on the programmer's knowledge of SQL and extending it with modular combinators we achieve tighter integration with the Scheme language, a better, more modular, parameterization of SQL statements and improved expressibility and abstraction.

PLT Scheme, the host language for both our little languages gives us a certain number of extra, and free advantages that makes them usable, through its DrScheme programming environment[11]: a syntax-sensitive editor, a syntax checker, an stepper, and interaction with other libraries, and plugins.

- Since our little languages consists entirely of tree-structure expressions, the editor's features are inherited. Users only needs to add the keywords in our little languages to DrScheme (to have them indented appropriately.)
- No modification is needed to work with the syntax checker, and the stepper since these two work transparently over procedures, and macros.
- Since all of the host language is available to users, a program can load, or enable a certain number of libraries, plugins, or other embedded little languages as needed with no extra fuss.

The only extra advantage we are not exploiting is the validity checking available through the MrFlow component of DrScheme though it should not be hard to expand the constructions of our little languages to type definitions, as in [8].

Finally we note that our language evaluation has been qualitative; based on our experiences using the languages in question. We are aware of some work in quantitative evaluation[7] and this research will contribute to a better understanding of what makes good language design.

6. ACKNOWLEDGMENTS

We are indebted to the following individuals:

Ryan Culpepper, who created the graphical interface to SchemeUnit and has contributed greatly to its design.

Paul Steckler, Shriram Krishnamurthi, Matt Jadud, MJ Ray and the anonymous reviewers who offered comments on the draft versions of this paper.

7. REFERENCES

[1] F. A. Adrian. Clunit. http://www.ancar.org/CLUnit/docs/CLUnit.html, 2002.

[2] K. Beck. *Kent Beck's Guide to Better Smalltalk*, chapter 21. SIGS Reference Library. Cambridge University Press, 1999. http://www.xprogramming.com/testfram.htm.

[3] K. Beck. *Extreme Programming Explained*. Addison-Wesley, 2000.

[4] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.

[5] J. Beekmann. Curlunit. http://curlunit.sourceforge.net/, 2002.

[6] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The jml and junit way. Technical Report 01-12, Department of Computer Science, Iowa State University, November 2001.

[7] S. Clarke. Evaluating a new programming language. In G. Kadoda, editor, *Proceeding of the 13th Workshop of the Psychology of Programming Interest Group*, volume 13, April 2001.

[8] J. Clements, P. Graunke, S. Krishnamurthi, and M. Felleisen. Little languages and their programming environments. In *Proceedings of the Monterey Workshop*, 2001.

[9] P. Doane. Fort: Framework for o'caml regression testing. http://www.sourceforge.net/projects/fort, 2002.

[10] M. Feathers. The 'self'-shunt unit testing pattern. http://www.objectmentor.com/-resources/articles/SelfShunPtrn.pdf, 2001.

[11] R. Findler, J. Clements, C. Flanagan, S. Krishnamurthi, P. Steckler, and M. Felleisen. Drscheme: A programming environment for scheme. *Journal of Functional Programming*, 2001.

[12] M. Fisher, R. Cattell, G. Hamilton, S. White, and M. Hapner. *JDBC API, Tutorial, and Reference, Second Edition: Universal Data Access for the Java 2 Platform*. The Java Series. Addison-Wesley Longman, 1999.

[13] P. Graham. *On Lisp*. Prentice Hall, 1993.

[14] D. Herington. Hunit. http://hunit.sourceforge.net/, 2002.

[15] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[16] M. Hostetter, D. Kranz, C. Seed, C. Terman, and S. Ward. Curl: A gentle slope language for the web. *World Wide Web Journal*, II(2), 1997. http://www.w3j.com/6/.

[17] Database language sql. International Organisation for Standardization (ISO), 1992.

[18] R. Jefferies. Software downloads. http://www.xprogramming.com/software.html.

[19] M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.

[20] G. King. Lift - the lisp framework for testing. Technical report, University of Massachusetts, 2001.

[21] S. Krishnamurthi, D. Mason, and M. Sperber. Scheme request for implementation. http://srfi.schemers.org/, 1998.

[22] D. Leijen and E. Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain-Specific Languages (DSL)*. USENIX, October 1999.

[23] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: Unit testing with mock objects. In G. Succi and M. Marchesi, editors, *Extreme Programming Examined*. Addison-Wesley, 2001.

[24] M. Marchiori. Xml query. http://www.w3.org/XML/Query, 2000.

[25] M. T. Nygard and T. Karsjens. Test infect your enterprise javabeans. *JavaWorld*, May 2000.

[26] L. Quin. Extensible markup language (xml). http://www.w3.org/XML/, 1997.

[27] R. E. Sanders. *ODBC 3.5 developer's guide*. McGraw-Hill, 1998.

[28] O. Shivers. A universal scripting framework, or lambda: The ultimate "little language". In J. Jaffar and R. H. C. Yap, editors, *Concurrency and Parallelism: Programming, Networking, and Security*, volume 1179 of *Lecture Notes in Computer Science*, pages 254–265. Springer, 1996.

[29] V. Tennen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proceedings of 3rd International Workshop on Database Programming Languages*, 1991. http://db.cis.upenn.edu/Publications/.

This is Scribe!

Manuel Serrano
Inria Sophia-Antipolis
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex
France
Manuel.Serrano@sophia.inria.fr
http://www.inria.fr/mimosa/Manuel.Serrano

Erick Gallesio
Université de Nice - Sophia Antipolis
930 route des Colles, BP 145
F-06903 Sophia Antipolis, Cedex
France
Erick.Gallesio@unice.fr
http://saxo.essi.fr/~gallesio

ABSTRACT

This paper presents SCRIBE, a functional programming language for authoring documents. Even if it is a general purpose tool, it best suits the writing of technical documents such as web pages or technical reports, API documentations, etc. Executing SCRIBE programs can produce documents of various formats such as PostScript, PDF, HTML, Texinfo or Unix man pages. That is, the very same program can be used to produce documents in different formats. SCRIBE is a full featured programming language but it *looks* like a markup language *à la* HTML.

1. INTRODUCTION

SCRIBE is a functional programming language designed for authoring documentations, such as web pages or technical reports. It is built on top of the Scheme programming language [5]. Its concrete syntax is simple and it sounds familiar to anyone used to markup languages. Authoring a document with SCRIBE is as simple as with HTML or LaTeX. It is even possible to use it without noticing that it is a programming language because of the conciseness of its original syntax: the ratio *markup/text* is smaller than with the other markup systems we have tested.

Executing a SCRIBE program with a SCRIBE evaluator produces a target document. It can be HTML files that suit web browsers, LaTeX files for high-quality printed documents, or a set of *info* pages for on-line documentation.

Building purely static texts, that is texts avoiding any kind of computation, is generally not sufficient for elaborated documents. Frequently one needs to automatically produce parts of the text. This ranges from very simple operations such as inserting in a document the date of its last update or the number of its last revision, to operations that work on the document itself. For instance, one may be willing to embed inside a text some statistics about the document, such as the number of words, paragraphs or sections it contains. SCRIBE is highly suitable for these computations. A program is made of *static texts* (that is, *constants* in the programming jargon) and various functions that dynamically compute (when the SCRIBE program runs) new texts. These functions are defined in the Scheme programming language. The SCRIBE syntax enables a sweet harmony between the static and dynamic components of a program.

Authoring documents with a programming language is of course not a novel idea, and a lot of systems have used this approach, such as the TeX [8] typesetting system. PostScript [1] can also be classified in this category. Even if it is not generally directly used for authoring, it represents a document as a program whose execution yields a set of printed pages.

On the other side, solutions based on the SGML [2] or XML [3] formats propose a model where all the computations on a document are expressed outside of the document itself. For instance, the DOM [20] approach extols a strict dichotomy between documents and programs. This dichotomy is presented as a virtue by its proponents, but it is our opinion that it makes simple documents harder to code than with a general linguistic tool because it requires the usage of several different languages with different semantics and different syntax.

With the development of dynamic content web sites, a great number of intermediate solutions based on programming languages have been proposed. These solutions generally consist in giving a way to embed calls to a programming language inside a document. PHP [9] is probably the most representative of this kind. A document is a mix of text and code expressed with different syntaxes. This implies that the author/programmer must deal at the very same time with the underlying text markup system as well as the programming language. Furthermore, these tools do not permit to reify a document structure and are generally limited to the production of web pages only.

The approach we propose is inspired by the LAML system

[12] which uses Scheme as a markup language. In LAML as in SCRIBE, a document is a program and its evaluation yields its final form. Both languages permit the user to typeset documents using an *unique* syntax. However, LAML is limited to the production of HTML, whereas, as said before, the evaluation of a SCRIBE program can produce several output formats.

In Section 2 we present an overview of the SCRIBE system for authoring simple static documents. We show that a SCRIBE program looks like a document specified in a markup language. A more complex usage of the language is shown in Section 3, where some simple text generations are done, as well as some text inclusions built by introspecting the document itself. Section 4 shows various customizations that can take place during the execution of a SCRIBE program. Finally, we compare in Section 5 SCRIBE with various tools or programming languages used for authoring documents.

2. SCRIBE OVERVIEW

This section presents an overview of the SCRIBE programming language and its implementation. First, the syntax is presented in Section 2.1. Then, in Section 2.2, the structure of a program is presented. Finally, Section 2.3 contains some few words about the current state of the SCRIBE implementation.

2.1 Sc-expressions

We have designed the SCRIBE syntax so that it as *unobtrusive* as possible. We have found of premium importance to minimize the weight of meta information when authoring documentations. A complex syntax would prevent it to be used by non computer scientists. A SCRIBE program is a list of expressions (Sc-expression henceforth) that are extended S-expressions [11]. An Sc-expression is:

- An *atom*, such as a string or a number.
- A *list* of Sc-expressions.
- A *text*.

Atomic expressions and *lists* are regular Scheme expressions. A *text* is a sequence of characters enclosed inside square brackets. This is the sole extension to the standard Scheme reader. The bracket syntax is very similar to the standard *quasiquote* Scheme construction. In Scheme, the *quasiquote* syntax allows to enter complex lists by automatically *quoting* the components of the list. It is to be used in conjunction of the *comma* operator that allows to *unquote* the expressions. For instance, the Scheme form:

```
`(compute pi = ,(* 4 (atan 1)))
```

is equivalent to the expression:

```
(list 'compute 'pi '= (* 4 (atan 1)))
```

which evaluates to:

```
(compute pi = 3.1415926535898)
```

The SCRIBE bracket form collects all the characters between the brackets in a list of characters strings. Computations inside brackets are handled by the characters sequence ",(". For instance, the text:

```
[text goodies: ,(bold "bold") and ,(it "italic").]
```

is parsed by the SCRIBE reader as:

```
(list "text goodies: " (bold "bold")
      "and" (it "italic") ".")
```

The SCRIBE syntax is unobtrusive, and easy to typeset with an editor aware of Lisp-like syntax, such as *Emacs*. Documents expressed in SCRIBE are also generally shorter to type-in than their counterpart expressed in classical formatting languages. For instance, the size of the SCRIBE source files of this paper is about 42,200 characters long, whereas it is 53,000 characters in LaTeX and 72,000 in HTML. Even if it is somehow unfair to compare hand-written code against generated ones, these figures give the intuition of the compactness of SCRIBE programs. The idea of extending a standard Scheme reader for text processing comes from the BRL system [10].

2.2 Scribe as a markup language

In this section, we present how to build a document using SCRIBE. As said before, programming skill is not needed to produce a document. In fact, non programmer writers can see SCRIBE as a simple document formatting system such as HTML or nroff [14].

SCRIBE provides an extensive set of pre-defined markups. These roughly correspond to the HTML markups. The goal of this section is to give an idea of the *look and feel* of this system. It will avoid the tedious presentation of an extensive enumeration of all the markups available. For a complete manual of SCRIBE, interested readers can have a look at `http://www-sop.inria.fr/mimosa/fp/Scribe`.

2.2.1 Scribe Markups

A SCRIBE markup is close to an XML element. The attributes that can appear inside an XML element are represented by Scheme keywords. They are identifiers whose first (or last character) is a colon. Scheme keywords have been introduced by DSSSL [4], the tree manipulation language associated to SGML. So, the following XML expression:

```
<elmt1 att1="v1" att2="v2">
   Some text <elmt2>for the example</elmt2>
</elmt1>
```

is represented in SCRIBE as:

```
(elmt1 :att1 v1 :att2 v2
       [Some text ,(elmt2 [for the example])])
```

2.2.2 Document Structure

As said before, a SCRIBE program consists in a list of Sc-expressions. Among these, the `document` one serves a special purpose. It is used to represent the complete document. All the subdivisions of a document must appear as arguments of the `document` call. So, the general structure of a SCRIBE document looks like:

```
<sc-expr>
...
(document :title <sc-expr> :author <sc-expr>
  (abstract <sc-expr>)
  (section :title <sc-expr>
     ...
     (subsection :title <sc-expr>)
     ...
     (subsection :title <sc-expr>)
     ...)
   ...
   (section :title <sc-expr>))
```

As we can see, all the sectioning components of a document are embedded in their containing component (i.e. subsections are embedded in sections, sections are inside chapters, and so on). This strict nesting of document components is particularly useful when one wants to do introspection on the structure of the document, as we will see in Section 3.2.

2.2.3 Scribe standard library

SCRIBE is provided with the usual functions for text processing. Some of these are presented here.

The *Lists* offered in SCRIBE are classical: itemization, enumeration and description. For instance, the following expression:

```
(itemize (item [A first item.])
         (item [A ,(bold "second") one.])
         (item (description
                 (item :key (bold "foo")
                       [is a usual Lisp identifier.])
                 (item :key (bold "bar")
                       [is another one.])))
         (item (enumerate (item "One.")
                          (item "Two."))))
```

produces the following output text:

- *A first item.*
- *A* ***second*** *one.*
- **foo** *is a usual Lisp identifier.*
  bar *is another one.*
- 1. *One.*
  2. *Two.*

Of course, all the usual text ornaments are available in SCRIBE, that is one can easily produce text in **bold**, *italic*, underline or combine ***them***.

The SCRIBE standard library also offers the usual tools for inter and intra document references, footnotes, tables, figures, ... It provides also an original construction, the `prgm` markup, to *pretty-print* codes or algorithms. In contrast with previous systems such as LaTeX there is no need, in SCRIBE to use external pre-processors such as SLaTex [17] and Lisp2TeX [15] for pretty-print programs inside texts. The `prgm` form takes as an option the language in which the code is expressed and its evaluation yields a form that is the pretty-printed version of this code. For instance, the following call

```
(prgm :language c (from-file "ex/C-code.c"))
```

produces the following output

```
int main(int argc, char **argv) {
  /* A variant of a classical C program */
  printf("Hello, Scribe\n");
  return 0;
}
```

if the C program source is located in file `ex/C-code.c`.

2.3 Front-ends and Back-ends

The current version of SCRIBE which is available at `http://www-sop.inria.fr/mimosa/fp/Scribe` contains two front-ends which are used to translate existing document sources into SCRIBE documents:

- `scribeinfo` compiles Texinfo into SCRIBE. An example of such a compilation can be browsed at `http://www.inria.fr/mimosa/fp/Bigloo/doc/r5rs.html`. It is an on-line version of the Scheme definition, automatically produced from a Texinfo source.
- `scribebibtex` translates Bibtex bibliography databases into SCRIBE sources. This tool is, for instance, used to produce the bibliographic references of this paper.

SCRIBE can produce various kinds of document formats. Currently five back-ends are supported:

- **HTML**: It is extensively used on the SCRIBE web page.
- **PS** or **PDF** (via LaTeX): That is, for instance, used to produce the PostScript version of this paper.
- **Man**: which is the format of Unix "man pages".
- **Text**: which is a plain text format.
- **Info**: which is the format of the Emacs documentation.

SCRIBE user programs are independent of the target formats. That is, using one unique program, it is possible to produce an HTML version, and a PostScript version, and an ASCII version, etc. The SCRIBE API is general purpose.

It is not impacted by specific output formats. Independence with respect to the final document format does not limit the expressiveness of SCRIBE programs because specificities of particular formats are handled by dedicated back-ends. Back-ends are free to find convenient ways to implement SCRIBE features. For instance, intra document references are handled differently by the HTML back-end and the TEX back-end. In HTML, they appear as hyper-links whose text is the title of the section. In TEX they appear as section numbers. An output target may even not support some SCRIBE features. In that case, the back-end could possibly omit them (for instance, figures in ASCII formats, or dialog boxes in PostScript documents).

When customization of the produced documents is required, the SCRIBE `hook` form must deployed. It enables to insert characters in the final document. Coupled with conditional evaluation, the `hook` form can be used to implement fine grain tuning aware of the idiosyncrasies of the target format (see Section 3.3).

3. DYNAMIC TEXTS

We show in this section various situations where *dynamic texts*, that is texts not written *as is* in the SCRIBE sources, can be used when authoring documents. We have isolated two kinds of computations. The ones that produce some parts of the document being processed (Section 3.1). The ones that involve *introspection* on the source text (Section 3.2). These computations correspond to two different evaluation stages of the SCRIBE evaluator. The first ones are *front-end* computations that take place at the very beginning of the execution of a program. The second ones are *back-end* computations that take place at the very end of the execution while an internal representation of the whole SCRIBE program has been loaded in memory.

3.1 Computing Sc-expressions

Many typesetting systems such as LATEX enable users to define convenience *macros*. In its simplest form, a *macro* is just a name that is *expanded into*, or *replaced with*, a text that is part of the produced document. Macros are implemented in SCRIBE by the means of functions that produce Sc-expressions. For instance, a macro defining the typesetting of the word "SCRIBE" is used all along this paper. It is defined as follow:

```
(define (Scribe.tex)
   (sc "Scribe"))
```

That can be used in a Sc-expression such as:

```
[This text has been produced by ,(Scribe.tex).]
```

That produces the following output:

"*This text has been produced by* SCRIBE."

The function `Scribe.tex` is overly simple because it merely *inserts* in the SCRIBE program one new string each time it is called. Sometimes we need to compute more complex parts of a document and some texts are better to be computed. Either because they contain pattern repetitions or because they are the result of the evaluation of an algorithm, such as the table of Figure 1.

n=	fact
3	*6*
4	*24*
5	*120*
6	*720*
7	*5040*
8	*40320*
9	*362880*
10	*3628800*
11	*39916800*

Figure 1: Factorial

This table can be *statically* declared in a program using a Sc-expression such as:

```
(table :border 1
       (tr (th "n=") (th "fact"))
       (tr (td :align 'center (bold 3))
           (td :align 'right (it 6)))
       (tr (td :align 'center (bold 4))
           (td :align 'right (it 24)))
       (tr (td :align 'center (bold 5))
           (td :align 'right (it 120)))
       (tr (td :align 'center (bold 6))
           (td :align 'right (it 720)))
       (tr (td :align 'center (bold 7))
           (td :align 'right (it 5040)))
       (tr (td :align 'center (bold 8))
           (td :align 'right (it 40320)))
       (tr (td :align 'center (bold 9))
           (td :align 'right (it 362880)))
       (tr (td :align 'center (bold 10))
           (td :align 'right (it 3628800)))
       (tr (td :align 'center (bold 11))
           (td :align 'right (it 39916800))))
```

Obviously the table construction can be automated. The factorial values can be computed and the table rows can be generated. Unlike many other markup languages, SCRIBE enables this computation to take place inside the document itself. Let us assume the standard definitions for the `upto` and `fact` functions:

```
(define (upto min max)
   (if (= min max)
       (list max)
       (cons min (upto (+ min 1) max))))

(define (fact n)
   (if (< n 2)
       n
       (* n (fact (- n 1)))))
```

The generation of the factorial table requires two additional SCRIBE functions. The first one builds table rows:

```
(define (make-fact-row n)
   (tr (td :align 'center (bold n))
       (td :align 'right (it (fact n)))))
```

The second one is in charge of creating the table:

```
(define (make-fact-table n)
   (apply table :border 1
          (tr (th "n=") (th "fact"))
          (map make-fact-row (upto 3 n))))
```

3.2 Computing Sc-ast

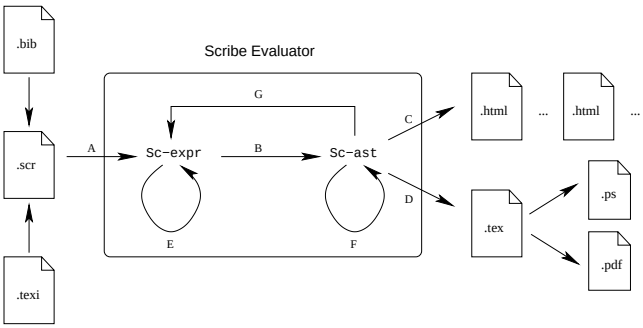

Figure 2: The Scribe process

The evaluation of a SCRIBE program involves three steps (see Figure 2):

- First, the source file is read and represented as a list of Sc-expressions (edge "A").
- Second, the Sc-expressions are evaluated using the standard SCRIBE library. This produces an abstract syntax tree named Sc-ast (edge "B").
- Third, the Sc-ast is translated into the target format *i.e.*, HTML, LaTeX , ... (edges "C" and "D").

The computations previously presented in Section 3.1 take place on the edge "E". This section focuses now on the computations that are involved on edges "F" and "G".

Frequently some parts of a document may refer to the document itself. For instance, *introspection* is needed to compute a table of contents. SCRIBE is provided with introspection facilities that can be used in user programs. For instance, it enables the computation of such a sentence:

"*This document contains 9 sections.*"

The actual number of sections is the result of a user computation. The whole sentence is computed by the following Sc-expression:

```
[This document contains
,(hook :after
       (lambda ()
          (display (length
                    (document-sections*
                     (current-document))))))
sections.]
```

It uses the SCRIBE library function `hook` which enables computations to take place while the Sc-ast is built, that is on the edge "F" of Figure 2. The `:after` argument is a function which is executed once the Sc-ast is translated into the target format. It *prints* a string that is inserted in the target. Obviously, the dynamic text of the previous example cannot be computed earlier in the SCRIBE evaluation process since the number of sections cannot be computed until all the sections are built! The function of the standard library `current-document` returns a structure that describes the document being processed. The function `document--sections*` returns the list of sections contained in a document. Not that, since the `hook` function enables arbitrary characters insertion, it can be used to introduce low level back-end commands such as TEX commands or HTML commands in the target. For instance, the SCRIBE command `LaTeX` which produces the following "LaTeX" is implemented as:

```
(define-markup (LaTeX)
   (if (scribe-format? 'tex)
       (hook :after (lambda () (display "\\LaTeX")))
       "LaTeX"))
```

Sometimes, instead of printing characters into the target, it is needed that the evaluation of a `hook` node produces a fresh Sc-expression. That is, an expression that has to be evaluated by the SCRIBE engine (the edge "G" of Figure 2)[1]. This is illustrated by the following example. The user function `document-tree` computes the hierarchical structure of a document. Applied to the current document it produces:

```
+--ABSTRACT
+--1 Introduction
+--2 Scribe overview
|  +--2.1 Sc-expressions
|  +--2.2 Scribe as a markup language
|  |  +--2.2.1 Scribe Markups
|  |  +--2.2.2 Document Structure
|  |  +--2.2.3 Scribe standard library
|  +--2.3 Front-ends and Back-ends
+--3 Dynamic texts
|  +--3.1 Computing Sc-expressions
|  +--3.2 Computing Sc-ast
|  +--3.3 Conditional execution
+--4 Customization
+--5 Related work
|  +--5.1 SGML and XML
|  +--5.2 Scheme vs. other functional languages
|  +--5.3 LAML
|  +--5.4 BRL
|  +--5.5 Wash
+--6 Conclusion
+--7 References
+--APPENDIX
```

Figure 3: Tree

The tree branches are displayed using a typewriter font and a layout that preserves spaces and line breaks. The tree

[1] Introducing a fresh Sc-expression in the tree may introduce incoherences for cross-references. When iterations are needed, it belongs to the programmer to implement it.

nodes are displayed underlined and in italic. The computation involved in `document-tree` produces a regular Scexpression that is evaluated by the SCRIBE engine. This ensures back-end independence because it prevents the `hook` call to specify how underline and italic have to be rendered for each specific target format. The function `document-tree` is defined as:

```
(define (document-tree)
   (hook :process #t
         :after (lambda ()
                  (prgm
                   (make-tree (current-document))))))
```

The argument `:process #t` means that the result of the application of the `:after` function has to be evaluated back by the SCRIBE engine. This function constructs a new Scexpression which is made of a `prgm` call. The definition of `make-tree` is:

```
(define (make-tree doc)
   (let loop ((s (scribe-get-children doc))
              (m "")
              (f underline))
        (if (null? s)
            '()
            (append (make-row m (car s) f)
                    (loop (scribe-get-children (car s))
                          (string-append m "|  ")
                          it)
                    (loop (cdr s) m f)))))
```

The function `make-row` is:

```
(define (make-row m s f)
   (list (string-append m "+--")
         (f (scribe-get-title s))
         "\n"))
```

The library function `scribe-get-children` returns the elements contained in a section or a subsection. The library function `scribe-get-title` returns the title of a section or a subsection.

In addition to illustrating SCRIBE introspection, this example also shows how suitable functional programming languages are to compute over texts: the whole implementation of Figure 3 is a simple recursive traversal of the tree representing the document (function `make-tree`).

3.3 Conditional execution

Conditional execution is required when the text to be produced depends on some properties of the target format. The `scribe-format?` predicate checks which target format is to be produced. It is used several times in the paper. For instance, in Section 3.1 we have presented the definition of the `Scribe.tex` macro. The actual macro used in the sources of this paper is slightly more complex. Instead of rendering the word "Scribe", when targeting HTML, it introduces a reference to the SCRIBE home page. Moreover, because of our poor English style, we have also decided to introduce an URL link only *once* per section. So, the actual function used in the paper source is defined as:

```
(define Scribe
   (let ((sec #f))
      (lambda ()
         (if (scribe-format? 'html)
             (hook :after
                   (lambda ()
                      (let ((s (current-section)))
                         (if (eq? s sec)
                             (Scribe.tex)
                             (begin
                                (set! sec s)
                                (ref :url (scribe-url)
                                     "Scribe")))))
                   :process #t)
             (Scribe.tex)))))
```

4. CUSTOMIZATION

A *real* and *practical* programming language is useful when considering *customizations* (in SCRIBE they usually take place in *style* files). SCRIBE customizations enable users to change the way documents are rendered. They are ubiquitous in the standard SCRIBE API. For instance, one may setup the way a bold text is rendered, configure the header and the footer of the document, or even define margins. One may also specify the structure of the produced documents. In this section we illustrate how one may benefit from the expressiveness of SCRIBE in order to achieve complex customizations. In particular, we will show how computers program can be rendered.

Depending of the specified language, SCRIBE uses different colors and fonts when rendering computer programs. The standard implementation supports several languages such as SCRIBE, Scheme, C, or XML. Computer programs are specified by the `prgm` markup (see Section 2.2.3) which accepts one optional argument which is a function implementing the rendering of the program. This function is called a *pretty-printer*. One may define its own pretty-printers.

For the sake of the example, let us implement a pretty-printer for rendering *makefiles* which uses some colors for `make` targets, variables, and comments. In addition, for back-ends supporting hyper links (such as HTML) a reference to its definition is added to the text when a variable is used. For other back-ends, variable references are underlined.

```
SCRIBE= scribe
SFLAGS= -J style

MASTER= main.scr
INPUT= abstract.scr intro.scr what.scr why.scr this.scr
EXAMPLE= ex0 ex1 ex2 ex3 ex4 makefile
STYLE= style/local.scr

# main entry
all: scribe.tex

scribe.tex: $(MASTER) $(INPUT) $(STYLE) $(EXAMPLE)
$(SCRIBE) $(SFLAGS) $(MASTER) -o scribe.tex
```

A pretty-printer function is a SCRIBE function accepting one parameter. This formal parameter is bound to a string representing the text to be pretty-printed. A pretty-printer returns a Sc-expression representing the pretty-printed program that must be included in the target document. The definition of the `makefile` pretty-printer is:

```
(define (makefile obj)
  (parse-makefile (open-input-string obj)))
```

In order to implement the pretty-printer we are using Bigloo regular parser [16]. This mechanism enables a lexical analysis of character strings.

```
(define (parse-makefile port::input-port)
   (read/rp
    (regular-grammar ()
       ((: #\# (+ all))
        ;; makefile comment
        (let ((cmt (the-string)))
           (cons (it cmt) (ignore))))
       ((bol (: (+ (out " \t\n:")) #\:))
        ;; target
        (let ((prompt (the-string)))
           (cons (bold prompt) (ignore))))
       ((bol (: (+ alpha) #\=))
        ;; variable definitions
        (let* ((len (- (the-length) 1))
               (var (the-substring 0 len)))
           (cons `(list ,(mark var)
                        (color :fg "#bb0000" (bold ,var))
                        ,"=") (ignore))))
       ((+ (out " \t\n:=$"))
        ;; plain strings
        (let ((str (the-string)))
           (cons str (ignore))))
       ((: #\$ #\( (+ (out " )\n")) #\))
        ;; variable references
        (let ((str (the-string))
              (var (the-substring 2 (- (the-length) 1))))
           (cons (ref :mark var (underline str))
                 (ignore))))
       ((+ (in " \t\n:"))
        ;; separators
        (let ((nl (the-string)))
           (cons nl (ignore))))
       (else
        ;; default
        (let ((c (the-failure)))
           (if (eof-object? c)
               '()
               (error "prgm(makefile)"
                      "Unexpected character"
                      c)))))
    port))
```

5. RELATED WORK

In this section we compare SCRIBE and other markup languages. We also compare it with other efforts for handling texts in functional programming languages.

5.1 SGML and XML

As stated in [3] *"XML, the Extensible Markup Language, is W3C-endorsed standard for document markup. It defines a generic syntax used to mark up data with simple, human-readable tags. It provides a standard format for computer documents"*. In other words, XML is a mean to specify external representations for data structures. It is a mere formalism for specifying grammars. It can be used to represent texts but this is not its main purpose. The most popular XML application used for representing texts (henceforth XML texts) is XHTML (a reformulation of HTML 4.0). XML can be thought as a simplification of SGML. They both share the same goals and syntax.

The fundamental difference between XML and SCRIBE is that the first one is definitely not a programming language. In consequence, any processing (formating, rendering, extracting) over XML texts requires one or several external tools using different programming languages which appear to be, most of the time, Java, Tcl, and C. A vast effort has been made to provide most of the functional programming languages with tools for handling XML texts. It exists XML parsers for mostly all functional programming languages. Haskell has HaXml [19], Caml has Px and Tony, and Scheme has SSax [7].

In addition to parsers, Scheme has also SXML [6] which is either an abstract syntax tree of an XML document or a concrete representation using S-expressions. SXML is suitable for Scheme-based XML authoring. It is a term implementation of the XML document.

The document style semantics and specification language (aka DSSSL [4]) defines several programming languages for handling SGML applications. The DSSSL suite plays approximatively the same role as XML XSLT, DOM and SSAX do: it enables parsing and computing over SGML documents. The DSSSL languages are based on a simplified version of Scheme.

XEXPR [21] is a scripting language that uses XML as its primary syntax. It has been defined to easily embed scripts inside XML documents and overcomes the usage of an external scripting language in order to process a document. The language defines itself to be *very close to a typical Lisp or combinator-based language where the primary means of programming is through functional composition.* XEXPR allows the definition of functions using the <`define`> element. Hereafter is a definition of the factorial function expressed in XEXPR:

```
<define name="factorial" args="n">
 <if>
   <lt><n/>2</lt>
   <n/>
   <multiply>
    <n/>
    <factorial>
      <substract><n/>1</substract>
    </factorial>
   </multiply>
 </if>
</define>
```

which must be compared with the Scheme version given in Section 3.1. Obviously, writing by hand large scripts seems hardly achievable in XEXPR. Furthermore, a careful reading

of the report defining this language seems to indicate that there is no way to manipulate the document itself inside an XEXPR expression. The language seems then limited to simple text generations inside an XML document, as the ones presented Section 3.1

Besides deploying one unique formalism and syntax for authoring documents we have found that SCRIBE enables more compact sources than XML (see Section 2.1). The SCRIBE syntax is less verbose than the XML one mainly because the closing parenthesis of a Sc-expression is exactly one character long when it is usually much more in XML.

5.2 Scheme vs. other functional languages

We have chosen to base SCRIBE on Scheme mainly because its syntax is genuinely close to traditional markup languages. Such as XML, the Scheme syntax is based on the representation of trees. The modifications to apply to the Scheme grammar are very limited and simple. This makes this language suitable for text representation. The other functional languages such as, Caml and Haskell, rely on LALR syntaxes that do not fit the markup look-and-feel.

In addition, we think that the Scheme type system is an advantage for SCRIBE programs. It is convenient to dispose of fully polymorphic data types. As presented in Section 2.1, an Sc-expression can be a list whose elements are of different types. For instance, the first element of such a list could be a character string and the next one a number. This enables compact representation of texts. If the underlying language imposes a stronger typing system, the source program, that is the user text, will be polluted with cast operations that transform all the values into strings.

We have considered using a *call-by-name* semantics for SCRIBE function application in order to implement the nesting of Sc-expressions. As presented in Section 3.2 the SCRIBE library proposes introspection functions. For instance, the `document-sections*` returns the list of sections contained in a document structured such as:

```
(document ...
   (chapter ...)
   (chapter
      ...
      (section ...)
      ...)
   ...)
```

The container nodes (representing documents, chapters, sections, ...) of the Sc-ast are provided with pointers to the children they contain and *vice versa*. Since laziness enables to postpone the computation of expressions until they are required, it can be used to delay the evaluation of inner elements of a document until the whole document is declared. We have obtained the same effect by adding a second traversal of the Sc-ast (see 3.2).

5.3 LAML

LAML (*Lisp as a Markup Language*) [13] is an attempt to use Scheme as a markup language. It mirrors the HTML markups in Scheme. That is, for each HTML markup there is a corresponding Scheme function in LAML. The HTML document:

```
<html>
  <head><title>An example</title></head>
  <body>
    <br>
    This is an <em>HTML</em>example.
    <br>
  </body>
</html>
```

is mirrored in LAML as:

```
(html
  (head (title "An example"))
  (body (br)
        "This is an" (em "HTML") "example."
        (br)))
```

So, LAML and SCRIBE are very close. They rely on the natural Scheme syntax and they both consider a document as a program. However, there is two important differences between them:

- The syntax: SCRIBE uses an extended Scheme syntax. As presented Section 2.1, it introduces the `[...]` notation that, as we have shown, enables compact source texts.
- The Sc-ast: The evaluation of a LAML function call directly produces an HTML expression. For instance, the definition of the LAML `em` function of the previous example is:

  ```
  (define (em str)
     (string-append "<EM>" str "</EM>"))
  ```

  Contrarily to SCRIBE, LAML does not build a tree representing the text to be generated. This direct mapping has three drawbacks:

  1. LAML sources cannot produce other formats than HTML.
  2. It is complex to implement efficiently a LAML interpreter. As reported in [12], the LAML evaluation process allocates a lot of strings of characters. This exercises intensively the memory manager (garbage collection and memory copies). These string manipulations are totally avoided by SCRIBE. One SCRIBE markup allocates one object that is a node of the Sc-ast. This node is used until the back-end has completed the file generation. It never happens that a node nor the characters is contains are duplicated.
  3. Introspection over a LAML document is complex. In particular, it has to take place before the string representing an HTML expression is built. That is, it has to take place before LAML functions are called. In other words, LAML is of no help for computing on documents. LAML users have to implement their own data representation before using LAML functions.

5.4 BRL

The *Beautiful Report Language* BRL [10] defines itself as a database-oriented language to embed in HTML and other markups. In some extent BRL approach is very similar to the PHP one: it proposes to mix the text and the program which form the document in the same source file. For BRL, a document is a sequence of either strings or Scheme expressions. BRL displays strings *as is* and *evaluates* Scheme expressions. To alleviate document typesetting using this conventions, BRL has introduced a new syntax for character strings: there is no need to put a quote for a string starting a file or terminating a file. Furthermore, "]" and "[" can be used to respectively open and close a string. So,

```
]a string[
```

is a valid string in BRL. The interest of this notation seems more evident in a construction such as

```
The value of pi is [(* 4 (atan 1))].
```

where we have a Scheme expression enclosed between two strings ("`The value of pi is`" and "`.`"). However, this syntax can be sometimes complex as it is shown in the following excerpt from the reference manual.

```
[(define rowcount (sql-repeat ...)
  (brl ]<li><strong>
<a href="p2.brl?[
(brl-url-args brl-blank? color)
]">(brl-html-escape color)]</a></strong>
[)))]
```

As we can see, BRL is just a sort of *preprocessor* and as such it cannot be used to do introspective work on a document.

5.5 Wash

Wash [18] is a family of embedded domain specific languages for programming Web applications. Each language is embedded in the functional language Haskell, which means that it is implemented as a combinator library. The basic idea of Wash is to build a data structure that can be rendered to HTML text. Because of the type system of the Haskell type checker, Wash guarantees the well-formedness of the generated HTML pages. Using a Haskell interpreter it is possible with Wash to interactively create and manipulate web pages.

If Wash shares with SCRIBE the construction of a data structure representing the text to be rendered, no effort is made to provide it with a concise syntax. A "hello, word" page which is in HTML:

```
<html>
  <head>
   <title>Hello, World</title>
  </head>
  <body>
    This is the traditional &#34;Hello, World!&#34; page.
    <hr>
  </body>
</html>
```

and that can be implemented in SCRIBE as:

```
(define *title* "Hello, World!")
(document :title "Hello, World" [
This is the traditional ,(begin *title*) page.
,(hrule)])
```

would be written in Wash as:

```
doc_head :: HEAD
doc_head =
  make_head
   `add` (make_title `add` "Hello, World")

doc_body :: BODY
doc_body =
  make_body
    `add` (make_heading 1 `add` title)
    `add` ("This is the traditional \""
           ++ title ++
           "\" page.")
    `add` make_hr
     where title = "Hello, World!"

doc :: HTML
doc = make_html `add` doc_head `add` doc_body

putStr (show_html doc)
```

It is obvious that Wash is designed *for* programmers. Unlike SCRIBE it cannot be as easily used in replacement of traditional markup languages.

6. CONCLUSION

SCRIBE is a functional programming language for authoring various kind of electronic documents. It can be used to produce target formats such as HTML and PostScript. It relies on an original syntax that makes it looking familiar to anyone used to markup languages such as HTML.

We have shown that the evaluation of a SCRIBE program involves two separate stages. During the first one the source expressions are read using the SCRIBE reader. These expressions are then evaluated using a classical Scheme interpreter. This stage produces an internal representation of the source program. The second evaluation stage uses that representation and, as a consequence, enables computations on the representation itself. That is, during the second stage a SCRIBE program may compute properties about itself.

SCRIBE is used on daily basis to produce large documents. For instance, the whole web page `http://www.inria.fr-/mimosa/fp/Bigloo` and the documentations it contains are implemented in SCRIBE. Obviously, the current paper *is* a SCRIBE program. An HTML version can be browsed at `http://www.inria.fr/mimosa/fp/Scribe/doc/scribe.html`.

7. REFERENCES

[1] Adobe System Inc. – **PostScript Language Reference Manual** – *Addison-Wesley, Readings, Massachusetts*, 1985.

[2] Goldfarb, C. – **The SGML Handbook** – *Oxford University Press*, 1991.

[3] Harold, E.R. and Means W.S. – **XML in a nutshell** – *O'Reilly*, Jan, 2001.

[4] ISO/IEC – **Information technology, Processing Languages, Document Style Semantics and Specification Languages (DSSSL)** – 10179:1996(E), *ISO*, 1996.

[5] Kelsey, R. and Clinger, W. and Rees, J. – **The Revised(5) Report on the Algorithmic Language Scheme** – *Higher-Order and Symbolic Computation*, 11(1), Sep, 1998.

[6] Kiselyov, O. – **Implementing Metcast in Scheme** – *Scheme workshop*, Montréal, Canada, Sep, 2000.

[7] Kiselyov, O. – **A better XML parser through functional programming.** – *Practical Aspects of Declarative Languages*, Portland, Oregon, USA, Jan, 2002.

[8] Knuth, D. – **The TEXbook,** – *Addison-Wesley, Readings, Massachusetts*, 1986.

[9] Lerdorf, R. – **PHP Pocket Reference** – *O'Reilly & Associates*, Jan, 2000.

[10] Lewis, B – **BRL Reference Manual** – `http://brl.sourceforge.net/`2002.

[11] McCarthy, J. – **Recursive functions of symbolic expressions and their computation by machine – I** – *Communications of the ACM*, 3(1), 1960, pp. 184–195.

[12] Nørmark, K. – **Programming World Wide Web Pages in Scheme** – *Sigplan Notices*, 34(12), 1999.

[13] Nørmark, K. – **Programmatic WWW authoring using Scheme and LAML** – *The Eleventh International World Wide Web Conference*, Honolulu, Hawaii, USA, May, 2002.

[14] Ossana, J. – **UNIX Programmer's manual: Supplementary Documents** – 1982.

[15] Queinnec, C. – **Literate programming from Scheme to TeX** – LIX RR 93.05, *Laboratoire d'Informatique de l' Polytechnique*, 91128 Palaiseau Cedex, France, Nov, 1993.

[16] Serrano, M. – **Bigloo user's manual** – 0169, *INRIA-Rocquencourt*, France, Dec, 1994.

[17] Sitaram, D. – **SLaTeX** – `http://www.ccs.neu.edu/home-/dorai/slatex/slatxdoc.html`.

[18] Thiemann, P. – **Modeling HTML in Haskell** – *Practical Aspects of Declarative Languages*, 2000, pp. 263–277.

[19] Wallace, M. and Runciman, C. – **Haskell and XML: Generic Combinators or Type-Based Translation?** – *Int'l Conf. on Functional Programming*, Paris, France, 1999.

[20] World Wide Web Consortium – **Document Object Model (DOM) Level 1 Specification** – *W3C Recommendation*, Oct, 1998.

[21] World Wide Web Consortium – **XEXPR - A Scripting Language for XML** – *W3C Note*, Nov, 2000.

APPENDIX

For the sake of the example, we present in this Annex, the whole SCRIBE source code for the abstract of this paper:

```
(paragraph [
This paper presents ,(Scribe), a functional programming
language for authoring documents. Even if it is a general
purpose tool, it best suits the writing of technical
documents such as web pages or technical reports, API
documentations, etc.  Executing ,(Scribe) programs can
produce documents of various formats such as PostScript,
PDF, HTML, Texinfo or Unix man pages. That is, the very
same program can be used to produce documents in different
formats. ,(Scribe) is a full featured programming language
but it ,(emph "looks") like a markup language ,(emph "à la")
HTML.
]))
```

This paper has been generated by Scribe (`http://www-sop.inria.fr-/mimosa/fp/Scribe`) (via LaTeX and the `ACMproc` class.)

Reachability-Based Memory Accounting

Adam Wick
awick@cs.utah.edu

Matthew Flatt
mflatt@cs.utah.edu

Wilson Hsieh
wilson@cs.utah.edu

University of Utah, School of Computing
50 South Central Campus Drive, Room 3190
Salt Lake City, Utah 84112–9205

ABSTRACT

Many language implementations provide a mechanism to express concurrent processes, but few provide support for terminating a process based on its resource consumption. Those implementations that do support termination generally charge the cost of a resource to the principal that *allocates* the resource, rather than the principal that *retains* the resource. The difference matters if principals represent distinct but cooperating processes.

In this paper, we present preliminary results for a version of MzScheme that supports termination conditions for resource-abusing processes. Unlike the usual approach to resource accounting, our approach assigns fine-grained (per-object) allocation charges to the process that retains a resource, instead of the process that allocates the resource.

1. MOTIVATION

Users of modern computing environments expect applications to cooperate in sophisticated ways. For example, users expect web browsers to launch external media players to view certain forms of data, and users expect a word processor to support active spreadsheets embedded in other documents. In a conventional operating system, however, programmers must exert considerable effort to integrate applications. Indeed, few software developers have the resources to integrate applications together as well as, for example, Adobe Acrobat in Microsoft's Internet Explorer.

Implementing cooperating applications in a conventional OS is difficult because the OS isolates applications to contain malfunctions. Cooperating applications must overcome this built-in isolation. In contrast, language run-time systems (a.k.a. "virtual machines") typically rely on language safety, rather than isolation, to contain malfunctions. Since VMs otherwise play the same role as OSes, and since they lack a bias towards isolation, safe VMs seem ideally suited as the platform for a next generation of application software.

Mere safety, however, does not provide the level of protection between applications that conventional OSes provide. Although language-based safety can prevent a program from trampling on another program's data structures, it cannot prevent a program from starving another process or from leaking resources. Regardless of the degree of cooperation, a practical OS/VM must track each application's resource consumption and prevent over-consuming applications from taking down the entire system.

A variation on conventional isolation can certainly enable resource tracking in a VM. For example, the VM can restrict values passed from one process to another to those values allocated within a certain pool of memory [1]. This compromise provides something better than a traditional OS, in that a suitably allocated value can be passed directly and safely between applications. Nevertheless, this kind of isolation continues to interfere with cooperation: even if a program can move values from one allocation pool to another, explicit accounting with allocation pools amounts to manual memory management as in `malloc` and `free`. This manual management encourages narrow communication channels; in order to foster communication, applications must be free to exchange arbitrary data with potentially complex allocation patterns.

We are investigating memory-management techniques that place the responsibility for accounting with the run-time system, instead of the programmer, while still enabling control over an application's memory use. The essential idea is that a garbage collector can account for memory use using reachability from an application's roots. Thus, an application is charged not for what it allocates, but for what it retains. This differentiation is critical in systems where one application may use memory allocated by another application. The central design problem is how to deal with these shared values usefully and efficiently.

We present preliminary results on our exploration, based on a new garbage collector for MzScheme [7]. Our results suggest that a garbage collector can maintain usefully precise accounting information with a low overhead, but that the implementation of the rest of the VM requires extra care to trigger reliable termination of over-consuming processes. This extra care is of the same flavor as avoiding references in the VM that needlessly preserve values from collection.

Section 2 describes the existing notion of "process" within MzScheme, and Section 3 describes our new API for resource enforcement. Section 4 describes in more detail possible accounting policies behind the API, including the two that we have implemented. Section 5 reports on our implementations, and Section 6 reports on our experience with them. Section 7 presents performance results.

Third Workshop on Scheme and Functional Programming. October 3, 2002, Pittsburgh, Pennsylvania, USA.

2. PROCESSES IN MZSCHEME

In MzScheme, no single language construct encompasses all aspects of a conventional process. Instead, various orthogonal constructs implement different aspects of processes:

- *Threads* implement the execution aspect of a process. The MzScheme thread function takes a thunk and creates a new thread to execute the thunk.

  The following example runs two concurrent loops, one that prints "1"s and another that prints "2"s:

```
(letrec ([loop (lambda (v)
                 (display v)
                 (loop v))])
  (thread (lambda () (loop 1)))
  (loop 2))
```

- *Parameters* implement process-specific settings, such as the current working directory. Each parameter is represented by a procedure, such as current-directory, that gets and sets the parameter value. Every thread has its own value for each parameter, so that setting a parameter value affects the value only in the current thread. Newly created threads inherit initial parameter values based on the current values in the creating thread.

  The following example sets the current directory to "/tmp" while running *do-work*, then restores the current directory:[1]

```
(let ([orig-dir (current-directory)])
  (current-directory "/tmp")
  (do-work)
  (current-directory orig-dir))
```

  Meanwhile, the current-directory setting of other executing threads is unaffected by the above code.

- *Custodians* implement the resource-management aspect of a process. Whenever a thread object is created, port object opened, GUI object displayed, or network-listener object started, the object is assigned to the current custodian, which is determined by the current-custodian parameter. The main operation on a custodian is custodian-shutdown-all, which terminates all of the custodian's threads, closes all of its ports, and so on. In addition, every new custodian created with make-custodian is created as a child of the current custodian. Shutting down a custodian also shuts down all of its children custodians.

  The following example runs *child-work-thunk* in its own thread, then terminates the thread after one second (also shutting down any other resources used by the child thread):

```
(let ([child-custodian (make-custodian)]
      [parent-custodian (current-custodian)])
  (current-custodian child-custodian)
  (thread child-work-thunk)
  (current-custodian parent-custodian)
  (sleep 1)
  (custodian-shutdown-all child-custodian))
```

  A thread's current custodian is *not* the same as the custodian that manages the thread. The latter is determined permanently when the thread is created. A thread can, however, change its current custodian at any time. In the above example, since *child-custodian* is current when the child thread is created, the child is placed into the management of *child-custodian*. Thus, (custodian-shutdown-all *child-custodian*) reliably terminates the child thread. In addition, if *child-custodian* is the only custodian accessible in *child-work-thunk*, then any custodian, thread, port, or network listener created by the child is reliably shut down by (custodian-shutdown-all *child-custodian*).

  Evaluating (current-custodian) immediately in *child-work-thunk* would produce *child-custodian*, because the initial parameter values for the child thread are inherited at the point of thread creation. The child thread may then change its current custodian at any time, perhaps creating a new custodian for a grandchild thread. Again, if *child-custodian* is the only custodian accessible in *child-work-thunk*, then newly created custodians necessarily fall under the management of *child-custodian*.

MzScheme includes additional constructs to handle other process aspects, such as code namespaces and event queues, but those constructs are irrelevant to accounting.

[1]Production code would use the **parameterize** form so that the directory is restored if *do-work* raises an exception.

3. ACCOUNTING API

Accounting information in MzScheme depends only on custodians and threads. Accounting depends on custodians because they act as a kind of process ID for termination purposes. In particular, since the motivation for accounting is to terminate over-consuming processes, MzScheme charges memory consumption at the granularity of custodians. Accounting also depends on threads, because threads encompass the execution aspect of a process, and the execution context defines the set of reachable values. Thus, the memory consumption of a custodian is defined in terms of the values reachable from the custodian's threads.

We defer discussion of specific accounting policies until the next section. For now, given that accounting is attached to custodians, we define a resource-limiting API that is similar to Unix process limits:

- (custodian-limit-memory *cust1 limit-k cust2*) installs a limit of *limit-k* bytes on the memory charged to the custodian *cust1*. If there comes a time when *cust1* uses more than *limit-k* bytes, then *cust2* is shut down.

  Typically, *cust1* and *cust2* are the same custodian, but distinguishing the accounting center from the cost center can be useful when *cust1* is the parent of *cust2* or vice-versa.

Although custodian-limit-memory is useful in simple settings, it does not compose well. For example, if a parent process has 100 MB to work with and its child processes typically use 1 MB but sometimes 20 MB, should the parent limit itself to the worst case by running at most 5 children? And how does the parent know that it has 100 MB to work with in the case of parent-siblings with varying memory consumption?

In order to address the needs of a parent more directly and in a more easily composed form, we introduce a second interface:

- (custodian-require-memory *cust1 need-k cust2*) installs a request for *need-k* bytes to be available for custodian *cust1*. If there comes a time when *cust1* would be unable to allocate *need-k* bytes, then *cust2* is shut down.

Using custodian-require-memory, a parent process can declare a safety cushion for its own operation but otherwise allow each child process to consume as much memory as is available. A parent can also combine custodian-require-memory and custodian-limit-memory to declare its own cushion and also prevent children from using more than 20 MB without limiting the total number of children to 5.

In addition to the two memory-monitoring procedures, MzScheme provides a function that reports a given custodian's current charges:

- (current-memory-use *cust*) returns the number of allocated bytes currently charged to custodian *cust*.

4. ACCOUNTING POLICIES

4.1 Reachability

As described in the previous section, we define a custodian's memory consumption in terms of the values reachable from the custodian's threads, as opposed to the values originally allocated by the threads. In addition, we require that the custodian hierarchy propagates accounting charges: if a custodian B is charged for a value, then its parent custodian is charged for the value as well.

Generally, reachability for accounting coincides with reachability for garbage collection. In particular, a value is not charged to a custodian if it is accessible through only weak pointers. Finalization poses no problem for accounting, because every finalizer in Mzscheme is created with respect to a *will executor*. Running a finalizer requires an explicit action on the executor, which means that a finalized object can be charged to the holder of the finalizer's executor.

Accounting reachability deviates from garbage-collection reachability in one respect. If a value is reachable from thread A only because thread A holds a reference to thread B, then B's custodian is charged and not A's (unless A's custodian is an ancestor of B's). Similarly, if a value is reachable by A only through a custodian C, then C is charged instead of A's custodian.

This deviation makes intuitive sense, because holding a reference to another process does not provide any access to the process's values. Moreover, this deviation is necessary for making accounting useful in our test programs, as we explain in Section 6.

4.2 Sharing

In a running system, some values may be reachable from multiple custodians. Different accounting policies might allocate charges for shared values in different ways, depending, on the amount of sharing among custodians, the hierarchical relationship of the custodians, the original allocator for a particular value or other factors. Among the policies that seem useful, we have implemented two:

- The *precise* policy charges every custodian for each value that it reaches. If two custodians share a value, they are both charged the full cost of the value. For example, in figure 1, objects w and z will be charged to both custodians A and B, object x will be charged to both custodians B and C, and object Y will be charged only to custodian C.

- The *blame-the-child* policy charges every value to at least one custodian, but not every custodian that reaches the value. The main guarantee for blame-the-child applies to custodians A and B when A is an ancestor of B; in that case, if A and B both reach some value, then A is charged if and only if B is charged. Meanwhile, if B and C share a value but neither custodian is an ancestor of the other, then at most one of them will be charged for the object. For example, in figure 1, object Y will be charged only to custodian C as in the precise policy. Also, since custodian B is a child of custodian A, B will necessarily be charged for W and Z. In the case of X, since there is no ancestral relationship between B and C, no guarantees are given as to which will be charged.

The precise policy is the most obvious one, and seems easiest to reason about. We have explored the blame-the-child policy, in addition, because it can be implemented more efficiently than the precise policy (at least in theory).

The blame-the-child policy, despite its imprecision, can work with custodian-limit-memory to control the memory consumption of a single sand-boxed application. Since the sand-boxed application will share only with its parent, accounting will reliably track consumption in the sand box.

Blame-the-child is less useful with custodian-limit-memory in a setting of multiple cooperating children. In that case, a well-behaved, cooperating application might incur all of the cost of all shared values, triggering the termination of the over-charged child (possibly leaving the rest stuck, lost without a collaborator). However, blame-the-child always works well with custodian-require-memory. With memory requirements instead of memory limits, how memory charges are allocated among children does not matter.

One policy that we have not explored is a variant of precise that splits charges among sharing custodians. For example, suppose that x custodians share a value of size y. With splitting, each of the x custodians would be charged y/x. This policy is normally considered troublesome, because terminating one of the x custodians triggers a sudden jump in the cost of the other $x-1$. Like blame-the-child, though, this policy might be useful with custodian-require-memory. We have not explored the cost-splitting policy because it seems expensive to implement, and it does not appear to offer any advantage over blame-the-child.

4.3 Timing

Ideally, a policy should guarantee the termination of a custodian immediately after it violates a limit or requirement. A naive implementation of this guarantee obviously cannot work, as it amounts to a full collection for every allocation.

The policies that we have implemented enforce limits and requirements only after a full collection. Consequently, a custodian can overrun its limit temporarily. This temporary overrun seems to cause no problems in practice, because a custodian that allocates lots of memory (and thus might violate limits or requirements) tends to trigger frequent collections. Furthermore, a failure in allocation for any custodian triggers a garbage collection which will then terminate usage offenders to satisfy the allocation.

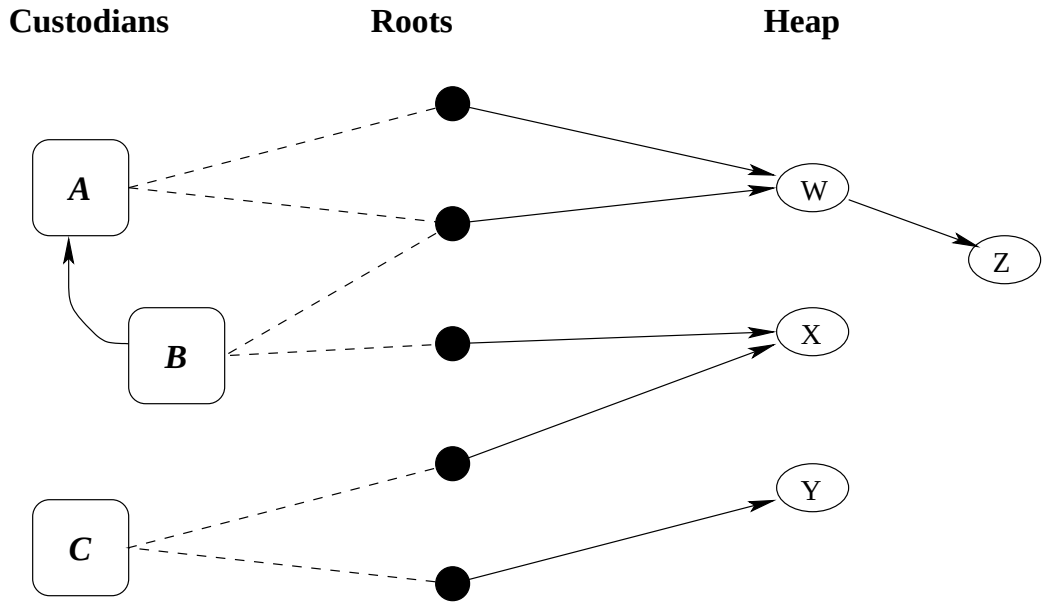

Figure 1: An example set of custodians and roots with a small heap

One potential problem is that a child overrun could push its parent past a limit, where terminating the child earlier might have saved the parent. Another problem is that a child overrun may occur at a time when custodians cannot be safely terminated. These potential problems have not appeared in practice, primarily because programmers cannot know the exact cost of values and must include significant safety margins. Nevertheless, the problems merit further investigation.

5. IMPLEMENTATION

The implementation of both the precise and blame-the-child policies proceeds roughly as follows:[2]

1. When a thread is created, the creating thread's current custodian is recorded in the new thread.
2. The collector's mark procedure treats thread objects as roots and as it marks from each thread, it charges the thread's custodian.
3. After collection, the collector checks the accumulated charges against registered memory limits and requirements. The collector schedules custodians for destruction (on the next thread-scheduling boundary) according to the comparison results.

Our two implementations differ only in the details of step 2. We first describe the implementation of precise accounting, then the implementation of blame-the-child accounting. Finally, we discuss the impact of generational garbage collection on the algorithms.

5.1 Precise Accounting

For precise accounting, the collector reserves space in the header of each object to record the object's set of charged

[2]The algorithms described should work in any collection system. We use the terminology of a mark/sweep style collector to simplify the description.

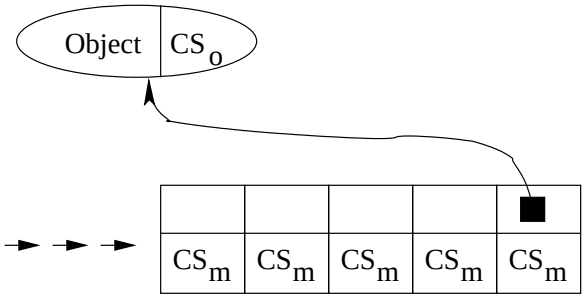

Figure 2: Mark queue with an object

custodians (CS_o in figure 2). During collection, the mark queue contains objects paired with the custodian set to be charged for the object. Initially, the charged set for all objects is the empty set. The initial mark queue contains all thread objects, where each thread is paired to the charged set containing only the thread's custodian.

When mark propagation reaches an object (see figure 2), the charged set in the object's header (CS_o) is compared to the charge used in marking (CS_m). If the charge set for marking is a subset of the charged set CS_o in the object header, no further work is performed for the object.[3] Otherwise, the union of the sets is computed and installed into the object's header, and charges for the object are shifted from the old set (if it is non-empty) to the unioned set. Marking continues with the object's content using the unioned set. After marking is complete, all garbage objects have an empty charged set, and the charges accumulated for each set can be relayed back to the set members.

[3]If the object contains a charge set, then it has been marked, and the mark propagation has either already been done or is queued. Since the item's charged set is a superset of the mark's charge set, then no additional information is available and no further work needs to be done.

In the case of a single custodian, the above algorithm degenerates to plain garbage collection, since the only possible charge sets are the empty set and the set containing the one custodian. In the case of c custodians, collection potentially requires c revisions to the charged set of every object. Thus, in the worst case, collection requires $O(c * r)$ time, where r is the size of reachable memory and c is the size of the set of all custodians. An entire heap containing only a single linked list with every thread pointing to the head of the list is an example of this worst case.

5.2 Blame-the-child

Unlike precise accounting, blame-the-child accounting requires only linear time in the amount of live memory. Roughly, the blame-the-child implementation works in the same way as the precise implementation, except that objects with non-empty charge sets are never re-marked. This change is enough to achieve linear time collection.

To completely implement the blame-the-child policy, the collector sorts the set of custodians before collection so that descendents precede ancestors. Then, the threads of each custodians are taken individually. Each thread is marked and the marks are propagated as far as possible before continuing with the next threads. Due to this ordering, objects reachable from both a parent and child will be first reached by tracing from the child's threads, and thus charged to the child. Once collection is complete, charges to child custodians are propagated back to their parents.

In our implementation, the blame-the-child implementation also incurs a smaller per-object overhead, since object headers need not contain a charge set. During marking, exactly one custodian is charged at a time, so that charges can be accumulated directly to the custodian. Each object needs only a mark bit, as in a normal collector.

A naive implementation of blame-the-child allows an obvious security hole. By creating sacrificial children, a malevolent custodian may arbitrarily delay its destruction when it uses too much memory. Such children would have pointers back into the malevolent custodian's space so that they would be blamed for its bad behavior. These, then, would be killed instead of the parent.

Several possible mechanisms can be used to keep this from happening, and we simply chose the easiest one from an implementation perspective. They are:

1. Place an order on the list of limits and requirements so that older custodians are killed first. In this case, the parent will be killed before the children, so creating sacrificial children is useless.

2. Kill every custodian that breaks a limit or requirement, rather than just one. Since a child's usage is included in the parent's usage, both will be killed.

3. Choose a random ordering. In this way, a malevolent program would have no guarantee that the above tactic would work.

Our implementation chooses the second tactic.

5.3 Generational Collection

After a full collection is finished and accounting is complete, comparing charges to registered limits and requirements is simple. Therefore, the collector can guarantee that a custodian is terminated after the first garbage collection cycle after which a limit or requirement is violated. This implies that there may be some delay between the detection of a violation and the actual violation. However, if the program is allocating this delay will be small, as frequent allocation will quickly trigger a garbage collection.

Accounting information after a minor collection is necessarily imprecise, however, since the minor collection does not examine the entire heap. Previously computed sets of custodians for older objects might be used regardless of changes since their promotion to an older generation. This old information may arbitrarily skew accounting. Worse, in the blame-the-child implementation described above, the collector does not preserve charges in object headers, so there is no information for older generations available to partial collections (except those that reclaim only the nursery).

Our implementation therefore enforces limits and requirements only after a full collection. This choice can delay enforcement by several collections, but should not introduce any new inherent potential for limit overruns, since overruns must lead to a full collection eventually.

6. EXPERIENCE

To determine the usefulness of our accounting policies in realistic environments, we wrote and modified several programs to take advantage of accounting. One program simply tests the ability of a parent to kill an easily sand-boxed child. A second program, DrScheme, tests child control where the parent and child work closely together. A third program, a web server allowing arbitrary servlet plug-ins, tests child control with some cooperation among the children.

6.1 Simple Kill Test

In the simple kill test, the main process creates a single sub-custodian, places a 64 MB limit on the sub-custodian's memory use, and creates a single thread in the sub-custodian that allocates an unbounded amount of memory:

```
(let ([child-custodian (make-custodian)])
  (custodian-limit-memory child-custodian
    (* 64 1024 1024) child-custodian)
  (current-custodian child-custodian)
  (thread-wait ; blocks until the thread completes
    (thread (lambda ()
              (let loop ()
                (+ 1 (loop)))))))
```

Since accounting works, the child custodian is destroyed, which in turn halts the child thread, and the entire program completes. If accounting were not successful, then the program would not terminate. Under both of our accounting system implementations, we find this program terminates. Unfortunately, it terminates several garbage collection cycles after the limit is actually violated.

Although simple, this program presents two items of interest. First, it shows that the blame-the-child policy can work, and that it allows the natural creation of parent/child pairs where the parent wishes to limit its children. Second, the program shows that generational collection does delay the detection of resource overruns.

Safety nets in our garbage collector assure that a program does not run out of available memory before its limit is noticed, but in systems with tight memory requirements, our technique may not be acceptable. We are investigating ways

to detect overruns more quickly.

6.2 DrScheme

The DrScheme programming environment consists of one or more windows, where each window is split into two parts. The top part of the window is used to edit programs. The bottom part is an interactive Scheme interpreter loop where the program can be tested. Each interpreter (one per window frame) is run under its own custodian. With a single line of code, we modified DrScheme to constrain each interpreter to 16 MB of memory.

Initial experiments with the single-line change did not produce the desired result, even with precise accounting. After opening several windows, and after making one interpreter allocate an unbounded amount of memory, *every* interpreter custodian in DrScheme terminated. Investigation revealed the problem:

- Each interpreter holds a reference into the DrScheme GUI. For example, the value of the parameter `current-output-port` is a port that writes to the text widget for the interaction half of the window. The text widget holds a reference to the whole window, and all open Drscheme windows are chained together.

- Each window maintains a reference to the interpreter thread, custodian, and other interpreter-specific values, including the interpreter's top-level environment.

Due to these references, every interpreter thread reaches every other interpreter's data through opaque closures and objects, even though programs running in different interpreters cannot interfere with each other. Hence, in the precise accounting system, every thread was charged for every value in the system, which obviously defeats the purpose of accounting.

Correcting the problem required only a slight modification to DrScheme. We modified it so that a window retains only weak links to interpreter-specific values. In other words, we disallow direct references from the parent to the child. Thus a child may trace references back to the parent's values, but will never trace these references back down to another child.

Finding the parent-to-child references in DrScheme—a fairly large and complex system—required only a couple of hours with garbage-collector instrumentation. The actual changes required only a half hour. In all, five references were changed: two were converted into weak links, two were extraneous and simply removed, and one was removed by pushing the value into a parameter within the child's thread.

Breaking links from parent to child may seem backward, but breaking links in the other direction would have required far too much work to be practical. For example, we could not easily modify the interpreter-owned port to weakly reference the DrScheme window. The port requires access to many internal structures within the GUI widget. Indeed, such a conversion would amount to the file-descriptor/handle approach of conventional operating systems—precisely the kind of design that we are trying to escape when implementing cooperation.

6.3 Web Server

In the DrScheme architecture, children never cooperate and share data. In the web server, however, considerable sharing exists between child processes. Whenever a server connection is established, the server creates a fresh custodian to take charge of the connection. If the connection requires the invocation of a servlet, then another fresh custodian is created for the servlet's execution. However, the servlet custodian is created with the same parent as the connection custodian, not as a child of the connection custodian, because a servlet session may span connections. Thus, a connection custodian and a servlet custodian are siblings, and they share data because both work to satisfy the same request.

The precise accounting system performs well when a servlet allocates an unbounded amount of memory. The offending servlet is killed right after allocating too much memory, and the web server continues normally.

The blame-the-child system performs less well, in that the servlet kill is sometimes delayed, but works acceptably well for our purposes. The delayed kill with blame-the-child arises from the sibling relationship between the connection custodian and the servlet custodian. When the servlet runs, the connection is sometimes blamed for the servlet's memory use. In practice, this happens often. The result is that the connection is killed, and then the still-live memory is not charged to the servlet until the next garbage collection. This example points again to the need for better guarantees in terms of the time at which accounting charges trigger termination, which is one subject of our ongoing work.

7. PERFORMANCE EVALUATION

Memory accounting incurs some cost, with trade-offs in terms of speed, space usage, and accounting accuracy. To measure these costs, we have implemented these two memory accounting systems within MzScheme.[4] Our collector is a generational, copying collector[8] implemented in C. This collector is designed for production-level systems; it can handle all situations that the default MzScheme garbage collector handles, including finalizers which may resurrect dying objects. For analysis purposes, the collector can be tuned statically to behave as one the following:

- **NoAccnt**: The base-line collector. No memory accounting functionality is included in this collector.

- **Precise**: The base-line collector plus the memory accounting system described in section 5.1.

- **BTC**: The base-line collector plus the memory blame-the-child accounting system described in section 5.2.

We evaluate the space usage, accuracy and time penalty of the **BTC** and **Precise** collectors on the following benchmark programs:

- **Prod**: An implementation of a producer/consumer system, with five producers and five consumers paired off. A different custodian is used for each producing or consuming thread. This case covers situations wherein sibling custodians share a large piece of common data; in this case, they share a common queue.

- **Kill**: A basic kill test for accounting. A child custodian is created and a limit is placed on its memory use.

[4]Accounting builds on the "precisely" collected variant of MzScheme, instead of the "conservatively" collected variant.

Test	Precise		BTC	
	# of owner sets	Additional required space	# of owner sets	Additional required space
Web	360	60,054 bytes	360	30,570 bytes
Prod	35	3,842 bytes	21	1,130 bytes
DrScheme	15	6100 bytes	9	5076 bytes
PSearch	4	266 bytes	3	186 bytes
Kill	2	146 bytes	2	146 bytes

Figure 3: Additional space requirements for accounting.

	NoAccnt		BTC			Precise		
	Time	S.D.	Time	S.D.	% slowdown	Time	S.D.	% slowdown
Web	1.30	0.05	1.77	0.06	36.2%	1.80	0.06	38.5%
Prod	2.60	0.05	1.31	0.04	n/a	1.41	0.04	n/a
DrScheme	23.10	0.14	23.55	0.11	1.7%	43.19	1.73	87.0%
PSearch	2.33	0.12	2.41	0.12	3.4%	2.42	0.13	3.9%
Kill	n/a	n/a	1.74	0.03	n/a	1.76	0.04	n/a

Figure 4: Timing results in seconds of user time with standard deviations. Where applicable, the table provides a percentage slowdown relative to the *NoAccnt* collector. All benchmark programs were run on a 1.8Ghz Pentium IV with 256MB of memory, running under FreeBSD 4.3 and MzScheme (or DrScheme) version 200pre19.

Under the child custodian, memory is then allocated until the limit is reached. This case covers the situation wherein proper accounting is necessary for the proper functioning of a program.

- **PSearch**: A search program that seeks its target using both breadth-first and depth-first search and uses whichever it finds first. This case is included to consider situations where there are a small number of custodians, but those custodians have large, unshared memory use.

- **Web**: A web server using custodians. This test was included as a realistic example where custodians may be necessary. The server is initialized, and then three threads each request a page 200 times. Every thread on the server side which answers a query is run in its own custodian.

- **DrScheme**: A program, run inside DrScheme, that creates three custodian/thread pairs and starts a new DrScheme process in each.

7.1 Space Usage

Regardless of the implemented policy, some additional space is required for memory accounting. Space is required internally to track the custodian of registered roots, and to track owner sets. In the case of **Precise**, additional space may be required for objects whose headers do not contain sufficient unused space to hold the owner set information for the object.

In our tests, the space requirements usually depend on the number of owner sets. Figure 3 shows the amount of space required for each of our test cases. These numbers show the additional space overhead tracking, roughly, the number of owner sets in the system. The numbers for **DrScheme** do not scale with the others because the start-up process for the underlying GUI system installs a large number of roots.

As expected, the additional space needed for precise accounting is somewhat larger than the space required for blame-the-child accounting. This space is used for union sets (owner sets which are derived as the union of two owner sets), and the blame-the-child implementation never performs a set union. The difference thus depends entirely upon the number of custodians and the sharing involved.

The MzScheme distribution includes a garbage collector that is tuned for space. In particular, it shrinks the headers of one common type of object, but this shrinking leaves no room for owner set information. Compared to the space-tuned collector, the **NoAccnt** and the accounting collectors require between 15% and 35% more memory overall.

7.2 Accuracy

To check the accuracy of memory accounting for different collectors, we tested each program under the precise system and compared the results to the blame-the-child system. The results were exactly as expected: the blame-the-child algorithm accounts all the shared memory to one random child. For example, in **DrScheme**, precise accounting showed that around 49 MB of data was shared among the children. Under **BTC**, one of the custodians (and not necessarily the same one every time) and its parent were charged 49 MB, but the other two child custodians were charged only for local data (around 80 KB each).

7.3 Time efficiency

To measure the trade-off between the accuracy of accounting information and the execution speed of the collector (and hence the program as a whole), we recorded the total running time of the test programs. Figure 4 shows the results of these benchmarks.

In every case, precise accounting takes additional time. The amount of additional time depends on the number of custodians, the amount of sharing among the custodians, and the size of the data set. In **Web**, **Prod**, **PSearch**, and **Kill**, the custodians and heap are arranged so that the additional penalty of precise accounting (that is, the penalty beyond that of **BTC** accounting) is minimal. The greatest slowdown in those cases, around two percent, is for **Web**. In

contrast, for cases where there is considerable sharing and the heap is large, the penalty for precise accounting can be quite large. **DrScheme** fits this profile, and the slowdown for precise accounting is predictably quite high.

Blame-the-child accounting also incurs a performance penalty. In both **DrScheme** and **PSearch**, the penalty is small. In **Web**, the penalty is significant. The difference between the former two tests and the latter one is primarily in the number of owner sets they use. The penalty difference, then, may result from cache effects during accounting. Since owner-set space usage is kept in a table, this table may become large enough that it no longer fits in cache. By reading and writing to this table on every mark, a large number of owner sets imply considerably more cache pressure and hence cache misses. In ongoing work, we are investigating this possibility.

The strange case in our results is **Prod**. In this case, the work of accounting actually speeds up the program. In ongoing work we are trying to determine the cause of the speed-up.

8. RELATED WORK

Recent research has focused on providing hard resource boundaries between applications to prevent denial-of-service. For example, the KaffeOS virtual machine [1] for Java provides the ability to precisely account for memory consumption by applications. Similar systems include MVM [5], Alta [2], and J-SEAL2 [4]. This line of work is limited in that it constrains sharing between applications to provide tight resource controls. Such restrictions are necessary to execute untrusted code safely, but they are not flexible enough to support high levels of cooperation between applications.

More generally, the existing work on resource controls—including JRes [6] and research on accounting principals in operating systems, such as the work on resource containers [3]—addresses only resource allocation, and does not track actual resource usage.

9. CONCLUSIONS

We have presented preliminary results on our memory-accounting garbage collection system for MzScheme. Our approach charges for resource consumption based on the retention of values, as opposed to allocation, and it requires no explicit declaration of sharing by the programmer. Our policy definitions apply to any runtime system that includes a notion of accounting principles that is tied to threads,

In the long run, we expect our blame-the-child accounting policy to become the default accounting mechanism in MzScheme. It provides accounting information that seems precise enough for many applications, and it can be implemented with a minimal overhead.

The main question for ongoing work concerns the timing of accounting checks. Our current implementation checks for limit violations only during full collections, and the charges for a terminated custodian are not transferred until the following full collection. Both of these effects delay the enforcement of resource limits in a way that is difficult for programmers to reason about, and we expect that much better guarantees can be provided to programmers.

A second question concerns the suitability of weak links for breaking accounted sharing between a parent and child, and perhaps between peers. The current approach of weakening the parent-to-child links worked well for our test programs, but we need more experience with cooperating applications.

The collectors described in this paper are distributed with versions 200 and above of the PLT distribution of Scheme for Unix.[5] Interactive performance of the accounting collectors is comparable to the performance of the default collector, although some pause times (particularly when doing precise accounting) are noticeably longer.

10. REFERENCES

[1] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, Oct. 2000. USENIX.

[2] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Java operating systems: Design and implementation. In *Proceedings of the USENIX 2000 Technical Conference*, pages 197–210, San Diego, CA, June 2000.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. ACM Symposium on Operating System Design and Implementation*, Feb. 1999.

[4] W. Binder, J. G. Hulaas, and A. Villazón. Portable resource control in java: The J-SEAL2 approach. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 139–155, 2001.

[5] G. Czajkowski and L. Daynès. Multitasking without compromise: a virtual machine evolution. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 125–138, 2001.

[6] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 21–35, 1998.

[7] M. Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997. `http://download.plt-scheme.org/doc/`.

[8] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.

[5] Configure with `--enable-account` and make the `3m` target.

Processes vs. User-Level Threads in Scsh

Martin Gasbichler Michael Sperber
Universität Tübingen
{gasbichl,sperber}@informatik.uni-tuebingen.de

Abstract

The new version of scsh enables concurrent system programming with portable user-level threads. In scsh, threads behave like processes in many ways. Each thread receives its own set of process resources. Like Unix processes, forked threads can inherit resources from the parent thread. To store these resources scsh uses *preserved thread fluids*, a special kind of fluid variables. The paper gives a detailed description of an efficient implementation for thread-local process resources. Scsh also provides an interface to the `fork` system calls which avoids common pitfalls which arise with a user-level thread system. Scsh contains a binding for `fork` that forks "only the current thread."

1 Introduction

Scsh [14] is a variant of Scheme 48 [11, 10] with extensive support for Unix systems and shell programming. Specifically, it contains full access to all basic primitive functions specified by POSIX. Scsh 0.1, the first version, came out in 1994.

In late 1999, the scsh maintainers set out to produce a version of scsh capable of multithreading. The main motivation was to improve scsh's abstraction of the operation system [15] as well as to implement multi-threaded Internet servers with scsh. At the time, scsh was based on Scheme 48 version 0.36 which did not support multithreading. Meanwhile, Scheme 48 had reached version 0.53 which did support fast, preemptive user-level multithreading. Hence, the task was originally to disentangle scsh from the underlying 0.36 substrate and port it to 0.53.

However, once the basic porting work was finished, it turned out that some of the POSIX functionality interfered with the user-level threads. Writing multi-threaded scsh programs is easiest when threads behave mostly like processes. However, this analogy breaks in a straightforward implementation of user-level threads and POSIX system calls in two important respects:

- A number of system resources, such as the environment or current working directory are process-*local* but thread-*global*. This would cause programs which would work correctly in a single-threaded system to interfere with each other when run concurrently in multiple threads, even though there is no explicit shared state or communication with other threads.
- The POSIX `fork` system call would copy the entire process, and all threads of the parent would also run in the child. This interferes with the intuition of the programmer who expects "only the current thread to fork." Moreover, it causes a number of race conditions associated with the `fork`/`exec`* pattern common in POSIX programming. Worse, the programmer cannot work around this problem easily because the primitives of the thread system are not powerful enough.

Moreover, the C library causes some problems: The `syslog` interface to the system's message logging facility offers only a single global, implicit connection which needs to be multiplexed among threads. Also, some POSIX library calls block indefinitely, making timely preemption of threads impossible while they are running. The most notable examples are `gethostbyname` and `gethostbyaddress` whose functionality is indispensable for implementing multi-threaded Internet servers.

This paper describes the steps taken in scsh 0.6 towards handling or working around those problems:

- Scsh represents thread-local process resources by *thread fluids*, thread-local cells which support binding, assignment, and preservation across a `fork`-like operation on threads.
- Scsh uses *resource alignment* to lazily keep the internal representation synchronized with the process state.
- Scsh's thread system supports a novel primitive called `narrow` which allows implementing a fork operation that forks "only the current thread."

Overview Section 2 gives a brief account of the Scheme 48 thread system. Section 3 briefly describes the process resource functionality POSIX offers. Section 4 describes thread fluids which scsh uses to represent process resources. Next, Section 5 describes how to use thread fluids to keep the thread-local process state lazily aligned with the actual process state. Scsh's implementation of `fork` is described in Section 6. Section 7 describes some of the miscellaneous problems with integrating the standard C library with user-level threads. Section 8 reviews some related work, and Section 9 concludes.

Third Workshop on Scheme and Functional Programming. October 3, 2002, Pittsburgh, Pennsylvania, USA.

2 The Scheme 48 Thread System

Concurrency within scsh is expressed in terms of the user-level thread system of Scheme 48 [2]. Its structure is inspired by nestable engines [7, 8, 4] and is almost entirely implemented in Scheme, and therefore extensible without changing the VM in any way. The VM supports the thread system in two ways:

- It schedules timer interrupts and thus allows preemption of running threads off the interrupt handler.[1]
- The VM I/O primitives are non-blocking. The VM manages queues of outstanding I/O requests and schedules interrupts as they become enabled.

Each thread is represented by a Scheme object which, while it is running, keeps track of its remaining time before preemption. As in other engines-based thread systems, Scheme 48 uses continuations for saving and restoring the control contexts of threads. For a blocked thread, the thread object contains a saved continuation and an interrupt mask.

As in any continuations-based system, Scheme 48 needs to take `dynamic-wind` into account: for context switching, the thread system employs the `primitive-cwcc` VM primitive which merely reifies the VM-level continuation. Each thread object also keeps track of the dynamic environment and the dynamic wind point, which in turn are used to implement `dynamic-wind` and the full-scale `call-with-current-continuation`.

The dynamic environment contains thread-local bindings for *fluid variables* (or just *fluids*) that implement a form of dynamic binding local to a single thread. Specifically, Scheme 48 holds the current input and output ports in fluids. Fluids play a crucial role in coordinating thread-local state and process state. Section 4 explains this issue in detail.

Each thread is under the control of a *scheduler*, itself a regular thread. Schedulers nest, so all threads in the running system are organized as a tree. A scheduler can run a thread for a slice of its own time by calling (`run` *thread time*). The call to `run` returns either when the time slice has expired or an *event* happened. This event might signify termination, an interrupt, a blocked operation, another thread becoming runnable, or a request from the thread to the scheduler. For example, a thread can cause the scheduler to spawn a new thread by returning a `spawned` event along with a thunk to be run in the new thread. Note that it is easily possible to pass an event upwards in the thread tree if the current scheduler is unwilling to handle it.

Thus, a scheduler performs at least two tasks: it implements a scheduling policy by deciding which threads to run for how long, and it must handle events returned by `run`.

A non-interactive Scheme 48 process has only a single *root scheduler*. The root scheduler, in addition to managing its subordinate threads, also periodically wakes sleeping threads and takes care of port flushing. An interactive Scheme 48 also has a scheduler for each *command level* that encapsulates a state of interaction with the user. This allows Scheme 48 to cleanly interrupt all running threads at any time by entering a new command level, and later continue them by throwing back into an old one. The built-in schedulers all use a simple round-robin scheduling policy.

[1] Scsh restarts system calls interrupted by the timer at the Scheme level.

3 Unix Process Resources

The representation of a process within the kernel of a Unix operation system contains several *process resources*. The kernel initializes these resources during creation of a process, typically by copying the values from the parent process. Here are the most important process resources:

- the current working directory,
- the file mode creation mask, called *umask*,
- the user and group ID,
- the environment.

For each resource the kernel provides system calls to read and set the resource. For the current working directory, `getcwd` returns the path as a string and `chdir` sets the directory to a new path.

A number of system calls implicitly consult the resources of the calling process. In the current working directory example, when the process uses the `open` system call to open a file, the kernel interprets the filename argument of `open` relative to the value of the current working-directory resource. Likewise, `chdir` resolves its path argument relative to the current working directory if it does not start with a slash.

4 Scheme Thread-Local Resources

Threads share state. This enables inter-thread communication by explicitly providing to several threads access to shared state by lexical binding. The various process resources, however, constitute *implicit* state, just like the settings for `current-input-port` and `current-output-port`.

For managing the latter, Scheme 48 keeps their values in *fluid bindings*: a fluid is a cell that allows dynamic binding. (`make-fluid` *v*) creates a fluid with default value *v*, (`fluid` *f*) references the value bound to a fluid, and (`let-fluid` *f v t*) calls thunk *t* with fluid *f* bound to value *v* during the dynamic extent of this call. That is, the fluid mechanism resets the binding to the value before the `let-fluid` if the thunk calls a previously stored continuation, and if the thunk *t* captures a continuation, on a later call to this continuation the fluid mechanism again binds the fluid *f* to the value *v*. Here are some examples from a Scheme 48 session, where `>` marks the command prompt and `call-with-current-continuation` is abbreviated as `call/cc`:

```
> (define f (make-fluid 1))
> (fluid f)
⇝ 1
```

`Let-fluid` binds the fluid only during the execution of the thunk:

```
> (+ (let-fluid f 3 (lambda () (fluid f)))
     (fluid f))
⇝ 4
```

Save a continuation with a dynamic binding in `*k*`:

```
> (define *k*)
> (let-fluid f 25
      (lambda ()
         (* (call/cc (lambda (k) (set! *k* k) 10))
            (fluid f))))
⇝ 250
```

The top-level binding is still the initialization value:

```
> (fluid f)
⇝ 1
```

Throwing back into the thunk using the stored continuation reactivates the binding introduced by the `let-fluid` above:

```
> (*k* 100)
⇝ 2500
```

Capture a continuation that returns the value of the fluid added to the argument of the continuation:

```
> (define *kk*)
> ((lambda (x) (+ x (fluid f)))
   (call/cc
     (lambda (k)
       (set! *kk* k)
       20)))
⇝ 21
```

Calling the stored continuation amounts to throwing out of the dynamic extent of the thunk:

```
> (let-fluid f -1
      (lambda ()
         (*kk* 3)))
⇝ 4
```

The environment that associates fluids with their values is local to each thread. Each newly spawned thread gets a fresh dynamic environment from its scheduler, typically with all fluids bound to their default values:

```
> (define f (make-fluid 1))
```

Start a new thread:

```
> (let-fluid f 23
    (lambda ()
       (spawn (lambda ()
                (display (fluid f))))))
⇝ prints 1
```

For process resources, sharing their settings among the threads is undesirable, as threads might interfere with each other, even though there is no explicit, intended communication among them. Moreover, it often makes more sense to dynamically bind a process resource rather than mutate it permanently. (To this end, scsh has always offered constructs like `with-cwd`, `with-env` etc.)

Therefore, fluids seem like the right low-level means for implementing thread-local process resources. However, they do not support assignment, primarily because its intended semantics is not immediately obvious: should assignments be visible in other threads?[2] Scsh therefore offers a primitive mechanism orthogonal to fluids called *thread-local cells* or *thread cells*: a thread cell supports assignment, and assignment is always thread-local. (`make-thread-cell` *v*) returns a thread cell with default value *v*, (`thread-cell-ref` *c*) returns the current contents of the cell, and (`thread-cell-set!` *c v*) mutates the cell's value as seen by the current thread to *v*.

```
> (define a-cell (make-thread-cell 23))
> (thread-cell-ref a-cell)
⇝ 23
```

Start a new thread which mutates the cell:

```
> (spawn (lambda ()
           (thread-cell-set! a-cell 42)
           (let lp ()
              (display (thread-cell-ref a-cell))
              (lp))))
⇝ Keeps printing 42 until the end of days
```

The top-level thread still sees the initial value:

```
> (thread-cell-ref a-cell)
⇝ 23
```

Moreover, scsh also ships with an abstraction built upon thread cells—*thread fluids*. Thread fluids obey the rules of dynamic binding just as ordinary fluids but also support mutation like thread cells. In fact, a thread fluid corresponds to a fluid containing a thread cell. Here is the transcript of a Scheme 48 session using thread fluids:

```
> (define f (make-thread-fluid 1))
```

Save a continuation with a dynamic binding in `*k*`:

```
> (define *k*)
> (let-thread-fluid f 25
     (lambda ()
        (* (call/cc (lambda (k) (set! *k* k) 10))
           (thread-fluid f))))
⇝ 250
```

Modify the value of the thread fluid:

```
> (set-thread-fluid! f -1)
> (thread-fluid f)
⇝ -1
```

A call to the stored continuation shows that the dynamic binding is still active:

```
> (*k* 100)
⇝ 2500
```

To sum up, a thread fluid supports *both* binding and thread-local assignment, thereby offering the right functionality for representing process resources per thread.

Just as with fluids, a newly spawned thread receives the default values for the thread-fluid bindings from its scheduler, rather than from the thread which evaluated the call to `spawn`. This is contrary to how process resources work, where the child inherits from the parents.[3] Simple lexical bindings allows communicating a thread fluid to a spawned thread "by hand:"

[2] The *parameter* mechanism of MzScheme [6] supports both binding and assignment. Assignment is always thread-local. The (as of the time of writing) soon-to-be-released version of Gambit-C also has parameters. These will have "binding-local" assignment: assignment by default is visible in other threads unless there is an intervening binding [5].

[3] In fact, in MzScheme, a spawned thread inherits the parameter bindings from the spawning thread. However, the built-in `error-escape-handler` parameter alone does not propagate to spawned threads—this would cause a space leak [1]. The potential for space leaks alone suggests that the programmer should have control over the propagation of thread fluid values to spawned threads.

```
(define t-fluid (make-thread-fluid #f))
  ...
  (spawn
    (let ((val (thread-fluid t-fluid)))
     (lambda ()
       (let-thread-fluid t-fluid val
        ...))))
```

The thread-fluids library exports two procedures `make-preserved-thread-fluid` and `preserve-thread-fluids`: `make-preserved-thread-fluid` is just like `make-thread-fluid`, but marks the thread fluid for preservation. `Preserve-thread-fluids` accepts a thunk as an argument and returns another thunk wrapped in pairs of `let` and `let-thread-fluid` forms for all live thread fluids marked for preservation. Thus, the above code could be rewritten as:

```
(define t-fluid (make-preserved-thread-fluid #f))

  ...
  (spawn
    (preserve-thread-fluids
     (lambda ()
      ...)))
```

The `thread-fluids` package also exports a procedure `fork-thread` with the following definition:[4]

```
(define (fork-thread thunk . rest)
  (apply spawn (preserve-thread-fluids thunk) rest))
```

Now a forked thread can inherit values from its parent:

```
> (define f (make-preserved-thread-fluid 0))
> (let-thread-fluid f 1
      (lambda ()
         (fork-thread
            (lambda () (display (thread-fluid f))))))
⇝ prints 1
```

Mutation of preserved thread fluids is still thread-local:

```
> (begin
     (let-thread-fluid f 1
         (lambda ()
            (fork-thread
               (lambda () (set-thread-fluid! f -1)))))
     (thread-fluid f))
⇝ 0
```

5 Thread-Local Process Resources

To enable modular system programming in the presence of threads the values of process resources must be local to each thread. To mimic processes, freshly created threads should inherit the resources from their parents. Preserved thread fluids provide the right vehicle to store the values within the threads, but communicating the values to the actual process resources requires additional machinery.

A simple approach to implement thread-local process resources is to adjust the process resources on a thread context switch: If the scheduler suspends the current thread the values of all resources are saved in thread fluids. Before the scheduler runs the next thread, it updates the process resources with the values of the thread fluid of the respective thread. This means that the process resources are *aligned* with the thread fluids on a context switch. Unfortunately, this method requires system calls for saving and restoring on each context switch as well as crossing the C foreign function interface boundary, both of which are comparatively expensive.

As the kernel inspects the process resources only during certain system calls, it is not required that process resources and thread fluids match all the time. It is sufficient to align a process resource when the thread actually performs a system call which is affected by the resource. The `open` system call would then be defined as:

```
(define (open filename)
  (chdir-syscall (thread-fluid $cwd)
  (set-umask-syscall (thread-fluid $umask)
  (open-syscall filename))
```

This code has a race condition: Another thread could align the umask and the current working directory with its own values before the `open`. Locks remedy this problem by performing alignment and the actual system call atomically:

```
(define cwd-lock (make-lock))
(define umask-lock (make-lock))
(define (open filename)
  (obtain-lock cwd-lock)
  (obtain-lock umask-lock)
  (chdir-syscall (thread-fluid $cwd))
  (open-syscall filename)
  (release-lock umask-lock)
  (release-lock cwd-lock))
```

`Make-lock` creates a standard mutex lock. After one thread has called `obtain-lock` on this lock all other threads doing the same will block until the lock is released by `release-lock`.

The performance of this approach is still not optimal: for each `open`, scsh executes one `chdir` and one `set-umask`, regardless of the actual values of the respective resources. Scsh caches the value of the process resource whenever it is changed and compares the cache with the thread fluid to determine if the process needs to align with the resource. The rest of the section describes how scsh implements this strategy for the various process resources.

The umask case is the simplest. There is a cache and a replacement for `set-umask` that sets the cache:

```
(define *umask-cache* (process-umask)[5])
(define umask-lock (make-lock))
(define $umask (make-preserved-thread-fluid (umask-cache)))

(define (umask-cache)
  *umask-cache*)

(define (change-and-cache-umask new-umask)
  (set-process-umask new-umask)
  (set! *umask-cache* (process-umask)))
```

This code uses another call to `umask` to feed the cache: this ensures proper error detection in case the specified value was not valid.

[4] One reviewer rightly noted that "A fluid friendly version of `fork` would have to be called `spoon`." The next version of scsh will feature this alias.

[5] The actual implementation initializes the cache when the system starts.

Next, there is code to access and modify the thread fluid:

```
(define (umask) (thread-fluid $umask))
(define (thread-set-umask! new-umask)
  (set-thread-fluid! $umask new-umask))
(define (let-umask new-umask thunk)
  (let-thread-fluid $umask new-umask thunk))
```

To change the umask scsh provides the following procedure:

```
(define (set-umask new-umask)
  (with-lock umask-lock
    (lambda ()
      (change-and-cache-umask new-umask)
      (thread-set-umask! (umask-cache)))))
```

A lock is required to synchronize the access to the cache. The following procedure aligns the resource with the thread fluid:

```
(define (align-umask!)
  (let ((thread-umask (umask)))
    (if (not (= thread-umask (umask-cache)))
        (change-and-cache-umask thread-umask))))
```

The test of the conditional compares the value of the cache with the thread fluid; the code in the consequence adjusts the resource in case of a mismatch. The following procedure aligns the umask and then calls its argument which is typically the actual system call wrapped in a thunk:

```
(define (with-umask-aligned* thunk)
  (obtain-lock umask-lock)
  (align-umask!)
  (with-handler
   (lambda (cond more)
     (release-lock umask-lock)
     (more))
   (lambda ()
     (let ((ret (thunk)))
       (release-lock umask-lock)
       ret))))
```

The lock prevents another thread from aligning the umask with its own value before the system call completes. As always with locks, some care must be taken to ensure the code releases the lock under unusual circumstances. The `thunk` argument usually contains only the call to the C function which in turn performs the system call so throwing out and back into its execution state by saved continuations is not an issue. However, in case the system call fails the C code will immediately raise an exception which allows execution to resume at a different point. To release the lock in this case the code above installs an exception handler which releases the lock and passes the exception along to the next handler: The `with-handler` procedure installs its first argument as a exception handler for the second argument. The handler releases the lock and calls the surrounding handler passed as argument `more` afterwards.

For the current working directory, caching is more involved as the `chdir` syscall itself reads the current working directory in case the given path is not absolute. Scsh circumvents this case by making the path absolute:

```
(define (change-and-cache-cwd new-cwd)
  (if (not (file-name-absolute? new-cwd))
      (process-chdir (assemble-path (cwd) new-cwd))
      (process-chdir new-cwd))
  (set! *cwd-cache* (process-cwd)))
```

Again, the cache is fed by consulting the kernel, this time to find out if the kernel has resolved any symbolic links. Setting and aligning the current working directory is completely analogous to the umask case:

```
(define (chdir cwd)
  (with-lock cwd-lock
    (lambda ()
      (change-and-cache-cwd cwd)
      (thread-set-cwd! (cwd-cache)))))

(define (align-cwd!)
  (let ((thread-cwd (cwd)))
    (if (not (string=? thread-cwd (cwd-cache)))
        (change-and-cache-cwd thread-cwd))))
```

The environment requires special treatment: First, there is a direct access to the resource itself. It is stored in the C variable `environ` of type `char **`. Programs normally access this vector through the functions `getenv`, `putenv` and `setenv` provided by the C library. Moreover, the only system call the environment influences is `exec*`. Therefore, scsh represents the environment by an association list in Scheme and turns it into an C array on `exec*` only. In this case scsh maintains an association of the Scheme list and the C array to allow the latter to be reused and automatically deleted. The caching procedure sets `environ**`:

```
(define (change-and-cache-env env)
  (environ**-set env)
  (set! *env-cache* env))
```

Reading the resource is only required on startup of the system; There the C vector is read into a Scheme list.

The last remaining process resource is the user identification.[6] In Unix, user identification comes in three flavors:

1. The *real user ID* encodes the identity of the owner of the process. The kernel copies the value from the parent when creating the process.
2. The *effective user ID* determines which files the process may access.
3. The *saved set-user ID* is set by `exec*` on start of the process and provides an alternative value for the effective user ID.

For changing these values, POSIX specifies the system call `setuid`. Unfortunately, its semantics depends on the value of the effective user ID: If the effective user ID is the ID of the super user, `setuid` changes *all* three values to the *same* but arbitrary ID. However, afterwards the effective user ID is no longer the ID of the super user and `setuid` cannot change the IDs any more. Automatic maintenance as described for the other resources is therefore not possible in general.

For unprivileged users things look slightly different: here `Setuid` sets *only* the effective user ID to either the real user ID or the saved set-user ID. The other IDs remain untouched. As the real user ID and the saved set-user ID may be different, both can act as a source for the effective user ID in turn. This behavior is desirable for applications which are started with a special saved set-user ID but want to exploit it only for certain tasks such as maintaining lock files.

[6] The following description translates literally to group identification. The presentation therefore does not consider group IDs further.

A multi-threaded application possibly wants to equip each thread with one of the two IDs. To support this, scsh provides thread-local effective user IDs.

The implementation of effective user IDs per thread is analogous to the umask case. A thread can read the effective user ID with `user-effective-uid` and set it with `set-user-effective-uid`. Scsh guards system calls operation on files with the `with-euid-aligned` macro. Depending on the platform, scsh uses one of the non-standard system calls `setreuid` or `seteuid` which change only the effective user ID to prevent the super user from unintentionally changing all three IDs.

Now the machinery is in place to properly define Scheme bindings for resource-accessing system calls:

```
(define (open-fdes path flags . maybe-mode)
  (with-cwd-aligned
   (with-umask-aligned
     (with-euid-aligned
       (with-egid-aligned
         (%open path
                flags
                (:optional maybe-mode #o666)))))))
```

The `%open` procedure is bound to the `open` system-call. It opens the file specified by `path` with umask, current working directory, effective user id and effective group id aligned. The `optional` macro returns the default mode `#o666` if the caller supplied no third argument to `open-fdes`.

6 `Fork` vs. Threads

The counterpart to `spawn`/`fork-thread` in the realm of Unix processes is called `fork`: it creates and starts a child process that is a copy of the parent process, distinguished from the parent by the return value of `fork`. Moreover, the childhas its own process ID, parent process ID and resource utilizations. The child process also gets copies of the parents file descriptors which, however, reference the same underlying objects.

In a user-level thread system, all threads are contained in the process. Consequently, the child process also runs duplicates of the threads of the parent process. Depending on the concrete thread system, this is desirable for the the system threads, such as those doing I/O cleanup, run finalizers, etc. However, this is usually wrong for the threads explicitly created by a running program. The most common use of `fork` in scsh programs is from the `&` and `run` forms that run external programs: in Unix, the only way to run another program is to replace the running process by it via `exec*`(3). Hence, `run` and `&` first fork, and the newly created child then replaces itself by the new program. Unfortunately, the delay between `fork` and `exec*` create a race condition: other threads of the running program can get scheduled *in the child*.

This race can have disastrous consequences: the Scheme Underground web server [17] starts a separate thread for each connection. Some connection requests require starting an external program such as a CGI script [3]. Now, consider a web server simultaneously serving two connections as shown in Figure 1. Thread #1 is busy serving a connection on the shown socket. Thread #2 forks in order to exec a CGI program. This creates an exact replica of the parent process, including the scheduler and all of its children threads which share access to the file descriptors of the parent process. It is now possible that the child scheduler schedules thread #1, thereby

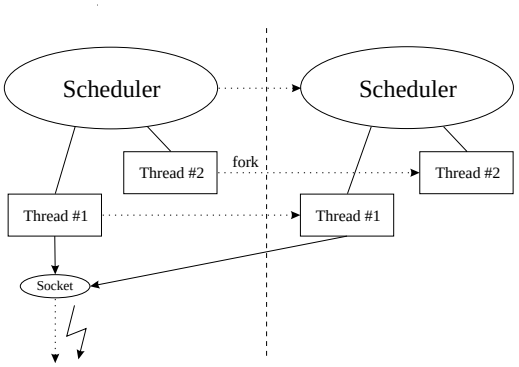

Figure 1. Interference between parent and child in a multi-threaded Internet server

interfering with the parent thread #1. This at least leads to mangling of the output.

This problem is well known in the realm of OS-level thread systems. Specifically, IEEE 1003.1-2001 [13] specifies that the child only runs the currently executing thread:

> A process shall be created with a single thread. If a multi-threaded process calls `fork()`, the new process shall contain a replica of the calling thread and its entire address space, possibly including the states of mutexes and other resources. [...]

„Forking the current thread“ is a more useful intuition for what `fork` should do. However, this notion as such is rife with ambiguity. (For example, what happens if the current thread is holding on to a mutex another thread is blocked on, and then, in the child, releases that mutex?) Moreover, `fork` has been notoriously difficult to implement correctly in Unix systems (see also Section 8).[7]

Fortunately, the implementation issues in the context of a nestable-engines-based thread system are entirely different ones from more traditional settings: Scsh solves the problem by providing a special scheduler which accepts an additional kind of event from its children threads called `narrow`. `Narrow` accepts a thunk as an argument, and causes the scheduler to spawn a new scheduler and suspend itself until the new scheduler terminates. The new scheduler starts off with a newly created single thread that runs the thunk.

The scsh scheduler sits beneath the root scheduler. Thus, the root scheduler can still perform the necessary housekeeping. Figure 2 shows the setup: The `narrow` call from thread #2 suspends to the scheduler, passing a thunk to run inside the narrowed thread under the new scheduler. The `fork` now happens in the narrowed thread, which also runs the thunk passed to fork. In the parent process, the narrowed thread terminates again which also returns operation to the original scheduler.

Thus, a simplified version of `fork` in scsh (the actual production code needs to perform more complex argument handling and avoid a subtle race condition) looks like this:

[7] In systems where threads are implemented as processes, the correct implementation of `fork` is trivial. However, then the implementation of `exec*`(2) becomes a problem because the new program must replace *all* threads of the old one. On the other hand, the implementation of `exec` is trivially correct in scsh.

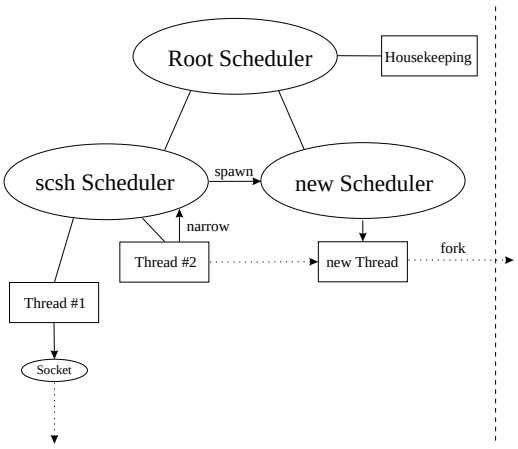

Figure 2. The `narrow` operation

```
(define (fork thunk)
  (let ((proc #f))
    (narrow
     (preserve-ports
      (preserve-thread-fluids
       (lambda ()
         (let ((pid (%%fork)))
           (if (zero? pid)
               (call-terminally thunk) ; Child
               (set! proc (new-child-proc pid))))))))
  proc))
```

`%%fork` is the pure POSIX *fork* system call. It returns 0 in the child, and a non-zero process ID in the parent. `Call-terminally` runs the thunk in an empty continuation to save space and guarantee that the child terminates once `thunk` returns.

Note that, just as with `thread-fork`, `fork` needs to preserve the thread fluids via `preserve-thread-fluids`. Moreover, `preserve-ports` preserves the regular fluids holding the `current-{input,output,error}` ports.

This implementation of `fork` avoids the various semantic pitfalls: All threads are still present after a `narrow`; they are merely children of a suspended scheduler. Therefore, if, for example, the current thread releases mutex locks other threads are blocked on, these threads are queued with their respective schedulers and can continue after the `narrow` completes. There are no restrictions on what the narrowed thread can do.

The implementation of `fork` actually shipped with scsh also allows duplicating all threads in the child. Consequently, through the use of nested schedulers `narrow` and `spawn`, the programmer has fine-grained control over the set of running threads.

Of course, the user-level program might create its own schedulers beneath the scsh scheduler. This, in general, requires that the new scheduler passes `narrow` events upwards in the scheduler tree to the scsh scheduler, which is trivial in the Scheme 48 thread system. On the other hand, it is possible that an application needs to handle `narrow` in a different way. The key observations of this work are that `narrow` is the appropriate mechanism for the feasible common cases, and that nestable schedulers provide a suitable implementation mechanism for providing a `fork` with well-defined behavior.

7 User-level threads and the C libraries

In addition to an interface to the Unix system calls, scsh also provides bindings for standard libraries. Two library facilities cause problems: DNS queries via `gethostbyname`/`gethostbyaddr` eventually block the process. The Syslog connections are an additional process resource. This section explains how scsh tackles these issues.

7.1 DNS queries

A user-level thread implementation must never call functions which might block the process and thereby stops all threads. All POSIX system calls can operate in non-blocking mode. Unfortunately, the same is not true for the standard C library: `gethostbyaddr` and `gethostbyname` turn host names into IP addresses and vice versa. These functions are indispensable for writing almost any kind of Internet server. They block until they receive an answer or timeout. Thus, the process calling `gethostby...` blocks for up to several minutes[8]. To prevent scsh from blocking, we have written a library for DNS queries directly in Scheme; it is part of the upcoming version of the Scheme Underground networking package[17].

7.2 Syslog

Another problem is the standard C library's interface to the system message logger: The `openlog` function opens a connection to the `syslogd` daemon. The `syslog` function sends the actual messages to the daemon. The syslog daemon processes the messages according to the parameters of `syslog` and the ones specified by the last `openlog` call. Calls to `openlog` may not nest.

Therefore, scsh treats the connection to the logger analogously to the process resources mentioned in Section 3: The interface to `openlog` virtualizes connections to the loggers as *syslog channels*. The syslog channel records all parameters given to `openlog`. Scsh stores the channel in a thread fluid and maintains a cache for the current channel. When another thread calls `syslog` and the cache differs from the thread's connection, scsh closes the connection to the syslog daemon using `closelog` and reconnects with the parameters obtained from the thread fluid. Thus, every thread has its own virtual connection to the syslog daemon.

7.3 FFI Coding Guidelines

Generally, threads complicate FFI issues because the language substrates on both sides of the FFI barrier are currently likely to be using different thread systems. The work on scsh indicates that coding guidelines should impose certain restrictions on foreign code called via the FFI:

- Foreign functions should not block indefinitely.
- Implicit state such as the process resources should be multiplexed via thread-local process resources.
- Non-reentrant foreign function APIs such as `syslog` should be virtualized to reentrant interfaces.

[8] Internet applications such as Netscape [12] and the Squid web cache [16] work around this problem by launching a second process to perform DNS queries. This allows the normal process to continue asynchronously or block on a pipe to the helper process using `select`.

8 Related work

The POSIX manpage [13] specifies that `fork` replicate only the calling thread. The manpage also mentions a proposed `forkall` function that replicates all running threads in the child. However, `forkall` was rejected for inclusion in the standard. The manpage lists a number of semantic issues for both `fork` and `forkall` that arise in the context of the Unix API. Specifically, a kernel-level thread system needs to deal with threads that are stuck in the kernel at the time of the `fork`. Reports of problems with handling or implementing `fork` with the proper semantics abound. Examples can be found in the FreeBSD commit logs and various Linux forums. Details vary greatly depending on implementation details of the operating system kernel and the thread system at hand.

The GNU adns C library [9] also provides an implementation of asynchronous DNS lookups.

9 Conclusion

Scsh combines user-level threads and the Unix API to yield a powerful tool for concurrent systems programming. The scsh API tries to maintain an analogy between threads and processes wherever possible. Specifically, threads see process resources as thread-local, and `fork` only "forks the current thread." The API issues involved are not new, but they occur in new forms in the context of Scheme 48's user-level thread system and scsh's support for the full POSIX API. The solutions have led to the design of the thread-fluid mechanism for managing thread-local dynamic bindings as well as of the `narrow` thread primitive which allows, together with nested threads, more fine-grained control over the set of running threads.

Acknowledgements

The implementation of thread-local process resources was developed in collaboration with Olin Shivers during the first author's visit at MIT. An email discussion with Richard Kelsey eventually led to the design of thread cells and thread fluids. He specifically proposed separating fluids from thread-local cells. Marcus Crestani implemented the DNS library for the Scheme Underground networking package. We also would like to thank all the users of scsh for constant feedback and valuable bug reports.

10 References

[1] Edoardo Biagioni, Ken Cline, Peter Lee, Chris Okasaki, and Chris Stone. Safe-for-space threads in Standard ML. *Higher-Order and Symbolic Computation*, 11(2):209–225, December 1998.

[2] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems*, 17(5):704–739, September 1995.

[3] CGI: Common gateway interface. `http://www.w3.org/CGI/`.

[4] R. Kent Dybvig and Robert Hieb. Engines from continuations. *Computer Languages*, 14(2):109–123, 1989.

[5] Marc Feeley. Parameters in Gambit-C. Personal communication, September 2001.

[6] Matthew Flatt. *PLT MzScheme: Language Manual*. Rice University, University of Utah, August 2000. Version 103.

[7] Christopher T. Haynes and Daniel P. Friedman. Engines build process abstractions. In *ACM Conference on Lisp and Functional Programming*, pages 18–24, 1984.

[8] Christopher T. Haynes and Daniel P. Friedman. Abstracting timed preemption with engines. *Computer Languages*, 12(2):109–121, 1987.

[9] Ian Jackson and Tony Finch. GNU adns. `http://www.chiark.greenend.org.uk/~ian/adns/`, 2000.

[10] Richard Kelsey and Jonathan Rees. *Scheme 48 Reference Manual*, 2002. Part of the Scheme 48 distribution at `http://www.s48.org/`.

[11] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1995.

[12] Netscape. Netscape browser central. `http://browsers.netscape.com/browsers/main.tmpl`, 2002.

[13] The Open Group Base Specifications Issue 6 IEEE Std 1003.1-2001. `http://www.opengroup.org/onlinepubs/007904975/`, 2001.

[14] Olin Shivers. A Scheme Shell. Technical Report TR-635, Massachusetts Institute of Technology, Laboratory for Computer Science, April 1994.

[15] Olin Shivers. Automatic management of operating system resources. In Mads Tofte, editor, *International Conference on Functional Programming*, pages 274–279, Amsterdam, The Netherlands, June 1997. ACM Press, New York.

[16] Team Squid. Squid web proxy cache. `http://www.squid-cache.org/`, 2002.

[17] The Scheme Underground networking package. `http://www.scsh.net/sunet/`.

Robust and Effective Transformation of Letrec

Oscar Waddell
owaddell@cs.indiana.edu

Dipanwita Sarkar
dsarkar@cs.indiana.edu

R. Kent Dybvig
dyb@cs.indiana.edu

Computer Science Department
Indiana University
Bloomington, IN 47408

ABSTRACT

A Scheme `letrec` expression is easily converted into more primitive constructs via a straightforward transformation given in the Revised[5] Report. This transformation, unfortunately, introduces assignments that can impede the generation of efficient code. This paper presents a more judicious transformation that preserves the semantics of the revised report transformation and also detects invalid references and assignments to left-hand-side variables, yet enables the compiler to generate efficient code. A variant of `letrec` that enforces left-to-right evaluation of bindings is also presented and shown to add virtually no overhead.

1. INTRODUCTION

Scheme's `letrec` permits the definition of mutually recursive procedures and, more generally, mutually recursive objects that contain procedures [2]. It is also a convenient intermediate-language representation for internal definitions and local modules [10]. When used for this purpose, the values bound by `letrec` are often a mix of procedures and nonprocedures.

A `letrec` expression has the form

`(letrec ([`x_1 e_1`] ... [`x_n e_n`]) `*body*`)`

where each x is a variable and each e is an arbitrary expression, often but not always a `lambda` expression. The Revised[5] Report on Scheme [2] defines `letrec` via the following transformation into more primitive constructs.

```
(letrec ([x1 e1] ... [xn en]) body)
 →  (let ([x1 undefined] ... [xn undefined])
      (let ([t1 e1] ... [tn en])
        (set! x1 t1)
        ...
        (set! xn tn))
      body)
```

where t_1 ... t_n are fresh temporaries.

Third Workshop on Scheme and Functional Programming. October 3, 2002, Pittsburgh, Pennsylvania, USA.

This transformation effectively defines the meaning of `letrec` operationally; a `letrec` expression (1) binds the variables x_1 ... x_n to new locations, each holding an "undefined" value, (2) evaluates the expressions e_1 ... e_n in some unspecified order, (3) assigns the variables to the resulting values, and (4) evaluates the body. The expressions e_1 ... e_n and `body` are all evaluated in an environment that contains the bindings of the variables, allowing the values to be mutually recursive.

The revised report imposes an important restriction on the use of `letrec`: it must be possible to evaluate each of the expressions e_1 ... e_n without evaluating a reference or assignment to any of the variables x_1 ... x_n. References and assignments to these variables may appear in the expressions, but they must not be evaluated until after control has entered the body of the `letrec`. We refer to this as the "`letrec` restriction." The revised report states that "it is an error" to violate this restriction. This means that the behavior is unspecified if the restriction is violated. While implementations are not required to signal such errors, doing so is desirable. The transformation given above does not directly detect violations of the `letrec` restriction. It does, however, imply a mechanism whereby violations can be detected, i.e., a check for the *undefined* value can be inserted before each reference or assignment to one of the left-hand-side variables occurring within a right-hand side.

The revised report transformation of `letrec` faithfully implements the semantics of `letrec` as described in the report, and it permits an implementation to detect violations of the `letrec` restriction. Yet, many of the assignments introduced by the transformation are unnecessary, and the obvious error detection mechanism inhibits copy propagation and inlining for `letrec`-bound variables.

This paper presents an alternative transformation of `letrec` that attempts to minimize the number of introduced assignments. It enables the compiler to generate efficient code while preserving the semantics of the revised report transformation. The alternative transformation is shown to eliminate most of the introduced assignments and to improve run time dramatically. The transformation incorporates a mechanism for detecting all violations of the `letrec` restriction that, in practice, has virtually zero overhead. The transformation assumes that an earlier pass of the compiler has recorded for each variable binding whether it has been referenced or assigned, and no other information is required.

This paper also investigates the implementation of a variant of `letrec`, which we call `letrec*`, that evaluates the right-hand sides from left to right and assigns each left-hand side immediately to the value of the right-hand side. It is often assumed that this would result in less efficient code; however, we show that this is not the case in practice. While there are valid software engineering reasons for leaving the evaluation order for `letrec` unspecified, `letrec*` would be a useful addition to the language and a reasonable intermediate representation for internal definitions, where left-to-right evaluation is often expected anyway.

The remainder of this paper is organized as follows. Section 2 describes our transformation in three stages, starting with a basic version, adding an assimilation mechanism for nested bindings, and adding valid checks for references and assignments to left-hand-side variables. Section 3 introduces the `letrec*` form and describes its implementation. Section 4 presents an analysis of the effectiveness of the various transformations. Section 5 describes related work. Finally, Section 6 summarizes the paper and presents our conclusions.

2. THE TRANSFORMATION

The transformation of `letrec` is developed in three stages. Section 2.1 describes the basic transformation. Section 2.2 describes a more elaborate transformation that assimilates `let` and `letrec` bindings that are nested on the right-hand side of a `letrec` expression. Section 2.3 shows how to efficiently detect violations of the `letrec` restriction.

The transformation expects that bound variables in the input program are uniquely named. It also assumes that an earlier pass of the compiler has recorded information about references and assignments of the bound variables. In our implementation, these conditions are met by running input programs through the `syntax-case` macro expander [1]. If this were not the case, a simple flow-insensitive pass to perform alpha conversion and record reference and assignment information could be run prior to the transformation algorithm.

The transformation is implemented in two passes. The first performs the transformation proper, and the second introduces the code that detects violations of the `letrec` restriction.

2.1 Basic transformation

Each `letrec` expression `(letrec ([`x e`] ...)` *body*`)` in an input program is converted as follows.

1. The expressions e `...` and *body* are converted to produce e' `...` and *body*′.
2. The bindings `[`x e'`] ...` are partitioned into several sets:
   `[`x_s e_s`] ...` *simple*
   `[`x_l e_l`] ...` *lambda*
   `[`x_u e_u`] ...` *unreferenced*
   `[`x_c e_c`] ...` *complex*
3. A set of nested `let` and `fix` expressions is formed from the partitioned bindings:

   ```
   (let ([x_s e_s] ... [x_c (void)] ...)
     (fix ([x_l e_l] ...)
       e_u ...
       (let ([x_t e_c] ...)
         (set! x_c x_t)
         ...)
       body′))
   ```

   where x_t `...` is a set of fresh temporaries, one per x_c. The innermost let is produced only if `[`x_c e_c`] ...` is nonempty. The expressions e_u `...` are retained for their effects.
4. Because the bindings for unreferenced `letrec`-bound variables are dropped, all assignments to unreferenced variables are also dropped.

During the partitioning phase, a binding `[`x e'`]` is considered

simple if x is referenced but not assigned and e' is a simple expression;

lambda if x is referenced but not assigned and e' is a lambda expression;

unreferenced if no references to x appear in the program;

complex if it does not fall into any of the other categories.

A simple expression contains no occurrences of the variables bound by the `letrec` expression and must not be able to obtain its continuation via `call/cc`, either directly or indirectly. The former restriction is necessary because simple expressions are placed outside the scope of the bound variables. Without the latter restriction, it would be possible to detect the fact that the bindings are created after the evaluation of a simple right-hand-side expression rather than before. To enforce the latter restriction, our implementation simply rules out all procedure calls except those to certain primitives (not including `call/cc`).

A `fix` expression is a variant of `letrec` that binds only unassigned variables to `lambda` expressions. It represents the subset of `letrec` expressions that can be handled easily by later passes of a compiler. In particular, no assignments through external variables are necessary to implement mutually recursive procedures bound by `fix`. Instead, the closures produced by a `fix` expression can be block allocated and "wired" directly together. This leaves the `fix`-bound variables unassigned for the duration, thus simplifying optimizations such as inlining and loop recognition. `fix` is identical to the `labels` operator handled by Steele's Rabbit compiler [9] and the `Y` operator of Kranz's Orbit compiler [4, 3] and Rozas' Liar compiler [7, 8].

The output expression includes calls to `void`, a primitive that evaluates to some "unspecified" value. It may be defined as follows.

```
(define void (lambda () (if #f #f)))
```

We do not use a special "undefined" value; instead, we use a different mechanism for detecting violations of the `letrec` restriction, as described in Section 2.3.

An unreferenced binding [x e'] may be dropped if e' is simple or a `lambda` expression, although the code generated is the same if a later pass eliminates such expressions when they are used only for effect, as is the case in our compiler.

2.2 Assimilating nested binding forms

When a `letrec` right-hand side is a `let` or `letrec` expression, the partitioning described above treats it as *complex*. For example,

```
(letrec ([f (letrec ([g (let ([x 5])
                          (lambda () ...))])
              (lambda () ... g ...))])
  f)
```

is translated into

```
(let ([f (void)])
  (let ([ft (let ([g (void)])
              (let ([gt (let ([x 5])
                          (lambda () ...))])
                (set! g gt))
              (lambda () ... g ...))])
    (set! f ft))
  f)
```

This is unfortunate, since it penalizes programmers who use nested `let` and `letrec` expressions in this manner to express scoping relationships more tightly.

We'd prefer a translation into the following equivalent expression.

```
(let ([x 5])
  (fix ([f (lambda () ... g ...)]
        [g (lambda () ...)])
    f))
```

Therefore, the actual partitioning used is a bit more complicated. When a binding [x e'] fits immediately into one of the first three categories, the rules above suffice. The exception to these rules occurs when x is unassigned and e' is a `let` or `letrec` binding, in which case the transformer attempts to fold the nested bindings into the partitioned sets, which leads to fewer introduced assignments and more direct call optimizations in later passes of the compiler.

When e' is a `fix` expression (`fix` ([$\overline{x_l}$ $\overline{e_l}$] ...) $\overline{body}$), the bindings [$\overline{x_l}$ $\overline{e_l}$] ... are simply added to the *lambda* partition and the binding [x $\overline{body}$] is added to the set of bindings to be partitioned.

Essentially, this transformation treats the nested bindings as if they had originally appeared in the enclosing `letrec`. For example,

(`letrec` ([`f` e_f] [`g` (`fix` ([`a` e_a]) e_g)] [`h` e_h]) *body*)

is treated as

(`letrec` ([`f` e_f] [`g` e_g] [`a` e_a] [`h` e_h]) *body*)

When e' is a `let` expression (`let` ([$\overline{x}$ $\overline{e'}$] ...) $\overline{body}$) and the set of bindings [$\overline{x}$ $\overline{e'}$] ... can be fully partitioned into a set of *simple* bindings [$\overline{x_s}$ $\overline{e_s}$] ... and a set of *lambda* bindings [$\overline{x_l}$ $\overline{e_l}$] ..., we add [$\overline{x_s}$ $\overline{e_s}$] ... to the *simple* partition, [$\overline{x_l}$ $\overline{e_l}$] ... to the *lambda* partition, and [$\overline{x}$ $\overline{body}$] to the set of bindings to be partitioned.

For example, when e_a is a `lambda` or simple expression,

(`letrec` ([`f` e_f] [`g` (`let` ([`a` e_a]) e_g)] [`h` e_h]) *body*)

is treated as

(`letrec` ([`f` e_f] [`g` e_g] [`a` e_a] [`h` e_h]) *body*)

If during this process we encounter a binding [$\overline{x}$ $\overline{e}$] where $\overline{x}$ is unassigned and $\overline{e}$ is a `let` or `fix` expression, we simply fold the bindings in and continue.

While Scheme allows the right-hand sides of a binding construct to be evaluated in any order, the order used must not involve (detectable) interleaving of evaluation. For possibly assimilated bindings only, the definition of *simple* must therefore be modified to preclude effects. Otherwise, the effects caused by the bindings and body of an assimilated `let` could be separated, producing a detectable interleaving of the assimilated `let` with the other expressions bound by the outer `letrec`.

One situation not handled by this transformation is the following, in which a local binding is used to hold a counter or other similar piece of state.

```
(letrec ([f (let ([n 0])
              (lambda ()
                (set! n (+ n 1))
                n))])
  body)
```

We are prevented from assimilating cases like this because it may be possible to detect the separation of the creation of the (mutable) binding for `n` from the evaluation of the body of the nested `let` by invoking a continuation created in another of the `letrec` bindings that causes the body of the nested `let` to be evaluated multiple times. The separation cannot be detected in the given example, however, since the body of the nested `let` is a `lambda` expression, and assimilated bindings of `lambda` expressions are evaluated only once.

Because it is desirable not to penalize such uses of local state, we add an additional case to handle this situation. When e' is a `let` expression (`let` ([$\overline{x}$ $\overline{e}$] ...) $\overline{body}$) and the set of bindings [$\overline{x}$ $\overline{e}$] ... can be fully partitioned into a set of *simple* bindings [$\overline{x_s}$ $\overline{e_s}$] ... and a set of *lambda* bindings [$\overline{x_l}$ $\overline{e_l}$] ..., except that one or more of the variables $\overline{x_s}$ ... is assigned, and $\overline{body}$ is a `lambda` expression, we add [$\overline{x_s}$ $\overline{e_s}$] ... to the *simple* partition, [$\overline{x_l}$ $\overline{e_l}$] ... to the *lambda* partition, and [x $\overline{body}$] to the set of bindings to be partitioned.

For example, when e_a is a `lambda` or simple expression, a is assigned, and e_g is a `lambda` expression,

(`letrec` ([`f` e_f] [`g` (`let` ([`a` e_a]) e_g)] [`h` e_h]) *body*)

is treated as

(`letrec` ([`f` e_f] [`g` e_g] [`a` e_a] [`h` e_h]) *body*)

If during this process we encounter a binding [$\overline{x}$ $\overline{e}$] where $\overline{x}$ is unassigned and $\overline{e}$ is a `let` or `fix` expression, or if we find that the body is a `let` or `fix` expression, we simply fold the bindings in and continue.

The `let` and `fix` expressions produced by recursive transformation of a `letrec` expression can always be assimilated if they have no complex bindings. Thus, the assimilation of `let` and `fix` expressions in the intermediate language effectively implements the assimilation of `letrec` expressions in the source language.

2.3 Valid checks

According to the Revised[5] Report, it must be possible to evaluate each of the expressions e_1 ... e_n in

`(letrec ([`x_1 e_1`] ... [`x_n e_n`]) `*body*`)`

without evaluating a reference or assignment to any of the variables x_1 ... x_n. This is the "`letrec` restriction" first mentioned in Section 1.

The revised report states that "it is an error" to violate this restriction. Implementations are not required to signal such errors; the behavior is left unspecified. An implementation may instead assign a meaning to the erroneous program. Older versions of our system "corrected" erroneous programs like the following.

```
(letrec ([x 1] [y (+ x 1)]) (list x y))  ⇒  (1 2)
(letrec ([y (+ x 1)] [x 1]) (list x y))  ⇒  (1 2)
```

We never liked this behavior, which fell out of an earlier version of the partitioning algorithm.

We believe it is better for an implementation to detect and report errors rather than to give meaning to technically meaningless programs. Reporting these errors also helps users create more portable programs. Fortunately, it turns out that these errors can be detected with practially no overhead, as we describe in this section.

It is possible to detect violations of the `letrec` restriction by binding each left-hand-side variable initially to a special "undefined" value and checking for this value at each reference and assignment to the variable within the right-hand-side expressions. This approach introduces many more checks than are actually necessary. More importantly, it prevents us from performing the transformations described in Sections 2.1 and 2.2 and, as a result, may inhibit later passes from performing various optimizations such as inlining and copy propagation.

It is possible to analyze the right-hand sides to determine the set of variables referenced or to perform an interprocedural flow analysis to determine the set of variables that might be undefined when referenced or assigned, by monitoring the flow of the undefined values. With this information, we could perform the transformations described in Sections 2.1 and 2.2 for all but those variables that might be undefined when referenced or assigned.

We use a different approach that never inhibits our transformations and thus does not inhibit optimization of `letrec`-bound variables merely because they may be undefined when referenced or assigned. Our approach is based on two observations: (1) a separate boolean variable may be used to indicate the validity of a `letrec` variable, and (2) we need just one such variable per `letrec`; if evaluating a reference or assignment to one of the left-hand-side variables is invalid at a given point, evaluating a reference or assignment to any of those variables is invalid. With a separate valid flag, the transformation algorithm can do as it pleases with the original bindings.

This flag is introduced as a binding of a fresh variable, `valid?`, wrapped around the code that evaluates the *unreferenced* and *complex* expressions. If a `letrec` has no *unreferenced* or *complex* bindings, no valid flag need be introduced. This flag is checked at each point where a valid check is deemed to be necessary. It is set initially to false, meaning that references to left-hand-side expressions are not allowed, and changed to true once control enters the body of the `letrec`.

```
(let ([x_s e_s] ... [x_c (void)] ...)
  (fix ([x_l e_l] ...)
    (let ([valid? #f])
      e_u ...
      (let ([x_t e_c] ...)
        (set! x_c x_t)
        ...)
      (set! valid? #t))
    body′))
```

In a naive implementation, valid checks would be inserted at each reference and assignment to one of the left-hand-side variables within the *unreferenced* and *complex* expressions. A valid check simply tests `valid?` and signals an error if `valid?` is false. For each valid check for a variable `x`, the valid check appears as follows.

```
(unless valid? (error 'x "undefined"))
```

No checks need to be inserted in the body of the `letrec`, since the bindings are necessarily valid once control enters the body. No checks are required within the right-hand sides of *lambda* bindings, since control cannot enter the body of one of these `lambdas` except by way of a reference to the corresponding left-hand-side variable. *Simple* bindings contain no references to the left-hand-side variables.

We can do even better than to limit the valid checks to the right-hand sides of *unreferenced* and *complex* bindings. To do so, we introduce the notion of *protected* and *unprotected* references. A reference (or assignment) to a variable is protected if it is contained within a `lambda` expression that cannot be evaluated and invoked during the evaluation of an expression. Otherwise, it is unprotected.

Valid checks are introduced during a second pass of the transformation algorithm. This pass uses a simple top-down recursive descent algorithm. While processing the *unreferenced* and *complex* right-hand sides of a `letrec`, the left-hand-side variables of the `letrec` are considered to be in one of three states: *protected*, *protectable*, or *unprotected*. A variable is protectable if references and assignments found within a `lambda` expression are safe, i.e., if the `lambda` expression cannot be evaluated and invoked before control enters the body of the `letrec`. Each variable starts out in the protectable state when processing of the right-hand-side expression begins.

Upon entry into a `lambda` expression, all protectable variables are moved into the protected state, since they can-

not possibly require valid checks. Upon entry into an unsafe context, i.e., one that might result in the evaluation and invocation of a `lambda` expression, the protectable variables are moved into the unprotected state. This occurs, for example, while processing the arguments to an unknown procedure, since that procedure might invoke the procedure resulting from a `lambda` expression appearing in one of the arguments.

For each variable reference and assignment, a valid check is inserted for the protectable and unprotected variables but not for the protected variables.

This handles well situations such as

```
(letrec ([x 0]
         [f (cons (lambda () x)
                  (lambda (v) (set! x v)))])
  body)
```

in which `f` is a sort of locative [6] for `x`. Since `cons` does not invoke its arguments, the references appearing within the `lambda` expressions are protected.

It doesn't handle situations such as the following.

```
(letrec ([x 0]
         [f (let ([g (lambda () x)])
              (lambda () (g)))])
  body)
```

In general, we must treat the right-hand side of a `let` expression as unsafe, since the left-hand-side variable may be used to invoke procedures created by the right-hand-side expression. In this case, however, the body of the `let` is a `lambda` expression, so there is no problem. To handle this situation, we also record for each `let`- and `fix`-bound variable whether it is protectable or unprotected and treat the corresponding right-hand side as an unsafe or safe context depending upon whether the variable is referenced or not. For `fix` this involves a sort of demand-driven processing, starting with the body of the `fix` and proceeding with the processing of any unsafe right-hand sides.

The original `letrec` expressions no longer exist by the time the second pass runs, so the first pass must leave behind sufficient information to allow the second pass to know which are the original `letrec`-bound variables and which expressions may require the insertion of valid checks. The actual output of the first pass is therefore as follows

```
(let ([x_s  e_s] ... [x_c (void)] ...)
  (fix ([x_l  e_l] ...)
    (bind-valid-flag (x ...)
      e_u ...
      (let ([x_t  e_c] ...)
        (valid-set! x_c  x_t)
        ...))
    body/))
```

where x `...` is the original list of `letrec`-bound variables. The `bind-valid-flag` expression expands into a `let` expression binding the variable `valid?` if any valid checks were inserted, otherwise it expands into the code in its body. It also inserts the assignment to set the valid flag true at the end of its body if the valid flag is introduced. The `valid-set!` expression is used in place of `set!` for the introduced assignments to the *complex* variables; this tells the second pass that this is already known to be valid so that no valid check is inserted for the assignment.

3. FIXED EVALUATION ORDER

The Revised[5] Report translation of `letrec` is designed so that the right-hand-side expressions are all evaluated before the assignments to the left-hand-side variables are performed. The transformation for `letrec` described in the preceding section loosens this structure, but in such a manner that cannot be detected, because an error is signaled for any program that prematurely references one of the left-hand-side variables and because the lifted bindings are immutable and cannot be (detectably) reset by a continuation invocation.

From a software engineering perspective, the unspecified order of evaluation is valuable because it allows the programmer to express lack of concern for the order of evaluation. That is, when the order of evaluation of two expressions is unspecified, the programmer is, in effect, saying that neither counts on the other being done first. From an implementation standpoint, the freedom to determine evaluation order may allow the compiler to generate more efficient code.

It is sometimes convenient, however, for the values of a set of `letrec` bindings to be established in a particular order. This seems to occur most often in the translation of internal definitions into `letrec`. For example, one might wish to define a procedure and use it to produce the value of a variable defined further down in a sequence of definitions.

```
(define f (lambda ...))
(define a (f ...))
```

One can nest binding contours to order bindings, but this is often inconvenient and prevents the sequenced bindings from being mutually recursive. It is therefore interesting to consider a variant of `letrec` that performs its bindings in a left-to-right fashion. Scheme provides a variant of `let`, called `let*`, that sequences evaluation of `let` bindings; we therefore call our version of `letrec` that sequences `letrec` bindings `letrec*`. The analogy to `let*` is imperfect, since `let*` also nests scopes whereas `letrec*` maintains the mutual recursive scoping of `letrec`.

`letrec*` can be transformed into more primitive constructs in a manner similar to `letrec` using a variant of the Revised[5] Report transformation of `letrec`.

```
(letrec* ([x_1  e_1] ... [x_n  e_n]) body)
 →  (let ([x_1 undefined] ... [x_n undefined])
      (set! x_1  e_1)
      ...
      (set! x_n  e_n)
      body)
```

This transformation is actually simpler, in that it does not include the inner `let` binding a set of temporaries to the right-hand-side expressions. This transformation would be incorrect for `letrec`, since the assignments are not all in the continuation of each right-hand-side expression, as in the revised report transformation. Thus, `call/cc` could be used to expose the difference between the two transformations.

The basic transformation given in Section 2.1 is also easily modified to implement the semantics of `letrec*`. As before, the expressions *e* ... and *body* are converted to produce *e′* ... and *body′*, and the bindings are partitioned into *simple*, *lambda*, *unreferenced*, and *complex* sets. The difference comes in the structure of the output code. If there are no *unreferenced* bindings, the output is as follows

```
(let ([x_s e_s] ... [x_c (void)] ...)
  (fix ([x_l e_l] ...)
    (set! x_c e_c)
    ...
    body′))
```

where the assignments to x_c are ordered as the bindings appeared in the original input.

If there are *unreferenced* bindings, the right-hand sides of these bindings are retained, for effect only, among the assignments to the *complex* variables in the appropriate order.

The more elaborate partitioning of `letrec` expressions to implement assimilation of nested bindings as described in Section 2.2 is compatible with the transformation above, so the implementation of `letrec*` does not inhibit assimilation.

On the other hand, a substantial change to the introduction of valid flags is necessary to handle the different semantics of `letrec*`. This change is to introduce one valid flag for each *unreferenced* and *complex* right-hand side, in contrast to one per `letrec` expression. The valid flag for a given expression represents the validity of references and assignments to the corresponding variable and all subsequent variables bound by the `letrec`. This may result in the introduction of more valid flags but should not result in the introduction of any additional valid checks. Due to the nature of `letrec*`, in fact, there will likely be fewer valid checks and possibly fewer actual valid-flag bindings.

As with `letrec`, the first pass of the transformation algorithm inserts `bind-valid-flag` expressions to tell the second pass where to insert valid flags and checks. If there are no unreferenced bindings, the output is as follows

```
(let ([x_s e_s] ... [x_c (void)] ...)
  (fix ([x_l e_l] ...)
    (set! x_c
      (bind-valid-flag (x_c+ ...)
        e_c))
    ...
    body′))
```

where x_c+ ... represents the sublist of original left-hand-side variables from x_c on. If there are *unreferenced* bindings, the right-hand sides are inserted into the code in the proper sequence, each wrapped in a `bind-valid-flag` expression that lists all variables from the next referenced variable on.

The second pass operates as before: no changes are needed to support `letrec*`.

4. RESULTS

We have implemented the complete algorithm described in Section 2 and incorporated it as two new passes in the Chez Scheme compiler. The first pass performs the transformations described in Sections 2.1 and 2.2, and the second pass inserts the valid checks described in Section 2.3. We have also added a `letrec*` form that guarantees left-to-right evaluation as described in Section 3 and a compile-time parameter that allows internal definitions (including those within modules) to be expanded into `letrec*` rather than `letrec`.

We measured the performance of the benchmark programs using several transformations:

- the standard Revised^5 Report (R^{5}RS) transformation;
- a modified R^{5}RS transformation (which we call "easy") that treats "pure" (`lambda` only) `letrec` expressions as `fix` expressions and reverts to the standard transformation for the others;
- versions of R^{5}RS and "easy" with naive valid checks;
- our transformation with and without assimilation and with and without valid checks; and
- our transformation with assimilation and valid checks, treating all `letrec` expressions as `letrec*` expressions.

Not surprisingly, the benchmark programs still run in the system that treats `letrec` as `letrec*`, since none contain code that detects the failure of that system to be faithful to the Revised^5 Report transformation. (Some of the tests in our test suite did fail, but only because they were there to keep our compiler honest in this regard.)

We compare these systems along several dimensions: run time, compile time, code size, number of introduced assignments, number of valid checks, and numbers of bindings classified as *lambda*, *complex*, *simple*, and *unreferenced*. Run times were determined by averaging three runs for each benchmark; programs were configured so that each run required at least two seconds. Code size was determined by recording the size of the actual code objects written to compiled files. Compile times were recorded for a single compilation of each benchmark, with the exception of the compiler bootstrapping benchmark (chezscheme), where three such runs were averaged. With the exception of `chezscheme`, `similix`, and `texer`, each benchmark was placed within a `module` form, converting top-level definitions to internal definitions. A few programs that relied on left-to-right evaluation of top-level definitions were edited so that they could run successfully in all of the systems.

The results are given in Tables 1–4. Programs in these tables are listed in sorted order, with larger programs (in terms of object code) after smaller ones. The run-time results show that the transformation is successful in reducing run-time overhead in many cases and never increases overhead, even with valid checks enabled. Using the "easy" transformation to catch pure `letrec` expressions is also effective, but our transformation is even more effective, with noticible improvements on several benchmarks, including `lattice-jw`, `ray`, `maze`, and `conform`.

Using our algorithm, run times are almost identical with or without valid checks, so strict enforcement of the `letrec`

restriction is achieved with practically no overhead. Most of the benchmarks require no valid flags and few require a substantial number of valid checks. In contrast, naive valid checks significantly reduce the performance of the R^{5}RS and "easy" transformations in some cases.

For our compiler, the most substantial program in our test suite, assimilating nested bindings allows the transformation to decrease the number of introduced assignments by 17%. Moreover, this allows the transformation to eliminate all of the valid checks that would otherwise be inserted. Assimilation of nested bindings does not seem to benefit run times, however. This is somewhat disappointing, but may simply indicate that few of the benchmarks try to express scoping relationships more tightly, perhaps even because of a fear that the resulting code would not be as efficient. We believe it is an important optimization, nevertheless, as one of many "bullets in [the compiler's] gun" [5] that are not generally applicable but are very useful in certain circumstances.

Compile time increases are modest for our algorithm, with or without valid checks and assimilation. In many cases, the compile times are less, even though more effort is clearly expended in the new passes than is required to do the R^{5}RS transformation. This is because our transformation enables more optimizations by later passes, leading to smaller code and an overall reduction in compile times.

The numbers for `letrec*` indicate that there is no overhead in practice for fixing the order of evaluation, even though our compiler reorders expressions when possible to improve the generated code. This is likely due in part to the relatively few cases where our translation of `letrec*` actually introduces constraints on the evaluation order. In addition, almost no valid flags and checks are required for `letrec*`. So while the implementation of `letrec*` may require more valid flags in principle, it requires fewer in practice, since the fixed evaluation order eliminates the need for most valid checks and the flags used to support them.

As shown in Table 1, the "easy" algorithm, which is attractive for its simplicity, often introduces many more assignments than are necessary, since not all `letrec` bindings are `lambda` expressions. Naively enforcing the `letrec` restriction also introduces far more valid checks than necessary, even when pure `letrec` expressions are recognized.

Our algorithm identifies "simple" bindings in many of the benchmarks and avoids introducing assignments for these. Moreover, it avoids introducing assignments for pure `lambda` bindings that happen to be bound by the same `letrec` that binds a simple binding. In several cases, assimilating nested `let` and `letrec` bindings allows the algorithm to assign more of the bindings to the *lambda* or *simple* partitions.

5. RELATED WORK

Much has been written about generating efficient code for ideal recursive binding forms, like our `fix` construct or the `Y` combinator, that bind only `lambda` expressions. Yet virtually nothing has been written explaining how to cope with the reality of arbitrary `letrec` expressions, e.g., by transforming them into one of these ideal forms. Moreover, nothing has been written describing efficient strategies for detecting violations of the "`letrec` restriction."

Steele [9] developed strategies for generating good code for mutually recursive procedures bound by a `labels` form that is essentially our `fix` construct. Because `labels` forms are present in the input language handled by his compiler, he does not describe the translation of general `letrec` expressions into `labels`.

Kranz [4, 3] also describes techniques for generating efficient code for mutually recursive procedures expressed in terms of the `Y` operator. He describes a macro transformation of `letrec` that introduces assignments for any right-hand side that is not a `lambda` expression and uses `Y` to handle those that are `lambda` expressions. This transformation introduces unnecessary assignments for bindings that our algorithm would deem *simple*. His transformation does not attempt to assimilate nested binding constructs. The `Y` operator is a primitive construct recognized by his compiler, much as `fix` is recognized by our compiler.

Rozas [7, 8] shows how to generate good code for mutually recursive procedures expressed in terms of `Y` without recognizing `Y` as a primitive construct, that is, with `Y` itself expressed at the source level. He does not discuss the process of converting `letrec` into this form.

6. CONCLUSION

We have presented an algorithm for transforming `letrec` expressions into a form that enables the generation of efficient code while preserving the semantics of the `letrec` transformation given in the Revised^5 Report on Scheme [2]. The transformation avoids many of the assignments produced by the Revised^5 Report transformation by converting many of the `letrec` bindings into simple `let` bindings or into a "pure" form of `letrec`, called `fix`, that binds only unassigned variables to `lambda` expressions. `fix` expressions are the basis for several optimizations, including block allocation and internal wiring of closures. We have shown the algorithm to be effective at reducing the number of introduced assignments and improving run time with little compile-time overhead.

The algorithm also inserts "valid checks" to implement the `letrec` restriction that no reference or assignment to a left-hand-side variable can be evaluated in the process of evaluating the right-hand-side expressions. It inserts few checks in practice and adds practically no overhead to the evaluation of programs that use `letrec`. More importantly, it does not inhibit the optimizations performed by subsequent passes. Most Scheme implementations currently omit such checks, but this paper shows that the checks can be performed even in compilers that are geared toward high-performance applications.

We have also introduced a variant of `letrec`, called `letrec*`, that establishes the values of each variable in sequence from left-to-right. `letrec*` may be implemented with a small modification to the algorithm for implementing `letrec`. We have shown that, in practice, our implementation of `letrec*` is as efficient as `letrec`, even though later passes of our compiler take advantage of the ability to reorder right-hand-side expressions. This is presumably due to the relatively few

cases where our translation of `letrec*` actually introduces constraints on the evaluation order, but in any case, debunks the commonly held notion that fixing the order of evaluation hampers production of efficient code for `letrec`.

While treating `letrec` expressions as `letrec*` clearly violates the Revised[5] Report semantics for `letrec`, we wonder if future versions of the standard shouldn't require that internal definitions be treated as `letrec*` rather than `letrec`. Left-to-right evaluation order of definitions is often what programmers expect and would make the semantics of internal definitions more consistent with external definitions. We have shown that there would be no significant performance penalty for this in practice.

7. REFERENCES

[1] Dybvig, R. K., Hieb, R., and Bruggeman, C. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation 5*, 4 (1993), 295–326.

[2] Kelsey, R., Clinger, W., and Rees, J. A. Revised[5] report on the algorithmic language Scheme. *SIGPLAN Notices 33*, 9 (1998), 26–76.

[3] Kranz, D. A. *Orbit, An Optimizing Compiler for Scheme*. PhD thesis, Yale University, May 1988.

[4] Kranz, D. A., Kelsey, R., Rees, J. A., Hudak, P., Philbin, J., and Adams, N. I. Orbit: an optimizing compiler for Scheme. *SIGPLAN Notices, ACM Symposium on Compiler Construction 21*, 7 (1986), 219–233.

[5] Peyton Jones, S. L., and Santos, A. A transformation-based optimiser for Haskell. *Science of Computer Programming 32*, 1–3 (1998), 3–47.

[6] Rees, J. A., Adams, N. I., and Meehan, J. R. *The T Manual*. Yale University, New Haven, Connecticut, USA, 1984. Fourth edition.

[7] Rozas, G. J. Liar, an Algol-like compiler for Scheme. S. B. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Jan. 1984.

[8] Rozas, G. J. Taming the Y operator. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming* (San Francisco, USA, June 1992), pp. 226–234.

[9] Steele Jr., G. L. Rabbit: a compiler for Scheme. MIT AI Memo 474, Massachusetts Institute of Technology, Cambridge, Mass., May 1978.

[10] Waddell, O., and Dybvig, R. K. Extending the scope of syntactic abstraction. In *Conference Record of the Twenty Sixth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1999), pp. 203–213.

	Introduced assignments					Valid checks					# bindings in each partition									
											easy		A				N			
	R^5RS	easy	A	N	S	R^5RS	easy	A	N	S	λ	c	λ	c	s	u	λ	c	s	u
fxtak	2	–	–	–	–	5	–	–	–	–	2	–	2	–	–	–	2	–	–	–
tak	2	–	–	–	–	5	–	–	–	–	2	–	2	–	–	–	2	–	–	–
div-iter	10	–	–	–	–	14	–	–	–	–	10	–	9	–	–	1	9	–	–	1
cpstak	3	–	–	–	–	5	–	–	–	–	3	–	3	–	–	–	3	–	–	–
takl	7	3	3	3	3	8	–	–	–	–	4	3	4	3	–	–	4	3	–	–
ctak	3	–	–	–	–	6	–	–	–	–	3	–	3	–	–	–	3	–	–	–
mbrot	8	–	–	–	–	7	–	–	–	–	8	–	8	–	–	–	8	–	–	–
deriv	4	–	–	–	–	8	–	–	–	–	4	–	4	–	–	–	4	–	–	–
destruct	9	–	–	–	–	8	–	–	–	–	9	–	9	–	–	–	9	–	–	–
fxtriang	13	7	4	4	4	6	–	–	–	–	6	7	6	4	3	–	6	4	3	–
fft-f	8	–	–	–	–	7	–	–	–	–	8	–	8	–	–	–	8	–	–	–
fft-d	11	–	–	–	–	12	–	–	–	–	11	–	11	–	–	–	11	–	–	–
dderiv	8	–	–	–	–	4	–	–	–	–	8	–	8	–	–	–	8	–	–	–
triang	13	7	4	4	4	6	–	–	–	–	6	7	6	4	3	–	6	4	3	–
lattice	22	14	–	1	–	37	29	–	–	–	8	14	21	–	2	–	19	1	2	–
boyer	25	23	2	2	2	50	48	–	–	–	2	23	22	2	–	1	22	2	–	1
boyer-jw	23	22	4	4	4	67	66	–	–	–	1	22	19	4	–	–	19	4	–	–
browse	20	8	1	1	1	33	21	–	–	–	12	8	19	1	–	–	19	1	–	–
traverse	47	38	5	5	5	69	60	–	–	–	9	38	39	5	–	3	39	5	–	3
lattice-jw	22	10	–	1	–	33	19	–	–	–	12	10	23	–	–	–	21	1	–	–
fft-g	11	4	–	–	–	9	2	–	–	–	7	4	8	–	3	–	8	–	3	–
ray	34	27	2	2	2	92	84	–	–	–	7	27	32	2	–	–	32	2	–	–
fxpuzzle	34	11	11	11	11	20	–	–	–	–	23	11	22	11	–	1	22	11	–	1
graphs	32	–	–	–	–	38	–	–	–	–	32	–	28	–	–	4	28	–	–	4
tcheck	35	32	3	3	3	84	79	–	–	–	3	32	22	3	12	–	22	3	10	–
simplex	32	1	–	–	–	67	–	–	–	–	31	1	31	–	1	–	31	–	1	–
graphs-jw	20	–	–	–	–	24	–	–	–	–	20	–	20	–	–	–	20	–	–	–
maze	83	62	1	1	1	183	161	–	–	–	21	62	68	1	3	11	68	1	3	11
maze-jw	85	–	–	–	–	37	–	–	–	–	85	–	74	–	–	11	74	–	–	11
puzzle	34	11	11	11	11	20	–	–	–	–	23	11	22	11	–	1	22	11	–	1
earley	75	–	–	–	–	117	–	–	–	–	75	–	73	–	–	2	73	–	–	2
splay	13	–	–	–	–	17	–	–	–	–	13	–	13	–	–	–	13	–	–	–
matrix	49	26	–	2	–	64	34	–	–	–	23	26	49	–	–	2	45	2	–	2
conform	104	82	5	5	5	261	240	–	–	–	22	82	91	5	5	3	91	5	5	3
matrix-jw	37	10	–	1	–	50	14	–	–	–	27	10	38	–	–	–	36	1	–	–
peval	55	41	3	3	3	175	134	–	–	–	14	41	44	3	8	–	44	3	8	–
nucleic-sorted	265	236	2	2	2	7	2	–	–	–	29	236	124	2	60	79	124	2	60	79
nucleic-star	265	260	5	5	5	743	738	–	–	–	5	260	124	5	57	79	124	5	57	79
fxtakr	101	–	–	–	–	401	–	–	–	–	101	–	101	–	–	–	101	–	–	–
em-imp	103	47	1	1	1	204	148	–	–	–	56	47	94	1	7	1	94	1	7	1
nucleic-jw	48	34	5	5	5	173	160	80	80	–	14	34	38	5	4	1	38	5	4	1
em-fun	102	62	1	1	1	264	224	–	–	–	40	62	94	1	7	–	94	1	7	–
lalr	349	292	3	6	3	303	245	–	–	–	57	292	186	3	16	163	166	6	15	163
takr	101	–	–	–	–	401	–	–	–	–	101	–	101	–	–	–	101	–	–	–
nbody	58	6	–	–	–	79	–	–	–	–	52	6	56	–	2	–	56	–	2	–
interpret	122	110	1	1	1	267	251	–	–	–	12	110	119	1	2	–	119	1	2	–
dynamic	201	187	2	2	2	561	548	–	–	–	14	187	144	2	41	14	144	2	41	14
texer	146	80	13	18	13	592	497	–	–	–	66	80	132	13	7	–	126	18	2	–
similix	527	141	–	1	–	1705	322	–	–	–	386	141	484	–	36	8	483	1	35	8
ddd	1161	550	14	45	14	3063	2164	2	2	2	611	550	1182	14	10	124	982	45	10	124
softscheme	1049	865	132	134	132	3382	2793	–	8	–	184	865	858	132	47	147	798	134	43	147
chezscheme	2411	1289	140	169	140	6051	4339	–	18	–	1122	1289	2137	140	110	114	2039	169	89	114

Table 1: Number of introduced assignments and valid checks for the straightforward R^5RS transformation, the modified R^5RS transformation (easy) described in Section 4, and for the Assimilating (A), Non-assimilating (N), and Sequential `letrec*` (S) variants of our transformation. Also shown are the number of bindings in the *lambda* (λ), *complex* (c), *simple* (s), and *unreferenced* (u), partitions for the modified R^5RS transformation and for our transformation with and without assimilation. (All bindings are *complex* in standard R^5RS transformation.) Since assimilation incorporates both nested `let` and `letrec` bindings, the total number of bindings may be greater when assimilation is enabled.

	R^5RS		R^5RS easy		A	N	A	N	S
Checks:	no	naive	no	naive	yes	yes	no	no	yes
fxtak	1.00	1.16	.81	.81	.81	.81	.81	.81	.81
tak	1.00	1.12	.87	.87	.87	.87	.87	.87	.87
div-iter	1.00	1.05	.93	.93	.93	.93	.93	.93	.93
cpstak	1.00	1.03	.85	.85	.85	.85	.85	.85	.85
takl	1.00	1.23	.71	.71	.71	.71	.71	.71	.71
ctak	1.00	1.02	.89	.89	.89	.89	.89	.89	.89
mbrot	1.00	1.01	.99	.99	.98	.99	.99	.98	.99
deriv	1.00	1.02	.97	.97	.96	.96	.96	.96	.96
destruct	1.00	1.05	.81	.82	.81	.81	.81	.81	.81
fxtriang	1.00	1.15	.87	.87	.81	.80	.80	.80	.81
fft-f	1.00	1.01	.91	.91	.91	.91	.91	.91	.91
fft-d	1.00	1.00	.99	.99	.99	.99	.99	.99	.99
dderiv	1.00	1.03	.94	.94	.94	.94	.94	.94	.94
triang	1.00	1.11	.90	.91	.85	.85	.85	.85	.85
lattice	1.00	1.04	.53	.54	.52	.53	.52	.53	.52
boyer	1.00	1.13	.88	.92	.86	.86	.86	.86	.86
boyer-jw	1.00	1.18	1.00	1.18	.96	.96	.96	.96	.96
browse	1.00	1.01	.97	.97	.94	.94	.94	.94	.94
traverse	1.00	1.10	1.05	1.09	.97	.97	.97	.97	.97
lattice-jw	1.00	1.04	.80	.84	.28	.28	.28	.28	.28
fft-g	1.00	1.01	.88	.89	.88	.88	.88	.88	.88
ray	1.00	1.09	.99	1.06	.76	.75	.75	.75	.76
fxpuzzle	1.00	1.15	.76	.76	.77	.77	.77	.77	.77
graphs	1.00	1.00	.39	.60	.39	.39	.39	.39	.39
tcheck	1.00	1.01	.99	1.00	.96	.96	.96	.96	.96
simplex	1.00	1.06	.55	.56	.54	.54	.54	.54	.54
graphs-jw	1.00	1.01	.54	.54	.54	.54	.54	.54	.54
maze	1.00	1.12	.79	.83	.55	.55	.55	.55	.55
maze-jw	1.00	1.03	.70	.70	.70	.70	.70	.70	.70
puzzle	1.00	1.10	.89	.88	.88	.88	.88	.88	.88
earley	1.00	1.03	.73	.73	.73	.73	.73	.73	.73
splay	1.00	1.00	.77	.77	.77	.77	.77	.77	.77
matrix	1.00	.99	.62	.63	.59	.59	.59	.59	.59
conform	1.00	1.13	.92	1.09	.38	.38	.38	.38	.38
matrix-jw	1.00	1.01	.68	.68	.60	.60	.60	.60	.60
peval	1.00	1.08	.93	.98	.78	.78	.78	.78	.78
nucleic-sorted	1.00	.99	.98	.99	.74	.74	.74	.74	.74
nucleic-star	1.00	1.08	1.00	1.09	.76	.76	.76	.76	.76
fxtakr	1.00	1.56	.72	.73	.73	.73	.72	.73	.73
em-imp	1.00	1.05	.75	.77	.66	.66	.66	.66	.66
nucleic-jw	1.00	1.00	1.00	1.00	.99	.98	.98	.98	.99
em-fun	1.00	1.04	.77	.81	.69	.69	.69	.69	.69
lalr	1.00	1.03	.89	.90	.82	.81	.82	.81	.82
takr	1.00	1.17	.56	.56	.56	.56	.56	.56	.56
nbody	1.00	1.01	.72	.72	.66	.66	.66	.66	.66
interpret	1.00	1.20	1.02	1.00	.90	.90	.91	.91	.90
dynamic	1.00	1.01	.97	1.01	.93	.93	.93	.93	.93
texer	1.00	.91	.55	.58	.53	.53	.53	.53	.53
similix	1.00	1.02	1.00	1.01	.97	.97	.96	.96	.96
ddd	1.00	1.03	1.00	.99	.97	.96	.97	.96	.98
softscheme	1.00	1.17	.96	1.14	.79	.79	.79	.80	.79
chezscheme	1.00	1.10	.75	.84	.65	.66	.66	.65	.65

Table 2: Run time of the code produced by the various algorithms, normalized to the R^5RS baseline.

	R^{5}RS		R^{5}RS easy		A	N	A	N	S
Checks:	no	naive	no	naive	yes	yes	no	no	yes
fxtak	1.00	1.30	.90	.90	.90	.90	.90	.90	.90
tak	1.00	1.24	.91	.91	.91	.91	.91	.91	.91
div-iter	1.00	1.21	.47	.55	.47	.47	.47	.47	.47
cpstak	1.00	1.18	.93	.75	.93	.93	.93	.93	.93
takl	1.00	1.24	.83	.83	.83	.83	.83	.83	.83
ctak	1.00	1.19	.95	.95	.95	.95	.95	.95	.95
mbrot	1.00	1.15	.72	.80	.72	.72	.72	.72	.72
deriv	1.00	1.19	.96	.96	.96	.96	.96	.96	.96
destruct	1.00	1.12	.68	.74	.68	.68	.68	.68	.68
fxtriang	1.00	1.10	.84	.86	.77	.77	.77	.77	.77
fft-f	1.00	1.11	.69	.73	.69	.69	.69	.69	.69
fft-d	1.00	1.17	.82	.84	.82	.82	.82	.82	.82
dderiv	1.00	1.07	1.15	1.15	1.15	1.15	1.15	1.15	1.15
triang	1.00	1.08	.88	.90	.83	.83	.83	.83	.83
lattice	1.00	1.29	.94	1.20	.61	.64	.61	.64	.61
boyer	1.00	1.39	.99	1.37	.63	.63	.63	.63	.63
boyer-jw	1.00	1.52	1.00	1.52	.76	.76	.76	.76	.76
browse	1.00	1.24	.91	1.07	.76	.76	.76	.76	.76
traverse	1.00	1.35	.99	1.22	.57	.57	.57	.57	.57
lattice-jw	1.00	1.21	.92	1.06	.73	.76	.73	.76	.73
fft-g	1.00	1.07	1.05	1.08	.98	.98	.98	.98	.98
ray	1.00	1.40	.97	1.29	.54	.54	.54	.54	.54
fxpuzzle	1.00	1.13	.74	.76	.74	.74	.74	.74	.74
graphs	1.00	1.17	.72	.83	.72	.72	.72	.72	.72
tcheck	1.00	1.43	.98	1.38	.77	.77	.77	.77	.77
simplex	1.00	1.27	.60	.65	.59	.59	.59	.59	.59
graphs-jw	1.00	1.10	.71	.71	.71	.71	.71	.71	.71
maze	1.00	1.46	.94	1.29	.46	.46	.46	.46	.46
maze-jw	1.00	1.10	.46	.46	.46	.46	.46	.46	.46
puzzle	1.00	1.09	.87	.88	.87	.87	.87	.87	.87
earley	1.00	1.28	.49	.52	.49	.49	.49	.49	.49
splay	1.00	1.08	.86	.86	.86	.86	.86	.86	.86
matrix	1.00	1.18	.91	1.01	.67	.71	.67	.71	.67
conform	1.00	1.49	.94	1.41	.51	.51	.51	.51	.51
matrix-jw	1.00	1.15	.88	.92	.80	.80	.80	.80	.80
peval	1.00	1.40	.95	1.26	.76	.76	.76	.76	.76
nucleic-sorted	1.00	1.01	.96	.97	.39	.39	.39	.39	.39
nucleic-star	1.00	1.56	.99	1.55	.40	.40	.40	.40	.40
fxtakr	1.00	1.54	.59	.59	.59	.59	.59	.59	.59
em-imp	1.00	1.25	.93	1.07	.66	.66	.66	.66	.66
nucleic-jw	1.00	1.32	.95	1.25	.90	.90	.64	.64	.64
em-fun	1.00	1.33	.99	1.24	.68	.68	.68	.68	.68
lalr	1.00	1.18	.92	1.06	.53	.54	.53	.54	.53
takr	1.00	1.38	.72	.72	.72	.72	.72	.72	.72
nbody	1.00	1.10	1.01	1.01	.98	.98	.98	.98	.98
interpret	1.00	1.30	.99	1.27	1.09	1.09	1.09	1.09	1.09
dynamic	1.00	1.34	1.02	1.36	1.16	1.16	1.16	1.16	1.16
texer	1.00	1.28	.94	1.16	.93	.93	.93	.93	.93
similix	1.00	1.19	.94	.98	.91	.91	.91	.91	.91
ddd	1.00	1.21	.93	1.08	.81	.87	.78	.87	.81
softscheme	1.00	1.23	.98	1.18	.87	.89	.87	.88	.87
chezscheme	1.00	1.18	.98	1.11	.96	.97	.96	.97	.96

Table 3: Size of the object code produced by the various algorithms, normalized to the R^{5}RS baseline.

	R^{5}RS		R^{5}RS easy		A	N	A	N	S
Checks:	no	naive	no	naive	yes	yes	no	no	yes
fxtak	1.00	1.00	.50	1.00	.50	.50	1.00	.50	.50
tak	1.00	.50	.50	1.00	.50	.50	.50	1.00	.50
div-iter	1.00	.50	1.00	.50	1.00	.50	.50	.50	.50
cpstak	1.00	1.00	1.00	1.00	.50	1.00	1.00	.50	1.00
takl	1.00	1.00	.50	1.00	1.00	1.00	.50	.50	1.00
ctak	1.00	.50	.50	1.00	.50	.50	1.00	.50	1.00
mbrot	1.00	1.00	1.00	.67	.67	1.00	.67	1.00	1.00
deriv	1.00	1.00	1.00	1.00	1.00	1.00	.50	.50	1.00
destruct	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
fxtriang	1.00	1.00	1.00	1.50	1.50	1.00	1.00	.50	1.00
fft-f	1.00	.75	.75	.75	.75	.75	.75	.75	.75
fft-d	1.00	1.00	1.00	1.00	1.00	1.00	.50	1.00	1.00
dderiv	1.00	2.00	2.00	2.00	2.00	1.00	1.00	2.00	1.00
triang	1.00	1.50	1.50	1.00	.50	1.00	1.00	1.00	1.50
lattice	1.00	1.33	1.00	1.00	.67	1.00	1.00	1.00	1.00
boyer	1.00	1.25	1.00	1.00	1.00	.50	.75	.50	.50
boyer-jw	1.00	1.67	1.00	1.67	1.33	1.33	.67	.67	1.33
browse	1.00	1.33	.67	1.00	.67	1.00	1.00	.67	1.00
traverse	1.00	1.25	1.00	1.25	.75	.75	.75	.50	.75
lattice-jw	1.00	.75	1.00	.75	1.00	1.00	1.00	1.00	1.00
fft-g	1.00	1.00	1.50	1.50	1.50	1.00	1.50	1.50	1.00
ray	1.00	1.17	1.00	.83	.67	.83	.83	.83	.83
fxpuzzle	1.00	1.33	1.33	1.33	1.33	1.33	1.00	1.33	1.33
graphs	1.00	1.00	1.00	.80	1.00	1.00	.80	1.00	1.00
tcheck	1.00	1.50	1.00	1.25	1.00	1.00	1.00	1.25	1.25
simplex	1.00	1.14	.86	1.00	1.00	.86	.86	.86	.86
graphs-jw	1.00	.83	.83	.83	.83	.83	.83	.83	.67
maze	1.00	1.56	1.11	1.33	.89	.89	.89	.89	.89
maze-jw	1.00	.90	.70	.60	.60	.80	.80	.80	.80
puzzle	1.00	1.67	1.33	1.67	1.67	1.67	1.33	1.67	1.33
earley	1.00	1.40	.90	1.00	.90	1.00	.80	.90	1.00
splay	1.00	1.29	1.00	1.14	1.14	1.14	1.00	1.00	1.14
matrix	1.00	1.14	.86	1.14	.86	.86	1.00	1.00	.86
conform	1.00	1.36	.91	1.36	.73	.73	.73	.73	.73
matrix-jw	1.00	1.17	1.17	1.17	1.17	1.17	1.17	.83	1.17
peval	1.00	1.36	1.00	1.36	1.09	1.00	1.09	1.09	1.09
nucleic-sorted	1.00	1.00	1.03	1.07	.77	.77	.77	.77	.73
nucleic-star	1.00	1.41	1.00	1.38	.76	.76	.79	.76	.79
fxtakr	1.00	1.91	1.45	1.45	1.45	1.45	1.45	1.45	1.45
em-imp	1.00	1.25	1.00	1.12	.88	.88	.81	.75	.75
nucleic-jw	1.00	1.09	.95	.95	.95	.95	.91	.91	.82
em-fun	1.00	1.25	1.00	1.25	.88	.88	.88	.81	.88
lalr	1.00	1.11	1.00	1.14	.86	.89	.86	.89	.86
takr	1.00	1.29	1.06	1.06	1.06	1.06	1.06	1.06	1.06
nbody	1.00	1.06	1.00	1.06	1.12	1.12	1.12	1.00	1.06
interpret	1.00	1.17	.96	1.12	1.08	1.00	1.08	1.08	1.00
dynamic	1.00	1.32	1.05	1.32	1.29	1.29	1.16	1.13	1.26
texer	1.00	1.24	1.02	1.16	1.07	1.07	1.02	1.02	1.09
similix	1.00	1.16	1.01	1.04	.98	1.03	.98	.97	.98
ddd	1.00	1.19	.97	1.14	.91	.98	.87	.97	.91
softscheme	1.00	1.30	.99	1.27	1.01	1.02	.96	.96	1.01
chezscheme	1.00	1.17	1.01	1.16	1.10	1.10	1.09	1.08	1.10

Table 4: Total compile times, normalized to the R^{5}RS baseline. The coarse granularity of the timing mechanism gives us poor differentiation among many of the times, since compile times for most of the programs are very small.

A Variadic Extension of Curry's Fixed-Point Combinator

Mayer Goldberg (gmayer@cs.bgu.ac.il)*
Department of Computer Science
Ben Gurion University, Beer Sheva 84105, Israel

ABSTRACT

We present a systematic construction of a variadic applicative-order multiple fixed-point combinator in Scheme. The resulting Scheme procedure is a variadic extension of the n-ary version of Curry's fixed-point combinator. It can be used to create mutually-recursive procedures, and expand arbitrary `letrec`-expressions.

Keywords: Fixed points, fixed-point combinators, applicative order, lambda-calculus, Scheme, variadic functions

1. INTRODUCTION

Since the early days of Scheme programming, defining and using various fixed-point combinators have been classical programming exercises (for example, *Structure and Interpretation of Computer Programs* [1, Section 4.1.7, Page 393], and *The Little LISPer* [6, Chapter 9, Page 171]): Fixed-point combinators are used to replace recursion and circularity in procedures and data structures, with self application.

Replacing *mutual recursion* with self-application is done in one of two ways: (A) We can reduce mutually-recursive functions to simple recursive functions, and use a singular fixed-point combinator to replace singular recursion with self-application. Examples of this approach can be found in Bekič's theorem for the elimination of simultaneous recursion [3, Page 39], and in Landin's classical work on the mechanical evaluation of expressions [7]. (B) We can use a set of *multiple fixed-point combinators*. This approach is taken in a particularly beautiful construction due to Smullyan [2, Pages 334-335]. When replacing recursion among $n \geq 1$ recursive functions, a different set of multiple fixed-point combinators needs to be used for each n, each set containing progressively more complex expressions.

In Scheme, however, we can do better. Scheme provides syntax for writing *variadic procedures* (i.e., procedures that take arbitrarily many arguments), so that upon application, an identifier is bound to a list of these arguments. Scheme also provides two procedures that are particularly suitable for use in combination with variadic procedures:

- The `apply` procedure, which takes a procedure and a list, and applies the procedure to the elements of the list, as if it were called directly with the elements of the list as its arguments. For example: `(apply + '(1 2 3))` returns the same result as `(+ 1 2 3)`.

- The `map` procedure, which, in its simplest form, takes a procedure and a list of arguments, and applies the procedure to each one of these arguments, returning a list of the results. For example, `(map list '(1 2 3))` returns the list `((1) (2) (3))`.

By using variadic procedures, `apply`, and `map`, we can define a single Scheme procedure that can be used to define any number of mutually-recursive procedures. This way, we would not have to specify that number in advance.

The construction of variadic multiple fixed-point combinators is not immediate. We are only aware of one published solution — in Queinnec's book *LISP In Small Pieces* [8]. In Section 5 we compare our construction and the one found in Queinnec's book [8, Pages 457–458].

This work presents a variadic multiple fixed-point combinator that is a natural extension of Curry's fixed-point combinator. Our construction uses only as many Scheme-specific idioms as needed for working with variadic procedures (namely, `apply` and `map`), and is thus faithful both to the spirit of Scheme, in which it is written, as well as to the λ-calculus whence it comes.

The rest of this paper is organized as follows. We first review standard material about fixed-point combinators for singularly recursive procedures (Section 2), and then how to extend fixed-point combinators to handling mutual recursion among n procedures (Section 3). We then present our applicative-order variadic multiple fixed-point combinator (Section 4), and compares it with Queinnec's solution (Section 5). Section 6 concludes.

*This work was carried out while visiting Basic Research in Computer Science (`www.brics.dk`),. funded by the Danish National Research Foundation.

Third Workshop on Scheme and Functional Programming. October 3, 2002, Pittsburgh, Pennsylvania, USA.

2. SINGULAR FIXED-POINT COMBINATORS

Fixed-point combinators are used in the λ-calculus to solve fixed-point equations. Given a term M, we are looking for a term x (in fact, the "smallest" such x in a lattice-theoretic sense) that satisfies the equation $Mx = x$ (the fixed-point equation), where equality is taken to be the equivalence relation induced by the one-step $\beta\eta$-relation. A *fixed-point combinator* is a term that takes any term M as an argument and returns the fixed point of M. If Φ is a fixed-point combinator, and M is some term, then $x = \Phi M$ is the fixed point of M, and satisfies $Mx = x$. Substituting the definition of x into the fixed point equation, we see that a fixed-point combinator is a term Φ such that for any term M, $\Phi M = M(\Phi M)$.

There exist infinitely-many different fixed-point combinators, though some are particularly well-known. The best known fixed-point combinator is due to Haskell B. Curry [4, Page 178]:

$$Y_{\text{Curry}} \equiv \lambda f.((\lambda x.f(xx))(\lambda x.f(xx)))$$

Encoding literally the above in Scheme would not work: Under Scheme's *applicative order* the application of

```
(lambda (f)
  ((lambda (x) (f (x x)))
   (lambda (x) (f (x x)))))
```

to *any* argument will diverge, because the application `(x x)` will evaluate before `f` is applied to it, resulting in an infinite loop. The solution is to replace `(x x)` with an expression that is both equivalent, and in which the evaluation of the given appilcation is delayed, namely, with a `lambda`-expression: If `(x x)` should evaluate to a one-argument procedure, then we can replace it with `(lambda (arg) ((x x) arg))`; If to a two-argument procedure, then we can replace it with `(lambda (arg1 arg2) ((x x) arg1 arg2))`, etc.[1] Not wanting to commit, however, to the arity of `(x x)`, we will use a variadic version of the η-expansion, i.e., wrap `(x x)` with `(lambda args (apply ⋯ args))`, giving:

```
(define Ycurry
 (lambda (f)
   ((lambda (x)
      (f (lambda args (apply (x x) args))))
    (lambda (x)
      (f (lambda args (apply (x x) args)))))))
```
(1)

Fixed-point combinators are used in programming languages in order to define recursive procedures [6, 7]. The trick is to define the recursive procedure as the solution to some fixed-point equation, and then use a fixed-point combinator to solve this equation. For example, the Scheme procedure that computes the *factorial* function satisfies the following recurrence relation

```
fact ≡ (lambda (n)
         (if (zero? n) 1
             (* n (fact (- n 1)))))
```

can be rewritten as the solution of the following fixed-point equation:

```
fact = ((lambda (fact)
          (lambda (n)
            (if (zero? n) 1
              (* n (fact (- n 1))))))) fact)
```

and can be solved using, e.g., Curry's fixed-point combinator (Expr. (1)):

```
(define fact
  (Ycurry
   (lambda (fact)
     (lambda (n)
       (if (zero? n) 1
           (* n (fact (- n 1))))))))
```

[1]This transformation is known colloquially as "η-expansion." The *η-reduction* consists of replacing $(\lambda\nu.M\nu)$ with M when ν does not occur free in M. The point of the η expansion in Scheme is that the body of procedures evaluate at application time rather than at closure-creation time, and so the η-expansion is used to *delay* evaluation.

3. MULTIPLE FIXED-POINT COMBINATORS

Just as recursive functions are solutions to fixed-point equations, which can be solved using fixed-point combinators, so are *mutually recursive functions* the solutions to *multiple fixed-point equations*, which can be solved using *multiple fixed-point combinators*:

A set of n multiple fixed points is defined as follows: Given the terms $M_1, \ldots, M_n$, we want to find terms $x_1, \ldots, x_n$ (the set of fixed points), such that $x_i = M_i x_1 \cdots x_n$, for $i = 1, \ldots, n$ (a system of n multiple fixed-point equations).

Extending the notion of singular fixed-point combinators, n multiple fixed-point combinators are terms $\Phi_1^n, \ldots, \Phi_n^n$, such that for any $M_1, \ldots, M_n$, we can let $x_i = \Phi_i^n M_1 \cdots M_n$, for $i = 1, \ldots, n$, and $\{x_i\}_{i=1}^n$ are the multiple fixed points that solve the given system of equations. Substituting the definitions of $\{x_i\}_{i=1}^n$ into the system of multiple fixed-point equations, we arrive at the concise statement that multiple fixed-point combinators are terms that $\Phi_1^n, \ldots, \Phi_n^n$, such that for any $M_1, \ldots, M_n$, we have

$$(\Phi_i^n M_1 \cdots M_n) = M_i(\Phi_1^n M_1 \cdots M_n) \cdots (\Phi_n^n M_1 \cdots M_n)$$

for all $i = 1, \ldots, n$.

Curry's fixed-point combinator can be extended to a set of n multiple fixed-point combinators for solving a system of n multiple fixed-point equations. The i-th such extension, $Y_{\text{Curry}_i^n}$, is given by

$$\begin{aligned} Y_{\text{Curry}_i^n} \equiv \lambda f_1 \cdots f_n.((\lambda x_1 \cdots x_n.f_i(x_1 x_1 \cdots x_n) \\ \cdots \\ (x_n x_1 \cdots x_n)) \\ (\lambda x_1 \cdots x_n.f_1(x_1 x_1 \cdots x_n) \\ \cdots \\ (x_n x_1 \cdots x_n)) \\ \cdots \\ (\lambda x_1 \cdots x_n.f_n(x_1 x_1 \cdots x_n) \\ \cdots \\ (x_n x_1 \cdots x_n))) \end{aligned}$$

Encoded in Scheme, the *applicative-order* version of Curry's multiple fixed-point combinator, $Y_{\mathrm{Curry}_i}{}^n$, is given by:

```
(define Ycurryin
  (lambda (f1 ... fn)
    ((lambda (x1 ... xn)
       (fi (lambda args
             (apply (x1 x1 ... xn) args))
           ...
           (lambda args
             (apply (xn x1 ... xn) args))))
     (lambda (x1 ... xn)
       (f1 (lambda args
             (apply (x1 x1 ... xn) args))
           ...
           (lambda args
             (apply (xn x1 ... xn) args))))
     ...
     (lambda (x1 ... xn)
       (fn (lambda args
             (apply (x1 x1 ... xn) args))
           ...
           (lambda args
             (apply (xn x1 ... xn) args)))))))
```
(2)

Obviously, this is an *abbreviated meta-notation*, and for any specific n, the ellipsis ('$\cdots$') would need to be replaced with the corresponding Scheme expressions (so that `fi` refers to one of an actual list of parameters). In fact, one way to describe the aim of this paper is that we would like to avoid this meta-linguistic shorthand, and construct a Scheme procedure that takes arbitrarily-many arguments and returns a list of their multiple fixed points.

Mutually recursive function definitions can be rewritten as solutions to multiple fixed-point equations. For example, the Scheme procedures that compute the predicates *even*, *odd* satisfy the following mutual recurrence relation:

```
even?  ≡  (lambda (n)
            (if (zero? n) #t
              (odd?  (- n 1))))

odd?   ≡  (lambda (n)
            (if (zero? n) #f
              (even? (- n 1))))
```

can be rewritten as the solutions of the following system of multiple fixed-point equations:

```
even?  =  ((lambda (even?  odd?)
             (lambda (n)
               (if (zero? n) #t
                 (odd? (- n 1))))) even? odd?)

odd?   =  ((lambda (even?  odd?)
             (lambda (n)
               (if (zero? n) #f
                 (even? (- n 1))))) even? odd?)
```

and can be solved using Curry's multiple fixed-point combinators, where $n = 2$, and $i = 1, 2$:

```
(define E
  (lambda (even? odd?)
    (lambda (n)
      (if (zero? n) #t
          (odd? (- n 1))))))
(define O
  (lambda (even? odd?)
    (lambda (n)
      (if (zero? n) #f
          (even? (- n 1))))))
(define even? (Ycurry12 E O))
(define odd? (Ycurry22 E O))
```
(3)

The aim of the next section is to show how we can construct a *variadic* version of Curry's multiple fixed-point combinator that can be used to define any number of recursive mutually-recursive procedures.

4. A VARIADIC MULTIPLE FIXED-POINT COMBINATOR

Expr. (Expr. (2)) specifies `Ycurryin` for any i, n, such that $1 \leq i \leq n$. For different choices of i, n, we would get a different procedure, and as n grows, each procedure gets progressively larger and more complex. This could be a real problem, for example, if we were to use multiple fixed-point combinators to expand `letrec`-expressions: We would need many different multiple fixed-point combinators, for many different values of n, even in a moderately-large program. We could, of course, hide the multiple fixed-point combinators through the use of a macro, but we couldn't hide the code bloat that would follow from the creation of a large number of these ever-growing "recursion-makers."

We address this issue by constructing a variadic multiple fixed-point combinator in Scheme. Variadic procedures, used together with the builtin procedures `apply` and `map`, form the basis for our programming idioms for working with meta-linguistic ellipsis in Scheme.

Throughout the rest of the section we are going to employ the following conventions, or rules, for converting "meta-linguistic Scheme" into actual Scheme:

Argv Lists of arguments will be written in Scheme as a single variable named in the *plural*. For example, `x1` $\cdots$ `xn` and `f1` $\cdots$ `fn` will be written as `xs` and `fs` respectively.

AbsArgv An abstraction over a list of arguments will be written in Scheme using a variadic lambda. For example: `(lambda (f1` $\cdots$ `fn)` *`M`*`)` will be written as `(lambda fs` *`M`*`)`.

AppArgv An application of a procedure to a list of arguments will be written as an application of the Scheme procedure `apply` to the procedure and the variable denoting the list of arguments. For example: The expression `(xi x1` $\cdots$ `xn)` will be written as `(apply xi xs)`.

IndAbsArgv A list of expressions that is indexed by some variable (e.g., `(x`*`i`* `x1` $\cdots$ `xn)`, which is indexed by `x`*`i`* $=$ `x1` $\cdots$ `xn`) will be written in

the following way: We shall consider a "representative member" indexed by some variable, and then abstract over that variable and map the resulting procedure over the indexing set, using the `map` procedure. For example, the list of terms `(apply x1 xs)` $\cdots$ `(apply xn xs)` will be obtained by considering the "representative member" `(apply x`i `xs)`, abstracting over `x`i, and mapping the resulting procedure over `xs`, giving

```
(map (lambda (xi) (apply xi xs)) xs)
```

Recall ${Y_{\text{Curry}}}_i^n$, the variadic extension of Curry's fixed-point combinator, in Scheme (Expr. (2)):

```
(lambda (f1 ··· fn)
 ((lambda (x1 ··· xn)
   (fi (lambda args
         (apply (x1 x1 ··· xn) args))
       ···
       (lambda args
         (apply (xn x1 ··· xn) args))))
  (lambda (x1 ··· xn)
   (f1 (lambda args
         (apply (x1 x1 ··· xn) args))
       ···
       (lambda args
         (apply (xn x1 ··· xn) args))))
  ···
  (lambda (x1 ··· xn)
   (fi (lambda args
         (apply (x1 x1 ··· xn) args))
       ···
       (lambda args
         (apply (xi x1 ··· xn) args))
       ···
       (lambda args
         (apply (xn x1 ··· xn) args))))
  ···
  (lambda (x1 ··· xn)
   (fn (lambda args
         (apply (x1 x1 ··· xn) args))
       ···
       (lambda args
         (apply (xn x1 ··· xn) args)))))))
```

The various representative sub-expressions we will consider are enclosed in nested frames.

Starting with the innermost frame, for any `x`i = `x1` ... `xn`, the application `(x`i `x1` $\cdots$ `xn)` is written, using the AppArgv rule, as

```
(apply xi xs)                                    (4)
```

Moving outward, towards the next enclosing frame, we apply to Expr. (4) the variadic version of the η-expansion in order to make sure that our fixed points reduce properly under applicative order:

```
(lambda args                                     (5)
 (apply (apply xi xs) args))
```

Note that the above expression is indexed by `x`i (that is, `x`i is a free variable that ranges over a list) in Expr. (5), and we need to obtain the list of such expressions for each `x`i = `x1` $\cdots$ `xn`. Using the IndAbsArgv rule, we abstract `x`i over Expr. (5) and map the resulting procedure over the list `xs`. The list of applications is therefore given by

```
(map (lambda (xi)                                (6)
       (lambda args
         (apply (apply xi xs) args))) xs)
```

Moving outward towards the next enclosing frame, we see that Expr. (6) forms the list of arguments to `f`i (which is also a free variable that ranges over a list). Using the AppArgv rule, the application is written out using `apply`:

```
(apply fi                                        (7)
 (map (lambda (xi)
        (lambda args
          (apply
            (apply xi xs) args))) xs))
```

Moving outward, towards the next enclosing frame, we see that Expr. (7) is the body of an abstraction over `x1` $\cdots$ `xn`. Using the AbsArgv rule, we encode this abstraction using a variadic lambda with the parameter `xs`:

```
(lambda xs                                       (8)
 (apply fi
  (map (lambda (xi)
         (lambda args
           (apply
             (apply xi xs) args))) xs)))
```

Moving outward, towards the next enclosing frame, we see that Expr. (8) is indexed by `f`i (that is, `f`i is a free variable that ranges over a list) in Expr. (8), and we need to obtain the list of such expressions for each `x`i = `x1` ... `xn`. Using the IndAbsArgv rule, we abstract `f`i over Expr. (8) and map the resulting procedure over the list `fs`. The list of applications is therefore given by

```
(map (lambda (fi)                                (9)
       (lambda xs
         (apply fi
           (map (lambda (xi)
                  (lambda args
                    (apply (apply xi xs) args)))
                xs))))
     fs)
```

The above expression corresponds to the list `x1` $\cdots$ `xn`. The next step is to compute the list of multiple fixed points: For any particular $i \in \{1, \ldots, n\}$, the i-th fixed-point combinator ${Y_{\text{Curry}}}_i^n$ is given by `(x`i `x1` $\cdots$ `xn)`, which, in Scheme would be written as `(apply x`i `xs)`. Using the IndAbsArgv rule, to obtain the list of all such terms, for each `x`i = `x1` ... `xn`, we abstract `x`i over Expr. (9) and map the resulting procedure over the list `xs`. This is the second time we have referred to the list `x1` $\cdots$ `xn` in this step, so rather than compute it twice, we bind its value to the identifier `xs`, using a `let`-expression, the body of which will be:

```
(map (lambda (xi)
       (apply xi xs)) xs)
```

Using the AbsArgv rule, we now abstract over `f1` $\cdots$ `fn` using a variadic `lambda`, and define the procedure `curry-fps` that takes any number of procedures and returns a list of their multiple fixed points:

```
(define curry-fps
  (lambda fs
    (let ((xs
           (map
            (lambda (fi)
              (lambda xs
                (apply fi
                 (map
                  (lambda (xi)
                    (lambda args
                      (apply (apply xi xs) args)))
                    xs))))
            fs)))
      (map (lambda (xi)
             (apply xi xs)) xs))))
```
(10)

On the other hand, if we are only interested in $Y_{\mathrm{Curry}_1}{}^n$, for example, for the purpose of macro-expanding `letrec`-expressions without using side-effects, then we can simplify the body of the `let`-expression in Expr. (10) so that we just compute the first fixed point:

```
(define curry-fps-1n
  (lambda fs
    (let ((xs
           (map
            (lambda (fi)
              (lambda xs
                (apply fi
                 (map (lambda (xi)
                        (lambda args
                          (apply
                           (apply xi xs) args)))
                      xs))))
            fs)))
      (apply (car xs) xs))))
```
(11)

For example, consider the general `letrec`-expression, where $M_1, \ldots, M_n$ denote the definitions of the procedures `f1`, $\ldots$, `fn` respectively, and $Expr_1, \ldots, Expr_m$ denote the expressions in the body of the letrec:

```
(letrec ((f1 M_1)
         ⋮
         (fn M_n))
  Expr_1 ··· Expr_m)
```

Using `curry-fps-1n`, the above expression can be rewritten, without side effects, using the *fresh variable* `body`, as follows:

```
(curry-fps-1n
 (lambda (body f1 ··· fn) Expr_1 ··· Expr_m)
 (lambda (body f1 ··· fn) M_1)
                          ⋮
 (lambda (body f1 ··· fn) M_n))
```

5. RELATED WORK

In his book *LISP In Small Pieces* [8, Pages 457–458], Queinnec exhibits the Scheme procedure `NfixN2`, that is a variadic, applicative-order multiple fixed-point combinator. The `NfixN2` procedure, along with a help procedure are given below:

```
(define NfixN2
  (let ((d
         (lambda (w)
           (lambda (f*)
             (map
              (lambda (f)
                (apply f
                 (map
                  (lambda (i)
                    (lambda a
                      (apply
                       (list-ref ((w w) f*)
                                 i)
                       a)))
                  (iota 0 (length f*)))))
              f*)))))
    (d d)))

(define iota
  (lambda (start end)
    (if (< start end)
        (cons start (iota (+ 1 start) end))
        '())))
```
(12)

While Queinnec's construction certainly works, it strikes us as unnatural in the context of Scheme:

The name of the `iota` help procedure comes from the programming language APL, where, given an integer argument n, the monatic *iota* function returns the vector of integer in the range $1, \ldots, n$. The above implementation of *iota* in Scheme takes two integers *start* and *end*, and returns the *list* of integers in the range of $start, \ldots, end - 1$. A common programming idiom in APL is to de-reference a vector `v` by another vector `w` of indecies into `v`, to obtain a permuted sub-vector of `v` that is the same size as `w`. We note the use of this idiom in the procedure `NfixN2`, where the list `(iota 0 (length f*))` is used as a list of indecies for extracting elements from the list returned by `((w w) f*)`, and is used to construct a new list in the expression `(map (lambda (i) ···) (iota 0 (length f*)))`. In fact, the procedure `NfixN2` could be coded directly and concisely into DyalogAPL [5], a dialect of APL2 that supports closures and higher-order functions.

Furthermore, the construction of the variadic fixed-point combinator is not a natural extension of one of the familiar singular fixed-point combinators, e.g., Curry's or Turing's fixed-point combinators. This is probably due to the use of APL idioms in the code.

We feel that from a pedagogical point of view, it would be better to exhibit a variadic fixed-point combinator that is a natural extension of one of the familiar singular fixed-point combinators, in a way that would be both systematic and native to Scheme.

The natural idioms for working with variadic procedures are `apply` and `map`. By sticking to these procedures and de-

clining the temptations to use other procedures and other metaphors, we obtained a construction for multiple fixed-point combinator that corresponds rather faithfully to the extension of Curry's fixed-point combinator to n multiple fixed-point equations.

An additional benefit is efficiency: While fixed-point combinators in Scheme are not normally evaluated by their efficiency (but rather by applicative order call-by-value), and while this remark is not intended to be a sales pitch for super-fast, super-efficient fixed-point combinators, it should not be surprising that when we adhere to programming metaphors that are natural for a given language, we are often rewarded by better execution times. The `NfixN2` (Expr. (12)) procedure presented in Section 5 uses a variant of the *iota* function in APL in order to generate indices, repeatedly, with each recursive call. But generating indices is only half the problem: The individual functions are then accessed using `list-ref`, which traversing a list of functions at linear time. This suggests that the average time to access a function will increase as the number of mutually recursive functions grows. Empirical evidence suggests that this is indeed the case.

We ran two kinds of tests: Firstly, we tried to see how the two variadic fixed-point combinators would perform on a set of mutually-recursive functions, as the input grew. Secondly, we tried to see how the two variadic fixed-point combinators would perform when the number of mutually-recursive functions was increased.

Below, we tabulated the CPU time (in milliseconds, under Petite Chez Scheme running on a dual UltraSPARC-II with 1G RAM) for evaluating the mutually-recursive `even?` and `odd?` procedures for various input values (given by N):

N	`curry-fps`	`NfixN2`
10^3	10	30
10^4	70	250
10^5	760	2,430
10^6	7,500	24,650
10^7	74,810	242,570

In order to test execution speeds as the number of mutually-recursive procedures changed, we created a Scheme procedure that given an integer argument `n` creates n mutually-recursive procedures. We started with the standard definition of *Ackermann's function*:

```
(lambda (a b)
 (cond ((zero? a) (+ b 1))
       ((zero? b) (ack (- a 1) 1))
       (else (ack (- a 1)
                  (ack a (- b 1))))))
```

We duplicated this definition n times, numbering each instance sequentially. Then, for each call to `ack` in the body of each of the instances of `ack`, we randomly select one of the numbered instances. We then use the same n mutually-recursive procedures to time the computation of *Ackermann*(3, 5) using both variadic multiple fixed-point combinators.

The table below lists the CPU time (again, milliseconds, under Petite Chez Scheme running on a dual UltraSPARC-II with 1G RAM) for evaluating mutually-recursive versions of Ackermann's function, as the number of mutually-recursive procedures (given by N) varies.

N	`curry-fps`	`NfixN2`
1	320	570
2	390	1,160
3	390	1,910
10	770	14,280
20	1,240	51,670
30	1,770	112,050
100	5,640	1,199,370
200	12,650	5,020,170
300	18,970	$>$ 12 hours
1000	99,929	—

It is clear that using either `NfixN2` or `curry-fps` to define a large set of mutually-recursive procedures will result in performance penalties that are proportional to the number of procedures, however it is also clear that using the built-in support for working with lists of arguments (i.e., `apply`, `map` and variadic procedures) is superior to picking individual elements explicitly.

6. CONCLUSION

We presented a systematic construction for an applicative-order variadic multiple fixed-point combinator in Scheme. Starting out with Curry's singular fixed-point combinator, we considered the extension to multiple fixed-point equations, parameterized by the number of equations and the index of the fixed point. Using variadic `lambda`-expressions, and the elementary procedures for working with lists of arguments (`apply`, `map`), we were able to formulate four rules, or conventions, for converting Scheme expressions written with meta-linguistic ellipses ('$\cdots$') into actual Scheme code. This enabled us to define a multiple fixed-point combinator without having to specify the number of equations. The value returned by this procedure is the list of all the fixed points.

This variadic multiple fixed-point combinator directly corresponds to the multiple fixed-point extension of Curry's singular fixed-point combinator.

7. ACKNOWLEDGMENTS

This work was supported in part by the Danish Research Academy. The author is grateful to BRICS[2] for hosting him and for providing a stimulating environment. Thanks are also due to Olivier Danvy for his invaluable suggestions and encouragement, and to the referees, whose comments and suggestions on an earlier version of the manuscript did much to improved the paper.

APPENDIX

A. SAMPLE RUN

[2]Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.

We encode the procedures `even?` and `odd?` in Scheme (Exprs. (3)) using the variadic `curry-fps` procedure (defined in Expr. (10)) for computing the list of multiple fixed points.

```
;;; defining the Even functional:
> (define E
    (lambda (even? odd?)
      (lambda (n)
        (if (zero? n) #t ; return Boolean True
            (odd? (- n 1))))))
;;; Defining the Odd functional:
> (define O
    (lambda (even? odd?)
      (lambda (n)
        (if (zero? n) #f ; return Boolean False
            (even? (- n 1))))))
;;; Finding the list of fixed points:
> (define list-even?-odd? (curry-fps E O))
> (define even? (car list-even?-odd?))
> (define odd? (cadr list-even?-odd?))
> (even? 6)
#t
> (odd? 4)
#f
```

B. REFERENCES

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, McGraw-Hill Book Company, Second edition, 1996.

[2] Hendrik P. Barendregt. Functional Programming and the λ-calculus. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 7, pages 323 – 363. MIT Press, Cambridge, Massachusetts, 1990.

[3] Hans Bekič. *Programming Languages and Their Definition*. Number 177 in Lecture Notes in Computer Science. Springer-Verlag, 1984.

[4] Haskell B. Curry, Robert Feys, and William Craig. *Combinatory Logic*, volume I. North-Holland Publishing Company, 1958.

[5] Dyadic Systems, Limited. Dyalog APL. `http://www.dyadic.com/`.

[6] Daniel P. Friedman and Matthias Felleisen. *The Little LISPer*. Science Research Associates, Inc, 1986.

[7] Peter J. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6:308–320, 1964.

[8] Christian Queinnec. *LISP In Small Pieces*. Cambridge University Press, 1996.

How to Write Seemingly Unhygienic and Referentially Opaque Macros with Syntax-rules

Oleg Kiselyov*
Software Engineering, Naval Postgraduate School, Monterey, CA 93943
oleg@pobox.com, oleg@acm.org

Abstract

This paper details how folklore notions of hygiene and referential transparency of R5RS macros are defeated by a systematic attack. We demonstrate syntax-rules that seem to capture user identifiers and allow their own identifiers to be captured by the closest lexical bindings. In other words, we have written R5RS macros that accomplish what commonly believed to be impossible. We build on the the fundamental technique by Petrofsky of extracting variables from arguments of a macro. The present paper generalizes Petrofsky's idea to attack referential transparency.

This paper also shows how to overload the `lambda` form. The overloaded `lambda` acts as if it was infected by a virus, which propagates through the `lambda`'s body infecting other `lambda`s in turn. The virus re-defines the macro being camouflaged after each binding. This redefinition is the key insight in achieving the overall referential opaqueness. Although we eventually subvert all binding forms, we preserve the semantics of Scheme as given in R5RS.

The novel result of this paper is a demonstration that although R5RS macros are deliberately restricted in expressiveness, they still wield surprising power. We have exposed faults and the lack of precision in commonly held informal assertions about syntax-rule macros, and pointed out the need for proper formalization. For a practical programmer this paper offers an encouragement: more and more powerful R5RS macros turn out to be possible.

1 Introduction

One of the most attractive and unsurpassed features of Lisp and Scheme is the ability to greatly extend the syntax of the core language and to support domain-specific notations [14]. These syntactic extensions are commonly called macros. A special part of a Lisp/Scheme system, a macro-expander, systematically reduces the extended language to the core one.

A naive macro system that merely finds syntactic extensions and replaces them with their expansions can corrupt variable bindings and break the block structure of the program. For instance, free identifiers in user code may be inadvertently captured by macro-generated bindings, which leads to insidious bugs. This danger is very well documented, for example in [8], [1]. Lisp community has developed techniques [1] that help make macros safer, but they rely on efforts and care of an individual macro programmer. The safety is not built into the system. Furthermore, the techniques complicate the macro code and make it more bug-prone.

Scheme community has recognized the danger of the naive macro expansion to the block structure of Scheme code. The community endeavored to develop a macro system that is safe and respectful of the lexical scope by default. In limited circumstances, exceptions to the block-structure-preserving policy of macros are useful and can be allowed. These exceptions however should be statically visible. A number of experimental macro systems with the above properties have been built ([8], [9], [1], [2], [4], [13]). The least powerful and the most restrictive set of common features of these macro systems has been standardized in R5RS [7]. An earlier version of that system has been mentioned in the previous Scheme report, R4RS, and expounded in [3]. The R5RS macro system permits no exceptions to the safety policy (so-called, hygiene, see below). Furthermore, R5RS macros are specified in a restricted pattern language, which gives the macros another name: syntax-rules. The pattern language is different from the core language and therefore removes the need for the full Scheme evaluator at macro-expand time. Therefore, R5RS macros are severely limited in their ability. The strict safety policy with no exceptions has lead to claims that "Scheme's hygienic macro system is a general mechanism for defining syntactic transformations that reliably obey the rules of lexical scope" [3]. However, there has been little work in formalizing this assertion. Only [8] took upon the task of proving that the systematical renaming of introduced identifiers indeed guarantees the hygiene condition, *in the macro system* of [8]. The latter is not an implementation of R5RS macros.

Surprising discoveries of R5RS macros' latent power question commonly held beliefs about syntax-rule macros. For example, the paper [3] claims "The primary limitation of the hygienic macro system is that it is thoroughly hygienic, and thus cannot express macros that bind identifiers implicitly.... The loop-until-exit macro that is used as an example of the low-level macro system in the Revised 4 Report is also a non-hygienic macro." In 2001, however, Al Petrofsky did express the loop-until-exit macro in the R5RS system [11] (see also [12] for more discussion). Al Petrofsky's article introduced

*Current affiliation: Fleet Numerical Meteorology and Oceanography Center, Monterey, CA 93943.

Third Workshop on Scheme and Functional Programming. October 3, 2002, Pittsburgh, Pennsylvania, USA.

a general technique, *Petrofsky extraction*, of writing macros that can extract a specific binding from their arguments. Al Petrofsky has also shown how to make such macros nest. The present paper generalizes Petrofsky's ideas to writing of seemingly referentially opaque R5RS macros.

A syntactic extension by its nature introduces a new language, which may differ in some aspects from the core language. Can we write a syntax-rule–based extension that looks like R5RS Scheme but allows seemingly referentially opaque and non-hygienic macros? Can such an extended language still be called R5RS Scheme? At first sight, the answer to both questions is negative. Although R5RS macros are Turing complete [6], they were regarded as "thoroughly hygienic" [3]. Furthermore, the fact that R5RS macros are written in a restricted pattern language rather than in Scheme makes them clearly incapable of certain computations (e.g., concatenating strings or symbols). It is impossible to write an R5RS macro `foo` such that `(foo` *a-symbol b-symbol*`)` expands into a `a-b-symbol`, where the latter is spelt as the concatenation of characters constituting `a-symbol` and `b-symbol`. It is not possible for an R5RS macro to tell if two identifiers have the same spelling. Ostensibly these restrictions were put in place to guarantee and enforce the rules of lexical scope for macros and their expansions (this sentiment was discussed in [1]). In this paper we demonstrate that the power of R5RS macros has been underestimated: We can indeed implement a syntax-rule extension of Scheme that permits seemingly referentially opaque and unhygienic macros [12]. Furthermore, this extended language literally complies with R5RS.

The next section briefly describes the notions of hygiene and referential transparency of macro expansions. Section 3 recalls Petrofsky extraction and its application to writing weakly non-hygienic macros. Section 4 introduces the key idea that re-defining a macro after each binding leads to the overall referential opaqueness. Carrying out such redefinitions requires overloading of all Scheme binding forms, in particular, the `lambda` itself. Section 5 accomplishes this overloading with the help of Petrofsky extraction. We demonstrate an R5RS macro that looks exactly like a careless, referentially opaque Lisp-style macro. The end result is a *library* syntax `let-leaky-syntax` that lets a programmer define a syntax-rule macro and designate a free identifier from that macro for capture by local bindings. The final section discusses what it all means: for macro writers, for macro users, and for programming language researchers.

2 Hygiene and Referential Transparency of Macro Expansions

This section introduces the terminology and the working examples that are used throughout the paper. We will closely follow [8] in our terminology. A syntactic extension, or a macro (invocation), is a phrase in an extended language distinguished by its leading token, or keyword. During the macro-expansion process the extended language is eventually reduced to the core Scheme, in one or several steps. One step in this transformation of a syntactic extension is called a (macro-) expansion step or a transcription step. A syntactic transform function (a.k.a. a macro (transformer)) is a function defined by the macro writer that expands the class of syntactic extensions introduced by the same keyword. A transcription step, which is an application of a transformer to a syntactic extension, yields a phrase in the core language or another syntactic extension. The latter will be expanded in turn. The result of an expansion step may contain identifiers that were not present in the original syntactic extension; we will call them generated identifiers.

A macro system is called hygienic, in the general sense, if it avoids inadvertent captures of free variables through systematic renaming [3]. The free variables in question can be either generated variables, or variables present in macro invocations (i.e., user variables). A narrowly defined hygiene is avoiding the capture of user variables by generated bindings. The precise definition, a hygiene condition for macro expansions (HC/ME), is given in [8]: "Generated identifiers that become binding instances in the completely expanded program must only bind variables that are generated at the same transcription step." If a macro system on the other hand specifically avoids capturing of generated identifiers, the latter always refer to the bindings that existed when the macro transformer was defined rather to the bindings that may exist at the point of macro invocations. This property is often called referential transparency.

The rest of the present section expounds sample R5RS macros chosen to illustrate HC/ME and referential transparency. We will be using the examples in the rest of the paper.

The HC/ME condition demands that bindings introduced by macros should not capture free identifiers in macro arguments. Let us define a sample macro `mbi` such that `(mbi body)` will expand into `(let ((i 10)) body)`. In the pattern language of R5RS macros, the definition reads:

```
(define-syntax mbi
 (syntax-rules ()
    ((mbi body) (let ((i 10)) body))))
```

A naive, non-hygienic expansion of `(mbi (* 1 i))` would have produced `(let ((i 10)) (* 1 i))`. The generated binding of `i` would have captured the free variable `i` occurring in the macro invocation. A hygienic expansion prevents such capture through a systematic renaming of identifiers. Therefore,

```
(let ((i 1)) (mbi (* 1 i)))
```

actually expands to

```
(let ((i~2 1))
  (let ((i~5 10)) (* 1 i~2))
```

and gives the result 1. The identifier `i~2` is different from `i~5`: we will call them identifiers of different *colors*.

The referential transparency facet demands that generated free identifiers should not be captured by local bindings that surround the expansion. To be more precise, if a macro expansion generates a free identifier, the identifier refers to the binding occurrence in the environment of the macro's definition. For example, given the definitions

```
(define foo 1)
(define-syntax mfoo
  (syntax-rules ()
    ((mfoo) foo)))
```

The form `(let ((foo 2)) (mfoo))` expands into

```
(let ((foo~1 2))
    foo)
```

and yields 1 when evaluated. The local `let` binds `foo` of a different color, and therefore, does not capture `foo` generated by the macro `mfoo`.

3 Petrofsky Extraction

In 2001 Al Petrofsky posted an article [11] that demonstrated how to circumvent a weak form of hygiene. The present paper generalizes Petrofsky's idea to attack referential transparency. For completeness and reference this section systematically derives the Petrofsky technique. We aim to write a macro `mbi` so that `(mbi 10 body)` expands into `(let ((i 10)) body)` and the binding of `i` captures free occurrences of `i` in the `body`. We assume that there are no other bindings of `i` in the scope of `(mbi 10 body)`, or `i` was defined early in the global scope and was not re-defined since. This assumption is the distinction between the weak hygiene and the true one.

Developing even weakly non-hygienic macros is challenging. We cannot just write

```
(define-syntax mbi
  (syntax-rules ()
    ((_ val body) (let ((i val)) body))))
```

because `(mbi 10 (* 1 i))` will expand into

```
(let ((i~5 10)) (* 1 i))
```

where `i` in `(* 1 i)` refers to the top-level binding of `i` or remains undefined. However, we can explicitly pass a macro the identifier to capture:

```
(define-syntax mbi-i
  (syntax-rules ()
    ((_ i val body) (let ((i val)) body))))
```

In that case,

```
(mbi-i i 10 (* 1 i))
```

expands into

```
(let ((i 10)) (* 1 i))
```

and the capture occurs. Hence to circumvent the hygiene in the weak sense, we only need to find a way to convert an invocation of `mbi` into an invocation of `mbi-i`. The macro `mbi-i` requires the explicit specification of the identifier to capture – which we can get by extracting the identifier `i`, *together* with its color, from the argument of `mbi`. That is the essence and the elegance of the Petrofsky's idea. Once we have the rightly colored occurrence of `i`, we can use it in the binding form and effect the capture.

The extraction of colored identifiers from a form is done by a macro `extract`, Fig. 1. This macro is the workhorse of the hygiene circumvention strategy. We also need a macro that extracts several identifiers, `extract*` (Fig. 2). Now we can define:

```
(define-syntax mbi-dirty-v1
  (syntax-rules ()
    ((_ _val _body)
     (let-syntax
         ((cont
           (syntax-rules ()
             ((_ (symb) val body)
              (let ((symb val)) body) ))))
       (extract i _body (cont () _val _body))))))
```

so that

```
(mbi-dirty-v1 10 (* 1 i))
```

expands into

```
(let ((i~11 10)) (* 1 i~11))
```

and evaluates to 10, as expected.

The macro `mbi-dirty-v1` seems to do the job, but it has a flaw. It does not nest:

```
(mbi-dirty-v1 10
     (mbi-dirty-v1 20 (* 1 i)))
```

expands into

```
(let ((i~16 10))
     (let ((i~17~25~28 20)) (* 1 i~16)))
```

and evaluates to 10 rather than to 20 as we might have hoped. The outer invocation of `mbi-dirty-v1` creates a binding for `i` – which violates the weak hygiene assumption. Petrofsky [11] has shown how to overcome this problem as well: we need to re-define `mbi-dirty-v1` in the scope of the new binding to `i`. Hence we need a macro that re-defines itself in its own expansion. We however face a problem: If the outer invocation of `mbi-dirty-v1` re-defines itself, this redefinition has to capture the inner invocation of `mbi-dirty-v1`. We already know how to do that, by extracting the colored identifier `mbi-dirty-v1` from the outer macro's body. We need thus to extract two identifiers: `i` and `mbi-dirty-v1`. We arrive at the following code:

```
; A macro that re-defines itself in its expansion:
;  (mbi-dirty-v2 val body)
; expands into
;  (let ((i val)) body)
; and also re-defines itself in the scope of body.
; myself-symb, i-symb are colored ids extracted
; from the 'body'

(define-syntax mbi-dirty-v2
  (syntax-rules ()
    ((_ _val _body)
     (letrec-syntax
         ((doit          ; continuation from extract*
           (syntax-rules ()
             ((_ (myself-symb i-symb) val body)
              (let ((i-symb val)) ; first bind 'i'
                (let-syntax       ; re-define oneself
                    ((myself-symb
                      (syntax-rules ()
                        ((_ val__ body__)
                         (extract*
                           (myself-symb i-symb)
                           body__
                           (doit () val__ body__))))))
                  body))))))
       (extract* (mbi-dirty-v2 i) _body
                 (doit () _val _body))))))
```

Therefore

```
(mbi-dirty-v2 10
   (mbi-dirty-v2 20 (* 1 i)))
```

now expands to

```
;    extract SYMB BODY CONT
; BODY is a form that may contain an occurrence of an identifier that
; refers to the same binding occurrence as SYMB.
; CONT is a form of the shape (K-HEAD K-IDL . K-ARGS)
; where K-IDL and K-ARGS are S-expressions representing lists or the
; empty list.
; The macro extract expands into
;   (K-HEAD (extr-id . K-IDL) . K-ARGS)
; where extr-id is the extracted colored identifier. If the symbol SYMB does
; not occur in BODY at all, extr-id is identical to SYMB.


(define-syntax extract
  (syntax-rules ()
    ((_ symb body _cont)
      (letrec-syntax
        ((tr
           (syntax-rules (symb)
             ; Found our 'symb' -- exit to continuation
             ((_ x symb tail (cont-head symb-l . cont-args))
              (cont-head (x . symb-l) . cont-args))
             ((_ d (x . y) tail cont)   ; if body is a composite form,
              (tr x x (y . tail) cont)) ; look inside
             ((_ d1 d2 () (cont-head  symb-l . cont-args))
              (cont-head (symb . symb-l) . cont-args)) ; symb does not occur
             ((_ d1 d2 (x . y) cont)
              (tr x x y cont)))))
        (tr body body () _cont)))))
```

Figure 1: Macro `extract`: Extract a colored identifier from a form

```
;    extract* SYMB-L BODY CONT
; where SYMB-L is the list of identifiers to extract, and BODY and CONT
; has the same meaning as in extract, see above.
;
; The macro extract* expands into
;   (K-HEAD (extr-id-l . K-IDL) . K-ARGS)
; where extr-id-l is the list of extracted colored identifiers. The extraction
; itself is performed by the macro extract.

(define-syntax extract*
  (syntax-rules ()
    ((_ (symb) body cont)      ; only one id: use extract to do the job
     (extract symb body cont))
    ((_ _symbs _body _cont)
     (letrec-syntax
         ((ex-aux               ; extract id-by-id
           (syntax-rules ()
             ((_ found-symbs () body cont)
              (reverse () found-symbs cont))
             ((_ found-symbs (symb . symb-others) body cont)
              (extract symb body
                       (ex-aux found-symbs symb-others body cont)))
             ))
          (reverse              ; reverse the list of extracted ids
           (syntax-rules ()     ; to match the order of SYMB-L
             ((_ res () (cont-head () . cont-args))
              (cont-head res . cont-args))
             ((_ res (x . tail) cont)
              (reverse (x . res) tail cont)))))
       (ex-aux () _symbs _body _cont)))))
```

Figure 2: Macro `extract*`: Extract several colored identifiers from a form

```
(let ((i~26 10)) (let ((i~52 20)) (* 1 i~52)))
```

and evaluates to 20.

The macro `mbi-dirty-v2` is still only weakly unhygienic. If we evaluate

```
(let ((i 1))
  (mbi-dirty-v2 10 (* 1 i)))
```

we obtain

```
(let ((i 1)) (let ((i~3~22~29 10)) (* 1 i)))
```

which evaluates to 1 rather than 10.

4 Towards the Referential Opaqueness: a `mylet` Form

In this section, we attack referential transparency by writing a macro that seemingly allows free identifiers in its expansion to be captured by the closest lexical binding. To be more precise, we want to write a macro `mfoo` that expands in an identifier `foo` in such a way so that the form `(let ((foo 2)) (let ((foo 3)) (list foo (mfoo))))` would evaluate to the list `(3 3)`. The key insight is a shift of focus from the macro `mfoo` to the binding form `let`. The macro `mfoo` is trivial:

```
(define-syntax mfoo
  (syntax-rules ()
    ((mfoo) foo)))
```

We will concentrate on re-defining the binding form to permit a referentially opaque capture. To make such redefinition easier, we introduce in this section a custom binding form `mylet`. The next section shall show how to make the regular `let` act as `mylet`.

The goal of this section is therefore developing a binding form `mylet` so that `(mylet ((foo 2)) (mylet ((foo 3)) (list foo (mfoo))))` would evaluate to the list `(3 3)`. To make this possible, the expression should expand as follows:

```
(let ((foo 2))
  (define-syntax-mfoo-to-expand-into-foo)
  (re-define-mylet-to-account-for-
      redefined-foo-and-mfoo)
  (let ((foo 3))
    (define-syntax-mfoo-to-expand-into-foo)
    (re-define-mylet-to-account-for-
        redefined-foo-and-mfoo)
    (list foo (mfoo))
 ))
```

Different bindings of a variable are typeset in different fonts. The expansion of the form `mylet` therefore binds `foo` and then re-defines the macro `mfoo` within the scope of the new binding. This `mfoo` will generate the identifier `foo` that refers to that local binding. The redefinition of `mfoo` after a binding is the key insight. It makes it possible for the expansion of the targeted macro to contain identifiers whose bindings are not inserted by the same macro. The process of defining and redefining macros during the expansion of `mylet` looks similar to the process described in the previous Section. Therefore, we take the macro `mbi-dirty-v2` as a prototype for the design of `mylet`. A generator (which helps us define and re-define the macro `mfoo`) and the macro `mylet` are given on Fig. 3. With these definitions, `(mylet ((foo 2)) (mylet ((foo 3)) (list foo (mfoo))))` expands to `((lambda (foo~47) ((lambda (foo~92) (list foo~92 foo~92)) 3)) 2)` and evaluates to `(3 3)`. The result demonstrates that `(mfoo)` indeed expanded to `foo` that was captured by the local binding. The macro `mfoo` seems to have inserted an opaque reference to the binding of `foo`. Because `mylet` constantly re-generates itself, it nests. The following test demonstrates the nesting and the capturing by the expansion of `(mfoo)` of the closest *lexical* binding:

```
(mylet ((foo 3))
    (mylet ((thunk (lambda () (mfoo))))
     (mylet ((foo 4)) (list foo (mfoo) (thunk)))))
```

This expression evaluates to `(4 4 3)`. The expansion of `(mfoo)` within the closure `thunk` refers to the variable `foo` that was lexically visible at that time.

5 Achieving the Referential Opaqueness: Redefining All Binding Forms

The previous section showed that we can indeed write a seemingly referentially opaque R5RS macro, if we resort to custom binding forms. R5RS does not prohibit us however from re-defining the standard binding forms `let`, `let*`, `letrec` and `lambda` to suit our nefarious needs. We need to 'overload' just one form: the fundamental binding form `lambda` itself.

This overloading is done by a macro `defile`, which defiles its body (Appendix B). It is worth noting a few fragments from the macro's long code. The first one

```
(letrec-syntax
  ...
   (lambda-native   ; capture the native lambda
      (syntax-rules ()
        ((_ . args) (lambda . args))))
```

does what it looks like: it captures the native `lambda`, which is needed to effect bindings. Another fragment is:

```
(letrec-syntax
...
  (let-symb     ; R5RS definition of let
    (syntax-rules ()
      ((_ . args)
       (glet (let-symb let*-symb letrec-symb
              lambda-symb) . args))))
```

A top-level macro `glet` (Appendix A) is a `let` with an extra first argument. This argument is the "environment", the list of custom-bound `let` and `lambda` identifiers for use in the macro expansion. The definition of `glet` is taken from R5RS verbatim, with the pattern modified to account for the extra first argument.

```
(define-syntax glet
  (syntax-rules ()
    ((_ (let let* letrec lambda) ; the extra arg
        ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...) val ...))
    ((_ (let let* letrec lambda)
        tag ((name val) ...) body1 body2 ...)
     ((letrec
       ((tag (lambda (name ...) body1 body2 ...)))
       tag) val ...))))
```

```
; Macro: make-mfoo NAME SYMB BODY
; In the scope of BODY, define a macro NAME that expands into an identifier SYMB

(define-syntax make-mfoo
  (syntax-rules ()
    ((_ name symb body)
      (let-syntax
         ((name
           (syntax-rules ()
             ((_) symb))))
        body))))


;   (mylet ((var init)) body)
; expands into
;   (let ((var init)) body')
; where body' is the body wrapped in the re-definitions of mylet and the macro mfoo.

(define-syntax mylet
  (syntax-rules ()
    ((_ ((_var _init)) _body)
     (letrec-syntax
       ((doit                   ; The continuation from extract*
         (syntax-rules ()       ; mylet-symb, etc. are extracted from body
           ((_ (mylet-symb mfoo-symb foo-symb) ((var init)) body)
            (let ((var init))   ; bind the 'var' first
              (make-mfoo mfoo-symb foo-symb  ; now re-generate the macro mfoo
                  (letrec-syntax
                      ((mylet-symb       ; and re-define myself
                        (syntax-rules ()
                          ((_ ((var_ init_)) body_)
                           (extract* (mylet-symb mfoo-symb foo-symb) (var_ body_)
                                     (doit () ((var_ init_)) body_))))))
                   body)))
           ))))
       (extract* (mylet mfoo foo) (_var _body)
                 (doit () ((_var _init)) _body))))))
```

Figure 3: Macros `make-mfoo` and `mylet`

The macro `glet` therefore relates the `let` form and the `lambda` precisely as R5RS does; `glet` however substitutes our custom-bound `lambda`. Finally, the overloaded `lambda` is defined as follows:

```
(letrec
...
 (lambda-symb      ; re-defined, infected lambda
   (syntax-rules ()
     ((_ _vars _body)
      (letrec-syntax
        ((doit (syntax-rules ()
          ((_ (mylet-symb mylet*-symb
               myletrec-symb mylambda-symb
               mymfoo-symb myfoo-symb)
               vars body)
           (lambda-native vars
             (make-mfoo mymfoo-symb myfoo-symb
               (do-defile  ; proliferate
                (mylet-symb mylet*-symb
                 myletrec-symb mylambda-symb
                 mymfoo-symb myfoo-symb)
                body)))))))
       (extract* (let-symb let*-symb letrec-symb
                  lambda-symb mfoo-symb foo-symb)
                 (_vars _body)
                 (doit () _vars _body))))))
```

We are relying on the previously captured `lambda-native` to create bindings. After that we immediately redefine all our macros in the updated environment. The corrupted `lambda` acts as if it were infected by a virus: every mentioning of `lambda` "transcribes" the virus and causes it to spread to other binders within the body.

The following are a few excerpts from the defile macro regression tests. An expression

```
(defile
   (let ((foo 2)) (list (mfoo) foo)))
```

expands into

```
((lambda (foo~186) (list foo~186 foo~186)) 2)
```

and predictably evaluates to `(2 2)`. The expansion of `(mfoo)` has indeed captured a locally-bound identifier. All the infected `lambda`s are gone: the expansion result is the regular Scheme code. Furthermore,

```
(defile
   (let ((foo 2))
     (let ((foo 3) (bar (list (mfoo) foo)))
       (list foo (mfoo) bar))))
```

evaluates to `(3 3 (2 2))` and

```
(defile
   (let ((foo 2))
     (list
      ((letrec
         ((bar (lambda () (list foo (mfoo))))
          (foo 3))
        bar))
       foo (mfoo))))
```

to `((3 3) 2 2)`. The defiled `let` and `letrec` indeed act precisely as the standard ones. Finally,

```
(defile
  (let* ((foo 2)
         (i 3)
         (foo 4)
         ; will capture binding of foo to 4
         (ft (lambda () (mfoo)))
         (foo 5)
         ; will  capture the arg of ft1
         (ft1 (lambda (foo) (mfoo)))
         (foo 6))
    (list foo (mfoo) (ft) (ft1 7) '(mfoo))))
```

evaluates to the expected `(6 6 4 7 (mfoo))`. In all these examples, the expansion of `(mfoo)` captures the closest (local) lexical binding of the variable `foo`. All the examples run with the Bigloo 2.4b interpreter and compiler and with Scheme48.

We must point out that the defiled examples behave as if `(mfoo)`, unless quoted, were just the identifier `foo`. In other words, as if `mfoo` were defined as a *non-hygienic*, referentially opaque macro

```
(define-macro (mfoo) foo)
```

To be able to capture a generated identifier by a local binding, we need to know the name of that identifier and the name of a macro that generates it. We also need to effectively wrap the `defile` macro around victim's code. We can do that explicitly as in the examples above. We can also accomplish the wrapping implicitly, e.g., by re-defining the top-level `let` or other suitable form so as to insert the invocation of `defile` at the right spot. It goes without saying that we assume no bindings to the identifiers `foo`, `mfoo`, `let`, `letrec`, `let*`, and `lambda` between the point the macro `defile` is defined and the point it is invoked.

It is possible to remove the dependence of the macro `defile` on ad hoc identifiers such as `foo` and `mfoo`. We can pass the targeted macro and the identifier to be captured by the closest lexical binding as arguments to `defile`. We arrive at a form `let-leaky-syntax` (Appendix C), which is illustrated by the following two examples. An expression

```
(let-leaky-syntax
 bar
 ((mbar
   (syntax-rules () ((_ val) (+ bar val)))))
 (let ((bar 1)) (let ((bar 2)) (mbar 2))))
```

evaluates to 4, whereas

```
(let-leaky-syntax
  quux
  ((mquux (syntax-rules ()
     ((_ val) (+ quux quux val)))))
  (let* ((bar 1) (quux 0) (quux 2)
         (lquux (lambda (x) (mquux x)))
         (quux 3)
         (lcquux (lambda (quux) (mquux quux))))
    (list (+ quux quux) (mquux 0) (lquux 2)
          (lcquux 5))))
```

evaluates to the list `(6 6 6 15)`. The form `let-leaky-syntax` is similar to `let-syntax`. The former takes an additional first argument, an identifier from the body of the defined syntax-rules. This designated identifier will be captured by the closest lexical binding within the body of `let-leaky-syntax`. The examples show that the variable is captured indeed. In

particular, the macro `mquux` in the last example expands to an expression that adds the value of an identifier `quux` twice to the value of the `mquux`'s argument. Because the identifier `quux` was designated for capture by the closest local binding, a procedure `(lambda (quux) (mquux quux))` effectively triples its argument.

We have thus demonstrated the syntax form `let-leaky-syntax` that defines a macro with a specific variable excepted from the hygienic rules. The form `let-leaky-syntax` is a *library* syntax, developed exclusively with R5RS (hygienic) macros.

6 Discussion

In this section we will discuss the implications of the defile macro. First however we have to assure the reader that `defile` is legal: it fully complies with R5RS and does not rely on unspecified behavior. Indeed, the macro defile is written entirely in the pattern language of R5RS. Re-binding of syntax keywords `lambda`, `let`, `let*`, and `letrec` is not prohibited by R5RS. On the contrary, R5RS specifically states that there are no reserved keywords, and syntactic bindings may shadow variable bindings and other syntactic bindings. Furthermore, re-defined `let`, `let*`, and `letrec` forms relate to the `lambda` form precisely as the R5RS forms do. The re-defined `lambda` form is also in compliance with its R5RS description ([7], Section 4.1.4).

One can argue that our re-defined `lambda` leads to a violation of the constraint that R5RS places on the macro system: "If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that may surround the use of the macro." This paragraph however applies exactly as it is to the defiled macros. In the code,

```
(define foo 1)
(defile
   (let ((foo 2)) (list (mfoo) foo)))
```

the identifier `foo` inserted by the expansion of the macro `mfoo` indeed refers to the binding of `foo` that was visible when the macro `mfoo` was defined. The twist is that the definition of the macro `mfoo` happened right after the local binding of `foo`. Despite `mfoo` being an R5RS, referentially transparent macro, the *overall result* is equivalent to the expansion of a referentially opaque macro.

The macro defile indeed has to surround the victim's code. One can therefore object if we merely create our own 'little language' that resembles Scheme but does not guarantee referential transparency of macro expansions. However, such a little language was presumed impossible with syntax rules [2][3]! Any macro by definition extends the language. The extended language is still expected to obey certain constraints. The impetus for hygienic macros was specifically to create a macro system with guaranteed hygienic constraints. Although syntax-rules are Turing complete, certain computations, for example, determining if two identifiers are spelled the same, are outside of their scope. It was a common belief therefore that syntax-rules are thoroughly hygienic [3].

To be more precise, the argument that syntax-rules cannot in principle implement macros such as `let-leaky-syntax` was informally advanced in [2]. That paper described a macro-expansion algorithm that is used in several R5RS Scheme systems, including Bigloo. Incidentally, the algorithm accounts for the possibility that the binding forms `lambda` and `let-syntax` may be redefined by the user. The paper [2] *informally* argues that the algorithm satisfies two hygiene conditions: (1) "It is impossible to write a high-level macro that inserts a binding that can capture references other than those inserted by the macro," and (2) "It is impossible to write a high-level macro that inserts a reference that can be captured by bindings other than those inserted by the macro." Unfortunately, the paper does not state the conditions with sufficient precision, which precludes a formal proof. The notion of 'inserting a binding' is particularly vague. The common folklore interpretation of the conditions is that generated bindings can capture only the identifiers that are generated at the same transcription step. Had this interpretation been true, `let-leaky-syntax` would have been impossible. However, the interpretation is false and Petrofsky's loop macro is a counter-example [12]. Several examples in Section 4 demonstrated the capture of generated identifiers *across* transcription steps.

It is interesting to ask if it is possible to create a macro system that is provable hygienic, which provably does not permit tricks such as the one in this paper. The paper [8] showed that if we do not allow macros to expand into the definitions of other macros, we can design a macro system that is provably hygienic. A MacroML paper [5] claimed that being generative *seems* to be a necessary condition for a macro extension to maintain strong invariants (static typing, in the context of MacroML). A generative macro can build forms from its arguments but cannot deconstruct or inspect its arguments.

We conclude that the subject of macro hygiene is not at all decided, and more research is needed to precisely state what hygiene formally means and which precisely assurances it provides.

For a practical programmer, we offer the `let-leaky-syntax` library form. The form lets the programmer write a new class of powerful syntactic extensions with the standard R5RS syntax-rules, without resorting to lower-level macro facilities. In general, the practical macro programmer will hopefully view the conclusions of this paper as an encouragement. We should realize the informal and narrow nature of many assertions about R5RS macros. We should not read into R5RS more than it actually says. Thus we can write more and more expressive macros than we were previously led to believe [12].

Acknowledgment

I am greatly indebted to Al Petrofsky for numerous discussions, which helped improve both the content and the presentation of the paper. Special thanks are due to Alan Bawden for extensive comments and the invaluable advice. I would like to thank Olin Shivers and the anonymous reviewers for many helpful comments and suggestions. This work has been supported in part by the National Research Council Research Associateship Program, Naval Postgraduate School, and the Army Research Office under contracts 38690-MA and 40473-MA-SP.

References

[1] Alan Bawden and Jonathan Rees. Syntactic closures. In Proc. 1988 ACM Symposium on Lisp and Functional Programming, pp. 86-95.

[2] William Clinger and Jonathan Rees. Macros that work. In Proc. 1991 ACM Conference on Principles of Programming Languages, pp. 155-162.

[3] William Clinger. Macros in Scheme. Lisp Pointers, IV(4):25-28, December 1991.

[4] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. Lisp and Symbolic Computation 5(4):295-326, 1993.

[5] S. Ganz, A. Sabry, W. Taha: Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML. Proc. Intl. Conf. Functional Programming (ICFP'01), pp. 74-85. Florence, Italy, September 3-5 (2001).

[6] Erik Hilsdale and Daniel P. Friedman. Writing macros in continuation-passing style. Scheme and Functional Programming 2000. September 2000.

[7] R. Kelsey, W. Clinger, J. Rees (eds.), Revised5 Report on the Algorithmic Language Scheme, J. Higher-Order and Symbolic Computation, Vol. 11, No. 1, September, 1998.

[8] Eugene E. Kohlbecker Jr., Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In Proc. 1986 ACM Conference on Lisp and Functional Programming, pp. 151-161.

[9] Eugene E. Kohlbecker and M. Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In Proc. 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 77 - 84, 1987.

[10] Al Petrofsky, Oleg Kiselyov. Re: Widespread bug (arguably) in letrec when an initializer returns twice. Messages posted on a newsgroup comp.lang.scheme on May 21, 2001 10:30:34 and 14:56:49 PST. http://groups.google.com/groups?-selm=7eb8ac3e.0105210930.21542605%40posting.google.com http://groups.google.com/groups?-selm=87ae468j7x.fsf%40app.dial.idiom.com

[11] Al Petrofsky. How to write seemingly unhygienic macros using syntax-rules. A message posted on a newsgroup comp.lang.scheme on November 19, 2001 01:23:33 PST. http://groups.google.com/groups?-selm=87oflzcdwt.fsf%40radish.petrofsky.org

[12] Al Petrofsky. Re: Holey macros! (was Re: choice for embed-
ding Scheme implementation?). A message posted on a newsgroup comp.lang.scheme on May 22 2002 10:21:31 -0700. http://groups.google.com/groups?-selm=874rh0p084.fsf%40radish.petrofsky.org

[13] Jonathan A. Rees. Implementing lexically scoped macros. Lisp Pointers. 'The Scheme of Things' (column), 1993.

[14] Olin Shivers. A universal scripting framework, or Lambda: the ultimate 'little language'. In "Concurrency and Parallelism, Programming, Networking, and Security," Lecture Notes in Computer Science 1179, pp 254-265, Editors Joxan Jaffar and Roland H. C. Yap, 1996, Springer.

Appendix A

The following are definitions of `let`, `let*` and `letrec` taken almost verbatim from R5RS. The only difference is in custom-bound `let`, `let*`, `letrec` and `lambda` identifiers, which we explicitly pass to the glet macros in the first argument.

```
(define-syntax glet
  (syntax-rules ()
    ((_ (let let* letrec lambda)
        ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...) val ...))
    ((_ (let let* letrec lambda)
        tag ((name val) ...) body1 body2 ...)
     ((letrec ((tag (lambda (name ...) body1 body2 ...)))
              tag) val ...))))

(define-syntax glet*
  (syntax-rules ()
    ((_ mynames () body1 body2 ...)
     (let () body1 body2 ...))
    ((_ (let let* letrec lambda)
        ((name1 val1) (name2 val2) ...) body1 body2 ...)
     (let ((name1 val1))
       (let* ((name2 val2) ...) body1 body2 ...)))))

; A shorter implementations of letrec [10]
(define-syntax gletrec
  (syntax-rules ()
    ((_ (mlet let* letrec lambda)
        ((var init) ...) . body)
     (mlet ((var 'undefined) ...)
       ; the native let will do fine here
       (let ((temp (list init ...)))
         (begin (set! var (car temp))
                (set! temp (cdr temp))) ...
              (let () . body))))))
```

Appendix B[1]

```
; This macro defiles its body.
; It overloads all the let-forms and the lambda, and defines a non-hygienic macro 'mfoo'. Whenever any
; binding is introduced, the let-forms, the lambdas and 'mfoo' are re-defined. The overloaded lambda acts
; as if it were infected by a virus, which keeps spreading within lambda's body to infect other lambda's there.

(define-syntax defile
  (syntax-rules ()
    ((_ dbody)
     (letrec-syntax
         ((do-defile
           (syntax-rules ()     ; all the overloaded identifiers
             ((_ (let-symb let*-symb letrec-symb lambda-symb mfoo-symb foo-symb) body-to-defile)
              (letrec-syntax
                  ((let-symb            ; R5RS definition of let
                    (syntax-rules ()
                      ((_ . args)
                       (glet (let-symb let*-symb letrec-symb lambda-symb)
                             . args))))

                   (let*-symb           ; Redefinition of let*
                    (syntax-rules ()
                      ((_ . args)
                       (glet* (let-symb let*-symb letrec-symb lambda-symb)
                              . args))))

                   (letrec-symb         ; Redefinition of letrec
                    (syntax-rules ()
                      ((_ . args)
                       (gletrec (let-symb let*-symb letrec-symb lambda-symb)
                              . args))))

                   (lambda-symb         ; re-defined, infected lambda
                    (syntax-rules ()
                      ((_ _vars _body)
                       (letrec-syntax
                           ((doit
                             (syntax-rules ()
                               ((_ (mylet-symb mylet*-symb myletrec-symb
                                               mylambda-symb mymfoo-symb
                                               myfoo-symb) vars body)
                                (lambda-native vars
                                  (make-mfoo mymfoo-symb myfoo-symb
                                    (do-defile  ; proliferate in the body
                                     (mylet-symb mylet*-symb myletrec-symb
                                                 mylambda-symb
                                                 mymfoo-symb myfoo-symb)
                                     body)))))))
                            (extract* (let-symb let*-symb letrec-symb lambda-symb
                                                mfoo-symb foo-symb)
                                      (_vars _body)
                                      (doit () _vars _body))))))

                   (lambda-native       ; capture the native lambda
                    (syntax-rules () ((_ . args) (lambda . args))))
                   )

                body-to-defile)))))

       (extract* (let let* letrec lambda mfoo foo) dbody
                 (do-defile () dbody))
       ))))
```

[1]The current implementation of the macro `defile` does not corrupt bindings that are created by internal `define`, `let-syntax` and `letrec-syntax` forms. There are no technical obstacles to corrupting those bindings as well. To avoid clutter, the present code does not detect a possible shadowing of the macro `mfoo` by a local binding. The full code with validation tests is available at `http://pobox.com/~oleg/ftp/Scheme/dirty-macros.scm`.

Appendix C

Given below is the implementation of a library syntax `let-leaky-syntax`. It is based on a slightly modified version of the macro `defile`. The latter uses parameters `leaky-macro-name`, `leaky-macro-name-gen`, and `captured-symbol` instead of hard-coded identifiers `mfoo`, `make-mfoo` and `foo`.

```
(define-syntax defile-what
  (syntax-rules ()
    ((_ leaky-macro-name leaky-macro-name-gen captured-symbol dbody)
     (letrec-syntax
       ((do-defile
         ... similar to the defile macro, Appendix B ...
  (extract*
      (let let* letrec lambda
       leaky-macro-name captured-symbol) dbody (do-defile () dbody)) ))))


(define-syntax let-leaky-syntax
  (syntax-rules ()
    ((_ var-to-capture ((dm-name dm-body)) body)
     (let-syntax
       ((dm-generator
          (syntax-rules ()
            ((_ dmg-name var-to-capture dmg-outer-body)
             (let-syntax
               ((dmg-name dm-body))
                dmg-outer-body)))))
      (defile-what
         dm-name dm-generator var-to-capture body)
))))
```

23 things I know about modules for Scheme

Christian Queinnec
Université Paris 6 — Pierre et Marie Curie
LIP6, 4 place Jussieu, 75252 Paris Cedex — France
Christian.Queinnec@lip6.fr

ABSTRACT

The benefits of modularization are well known. However, modules are not standard in Scheme. This paper accompanies an invited talk at the Scheme Workshop 2002 on the current state of modules for Scheme. Implementation is not addressed, only linguistic features are covered.

Cave lector, this paper only reflects my own and instantaneous biases!

1. MODULES

The benefits of modularization within conventional languages are well known. Modules dissociate interfaces and implementations; they allow separate compilation (or at least independent compilation *à la* C). Modules tend to favor re-usability, common libraries and cross language linkage.

Modules discipline name spaces with explicit names exposure, hiding or renaming. Quite often, they also offer qualified naming. These name spaces may cover variables, functions, types, classes, modules, etc.

Just as components, modules may explicit their dependences that is, the other modules they require in order to work properly. Building a complete executable is done via modules linking or module synthesis in case of higher-order modules. Modules dependencies usually form a DAG but mutually dependent modules are sometimes supported.

Proposals for modules for Scheme wildly differ among them (as will be seen) but they usually share some of the following features:

Determinization of the building of modules — For us, this is the main feature that allows users to build a system *S* exactly as it should stand, that is, without any interference of the current system where *S* is developped and/or compiled. This is in contrast with, say the Smalltalk way, where the state of the entire (development) system staid in memory (or in image files) making notoriously difficult to deliver (or even rebuild) stand-alone systems.

Third Workshop on Scheme and Functional Programming. October 3, 2002, Pittsburgh, Pennsylvania, USA.

Interfaces as collection of names — If modules are about sharing, what should be shared ? Values, locations (that is variables), types, classes (and their cortège of accessors, constructors and predicates) ?

The usual answer in Scheme is to share locations with (quite often) two additional properties: *(i)* these locations can only be mutated from the body of their defining modules (this favors block compilation), *(ii)* they should hold functions (and this should be statically (and easily) discoverable). This restricts linking with other (foreign) languages that may export locations holding non-functional data (the `errno` location for instance). This is not a big restriction since modern interfaces (Corba for example) tend to exclusively use functions (or methods). On the good side, this restriction allows for better compilation since non mutated exported functions may be directly invoked or inlined.

Let us remark that values, if staying in an entirely Scheme world, would be sufficient since closures are values giving access to locations (boxes for dialects offering them will equally serve). Since values may be shared via λ-applications, module linking would then be performed by λ-applications without the need for, say, first-class environments [10].

Creating a namespace — A module confines all the global variables defined within its body. This global environment is initially stuffed with locations imported from required modules. Some directives exist to specify the exported locations. A location is specified by the name of its associated variable though renaming (at export or import) often exists.

To use locations is very different from the COMMON LISP way that rather shares symbols, with a read-time resolution, assigning symbols to packages. The Scheme standard is mute with respect to read-time evaluation or macro-characters that both heavily depend on the state of the system while `read`-ing.

Explicitation of required modules — In order to ease the building of large systems, modules should automatically keep track of their dependencies so requiring a module would trigger the requisition of all the other modules it depends on.

Another personal word: I have ported Meroon for years on 12 different Scheme systems [7] and [8, p. 333]. While missing library functions (`last-pair` for instance) or obsolete signatures (binary only `apply` for instance) were always easily accomodated, the most problematic points had always been *(i)* the understanding of how macro-expansion and file compilation interfere and, *(ii)* how to install macros (`define-class` and related) into an REP loop. These problems were often solved by macros or invocations to `eval` thus introducing a new problem: the relationship between `eval` and macros!

Were not for macros, modules would probably be standard in Scheme for a long time. Alas! Macros or syntaxes extend Lisp or Scheme into new enriched languages providing syntactic abstractions that allows programmers to define abbreviations that simplifies the expression of how to solve problems. In mathematics, the "magic of (good) notations" has always transformed delicate semantical problems into syntactical routines (compare Euclide's elements language with usual algebra language). I tend to think that macros are probably the best reason for the survival of the Lisp family but there are the root of the problems for modules!

A module is written in some Scheme extended with various macros. It is expanded into core Scheme before being compiled. Macros do often occur in the module itself to be used in the rest of the module. This clearly requires the expansion engine to convert dynamically the definition of the macro (a text) into an expander (a function): this is the rôle of `eval` and the question is: what is the language used to define macros ? This language is another instace of Scheme possibly enriched with its own various macros. Therefore, in order to understand a module, a "syntax tower" or "macro-expansion tower" [9] must be erected. A module is therefore a tapestry of woven computations performed at different times within different variants of Scheme.

To sum up, a module proposal should solve many problems at the same time among which: share locations, manage their names, determine the exact language a module is written with, maintain module dependencies (for locations and languages) and, in case some invocations to `eval` appear in the built system, what language(s) do they offer?

2. TAXONOMY

In this Section, I will shallowly describe some features of some existing systems, I will then try to establish a rough taxonomy. This Section borrows some material from [8, p. 311]. I will use the term "abbreviation" for macros and syntaxes indifferently while I will only use syntaxes for R5RS hygienic syntaxes.

In the snippets, the "languages" in which they are written appear in right-aligned boxes. A language such as *Scheme+m1* means that the language is Scheme plus the `m1` abbreviation.

The classified systems are Bigloo [13], ChezScheme [14], Gambit [3], MzScheme [4] and Scheme48 [6].

2.1 Gambit

Gambit is probably the easiest to describe since there are no modules per se! Gambit [3] is centered on a REP loop; the predefined library offers the `compile-file` function compiling Scheme to C files that may be further compiled and linked with C means. A `(declare ...)` special form exists to alter the compilation behavior.

The language processed by the REP loop is assumed to be the one in which the file to be compiled is written. When an abbreviation is globally defined, it is immediately available. A global abbreviation defined while compiling a file is only available while compiling the rest of the file. Local abbreviations may be defined along with internal definitions.

Let us give an example of the various possibilities. The first snippet is performed within a first REP loop (whose prompt is `REP1>`). The snippet defines a function `f1` and an abbreviation `m1`: both are immediately usable in the REP loop and in the file to be compiled. The `m1` abbreviation is written in Scheme and may use `f1` at expansion-time. Uses of `m1` may be expanded into invocations to `f1`.

```
REP1> (define (f1 _) _)                                  Scheme
REP1> (define-macro (m1 _)
        ;;(f1 _) is OK
       )
      ;;;(f1 _) and (m1 _) are OK                         Scheme+m1
REP1> (compile-file "f.scm")
      ;;;(f2 _) and (m2 _) are not OK                     Scheme+m1
```

Here is the content of the file, *f.scm*, to be compiled. It defines a function `f2` that may use the `m1` abbreviation (and its expansion-time resource `f1`). Invocations to `f1` and `f2` are of course allowed. Another abbreviation, `m2`, is defined whose scope (though "global") is only the rest of the *f.scm* file. Eventually a function, named `compute`, is defined wrapping a call to `eval`. The result is a compiled module that may require `f1` to run but always provide `f2` to whom will load it.

```
;;;File "f.scm"
(define (f2 _)                                           (Scheme+m1)
  ;;(m1 _) and (f1 _) and (f2 _) are OK
  )
(define-macro (m2 _)
  ;;(m1 _) and (f1 _) are OK
  )                                                      (Scheme+m1)+m2
;;; (m1 _) and (m2 _) and (f1 _) and (f2 _) are OK
(define (compute exp)
  (eval exp) )
```

The language in which is written the *f.scm* file is not defined per se but due to the ambiant language in which `compile-file` is called, it is (scheme+m1). In the absence of compilation, the file just specifies that the `m2` abbreviation extends an unknown language. When compiled with the above conditions, the `m2` abbreviation is considered to be written in Scheme+m1. The body may also invoke `f1` which is indeed present at *f.scm* expansion-time.

The third snippet is run through another REP loop. The compiled *f.scm* file is loaded (a warning will be emitted to mention the absence of `f1`) then a function `f1` is defined (it might not be the same as the previous one defined in REP1) that will be used thoughout the rest of the REP loop. The language of this REP loop is Scheme without any abbreviation. The language accepted by the call to `eval` within the `compute` function is the current language in the current global environment.

```
REP2> (load "f")                                    Scheme
        ;;;(f2 _) is OK
REP2> (define (f1 _) _)
        ;;;(f1 _) and (f2 _) are OK
        ;;;(m1 _) and (m2 _) are not OK
REP2> (compute '(list (f1 _) (f2 _))) ;is OK
```

If REP2 were in fact REP1, `f2` would be loaded as before and `f1` would be redefined, the initial language would be Scheme+m1 instead of raw Scheme and the abbreviation `m1` would be allowed in `compute`.

Repeatibility of compilation is achieved by starting fresh REP loops. The model is simple, there is a single name space. No module dependency is explicit, missing or conflicting locations are caught by the compiler. The initialization order may be specified to the compiler.

The space of names is structured via namespaces offering the possibility of qualified names. A variable may be prefixed by the name of the namespace containing it therefore `m#f` is the variable `f` from namespace `m`. A `##namespace` directive rules, in the current scope, to which namespaces belong the defined variables.

2.2 Bigloo

Bigloo is compiler-centric. The compiler only compiles a single module i.e., some files with a prepended `module` clause. The `module` clause specifies the name of the module as well as some compilation directives. The rest of the file(s) is the source to compile (other files may also be adjoined with the `include` module directive.

The Bigloo compiler creates *.o* files (through C) or *.class* files for Java. When a module is mentioned in some module directives, the module is associated to at least one file and its `module` clause is read. Except when processing inlined exportations, the rest of the module is not read, that is, the `module` clause contains everything needed to compile or import it.

Expansion is performed with (EPS-style [2]) macros and/or (hygienically) with syntaxes. When a global abbreviation is defined, the compiler makes it available for the rest of the module. The language is which the module is written is specified by the `module` clause as well as its imported global environment.

Let us give a first, simple, example of a module, named `M1`. It only exports the immutable unary `f1` function (the arity and the immutability are implied by the shape of the `export` directive). It also defines an abbreviation `m1` whose definition is written in Scheme with the default global environment. This `m1` abbreviation may be used throughout the rest of the module.

```
(module M1 (export (f1 o)))

(define (f1 x) _)                                   Scheme

(define-macro (m1 _)
  ;;(cons _), (car _) are OK
  ;;(f1 _) is not OK
  )                                                 Scheme+m1
;;;(m1 _) is now OK
```

Let us define a second module, named `f`. It exports the `f2` mutable variable (this is implied by the `export` directive) as well as the immutable unary function `compute` (that embeds a call to `eval`) and the immutable nullary function `get-bar`. The body of module `f` defines a global (that is, until the end of the module) abbreviation `m2`.

```
(module f
  (export f2
          (compute x)
          (get-bar) )
  (load (M1 "m1.scm"))
  (import (f1 M1 "m1.scm"))
  (eval (export f2)
        (import bar) ) )

;;;(f1 _) is OK                                     Scheme+m1

(define (f2 _) _)

(define-macro (m2 _)
  ;;(m1 _) and (f1 _) are OK
  )
;;;                                                Scheme+m1+m2

(define (compute exp)
  (eval exp) )

(define (get-bar)
  bar )
```

The `load` clause of the `module` directive instructs the compiler to load the *m1.scm* file (not the `M1` module) therefore the `f1` function and the `m1` abbreviation are available to the compiler. The body of the `f` module may make use of the `m1` abbreviation and the expansion of an `(m1 _)` abreviation may call `f1`. However if the result of the expansion contains a call to `f1`, the compiler would not find it in the global environment of `f` and would therefore warn the user. To fix this, `f1` is explicitly imported with another module directive. This directive only `import`s `f1`, this is to show that importation may import all or an explicitly named subset of the global variables of a module.

The last clause, the `eval` clause, specifies that `f2` will be made available to the language processed by the call to `eval` within `compute`. Conversely, it also says that the `bar` variable of `eval` may be used as the `bar` global variable within module `f`.

Let us now give a third module, named `M2`.

```
(module M2
  (import (f "f.scm"))
  (main start)
  (eval (export f1)) )

;;;                                                Scheme

(define (start arglist)
  ;;f2, compute, get-bar are OK
  _ )

(define (f1 x)
  (list "f1@m2" x) )
```

This is a main module whose entry point is the `start` function. This function may use the functions imported from module `f`. A third module directive exports for `eval` the current `f1` function defined in the current `M2` module.

When the whole application is started, a call to `compute` will use Scheme as language in a global environment made of `f2` (from `f`), `f1` (from `M2`) and `bar` (seen from `f2`). This language may evolve if enriched with new abbreviations submitted via `compute`.

The language of module directives is rich. It specifies importation, exportation (but no renaming) and re-exportation. Repeatability is ensured since only one module is compiled at a time: abbreviations cannot share state between compilations. The language of abbreviations may be specified (in Scheme but not in terms of compiled modules). The language of (all occurrences of) `eval` may be specified as well.

2.3 Scheme48

Scheme48 compiles modules in memory. An application is built by dumping the current state of the heap (one may also specify the function to invoke first when the image is resumed). The initial image contains the byte-code compiler and offers a REP loop able to interpret Scheme expressions as well as commands to inspect values or specify the module within which interpretation is performed. Commands are recognized by their leading comma.

Abbreviations are defined as specified in R5RS. Syntaxes are available immediately after being defined throughout the rest of the module.

Modules are built with a `define-structure` form (the name comes from SML terminology since it is possible (but undocumented in [6]) to define higher order modules). Modules export names (locations or syntaxes). There are some possibilities to filter the names to export as well as to modify them (both locations and syntaxes).

Our first attempt will define a first module, named `M1`, defining and exporting a function `f1` and an abbreviation `m1`.

```
;;;Within file m1.scm
(define (f1 _) _)
```
Scheme

```
(define-syntax m1
  (syntax-rules ()
    ((m1 x) (list "m1@m1" (f1 x))) ) )
```
Scheme+m1

After going in the `config` module, the `M1` module is compiled, at the level of the REP loop, with:

```
,config
(define-structure M1
  (export f1 (m1 :syntax))
  (open scheme)
  (files "m1.scm") )
```

The `M1` module imports the `scheme` module to gain access to the associated global environment (for example, for `list`) and syntaxes (for instance, for `define-syntax`). This double-sided importation is easily specified with `(open scheme)`. On exportation-side, the `m1` abbreviation is exported with the `:syntax` type. Due to hygien, the `m1` syntax captured the location of the `f1` function.

This first module may be imported by another module, `f`, whose body is contained in the *f.scm* file. This second module is compiled as follows (where the `m1` syntax is renamed `mone`):

```
(define-structure f
  (export f2 compute)
  (open scheme
        f (modify M1 (rename (m1 mone))))
  (files "f.scm") )
```

And its content is:

```
;;;Content of file f.scm
```
Scheme+mone

```
(define (f2 _) _)

(define-syntax m2
  (syntax-rules ()
    ((m2 x)
     (list "m2@f" (f2 x)
           (mone x) (f1 x) ) ) ) )
```
Scheme+mone+m2

```
(define (compute exp)
  (eval exp
        (scheme-report-environment 5) ) )
```

Due to hygien, the macro `m2` captures `f2` and `f1` but it also captures `mone`. The language in which are written expanders is Scheme which happened to define `syntax-rules`. Were we to use another language, we may enrich it with help of the `for-syntax` clause. Here is a variation of module `f` where `m1` is available to define the `m2` macro while the `mone` macro may only appear in the expansion of `m2`. The example is a little contorted since the use of `m1` is very gratuitous but it shows that Scheme48 differentiates the language of the module from the language in which syntaxes are written. This shows the first two level of the macro-expansion tower [9] named "syntax tower" in [6].

```
;;;Content of file ff.scm
```
Scheme+mone

```
(define (f2 _) _)

(define-syntax m2
  (begin
    ;;
```
Scheme+m1

```
    (display (m1 111))(newline)
    (syntax-rules ()
      ((m2 x)
       (list "m2@f" (f2 x)
             (mone x) (f1 x) ) ) ) ) )
```
Scheme+mone+m2

To compile the above module, we just open (that is, import), for the language of syntaxes, the `scheme` (for `display` and `newline`) and `M1` (for `m1`) modules:

```
(define-structure ff
  (export f2)
  (open scheme
        (modify M1 (rename (m1 mone))) )
  (for-syntax (open scheme M1))
  (files "ff.scm") )
```

Scheme48 compiles in memory so it offers various interesting effects: it is possible, at the REP loop, to place oneself

in the context of a module to evaluate some code and even to enrich the current language and global environment:

```
,in F (list (f2 33) (m2 44) (m1 55))
,in F (define (f3) "f3@M1")
,in F (define-syntax m3
        (syntax-rules ()
          ((m3) (list (f2 (f3)))) ) )
,in F (m3)
```

The REP loop offers some features useful for development; for instance, it is possible to reload a module without changing the exportation contract.

Since all modules are known from the REP loop, there is no per se module dependencies. However to determinize the building of an image requires to be able to reset modules to their initial state (in order to reset syntaxes with shared state), something possible with the `reload-package` command.

Whereas the language of modules and syntaxes is well defined, I did not see any possibility to specify the language of `eval` when specifying the module. It is possible though in R5Rs with the usual `scheme-report-environment` function and the like; this is probably also possible making use of the internal `get-package` function.

2.4 Chez Scheme

This Section is only inferred from Waddell's and Dybvig's paper [14]. A module is alike a definition (it may appear wherever a definition may occur (globally or locally)) and looks like (`module` *`module-name`* (*`exported-names`*) *`body`*). A module opens a new namespace that captures all definitions (variables or syntaxes) among which some are exported as mentioned by *exported-names*.

Free variables of the module are also captured by the `module` form but they are not exported. Such a form defines a kind of first-class environment named *module-name* except that syntaxes are also exported.

```
(let ((x 1))
  (module A (f)
    (define (f z) (list x z)) )
  (module B (g)
    (define (g y) (f y)) )
  (import A)
  (let ((x 2))
    (import B)
    (g x) ) )  ;yields (1 2)
```

Modules are imported with the (`import` *`module-name`*) form. This is again a definition form that may appear wherever a definition may occur. When an importation occurs locally the exported names participate to the `letrec` effect as the other internal definitions.

This module system is intimately tied with `syntax-case`: an interesting corollary is that a whole program making use of `module` and `import` forms is transformed, after macro-expansion, into a core Scheme expression (that is, without abbreviations or derived syntaxes). The `syntax-case` facility allows for selective importations, renaming individual variables and gathering exportations with the sole means of hygien (see [14, Section 3.3] for details). It does not seem to allow the renaming or prefixing of all exported variables.

Here are some (untested) examples though they do not make these modules to shine.

```
(module M1 (f1 m1)
  (define f1 _)
  (define-syntax m1 _)
  )

(let ()
  (import M1)
  (module F (f2 compute)
    (define f2 _)
    (define-syntax m2 _)
    (define (compute exp) (eval exp)) ) )
```

Good examples where modules are imported in a local scope are given in the paper [14] however, separate compilation of local modules does not seem practical. These modules do not seem to allow the specification of the language of expanders though the strict and sole use of `syntax-case` alleviates this need. Nor they allow the specification of the language of `eval`.

A very interesting property mentioned in [14] is the structure of the compiled module. Since a module may export locations or syntaxes, the compiled code contains the code to initialize the locations and the code for the exported expanders. When `visit`-ing a module only the expansion resource are set up while `load`-ing a module also initialize the regular locations. This might have been done, in plain old Lisp, with `eval-when`: the compiled code related to syntaxes is therefore conditionalized with a kind of `(eval-when (visit) ...)`.

2.5 MzScheme

MzScheme 200 is the most recently implemented module proposal [4]. It improves on Chez Scheme's module system and solves a number of problems.

A module form specifies its name (bound in a specific namespace), the language it is written in and its body (a sequence of definitions (locations and syntaxes), exportations, importations and expressions). Exportations (of locations or syntaxes) are specified with the `provide` form. This form offers facility to rename, prefix or selectively hide names.

Importing a module is performed with the `require` form; importations may also rename, prefix or selectively hide names. The importation brings in names of locations or syntaxes. Note that importations and exportations are not gathered in a single place.

Let us give an example of a module `M1` exporting a function and a syntax. The module is written in MzScheme; the language of the abbreviation is not specified but as abbreviations adopt the syntax language of R5RS, it should at least contain this latter.

```
(module M1 MzScheme
                                        MzScheme
  (define (f1 _) _)
  (provide f1)
  (define-syntax m1
    (syntax-case _) )
                                        MzScheme+m1
  (provide m1)
)
```

Here is a second module, `F`, that imports `M1` environment

and syntax.

```
(module F MzScheme
  (provide f2 compute)
  (require M1)
                                    MzScheme+m1
  (define (f2 _) _)
  (require-for-syntax (rename M1 m1 mone))
  (begin-for-syntax
      ;;                            R5RS+mone
    (mone _) )
  (define-syntax m2
      ;;                            R5RS+mone
      _ )
                                    MzScheme+m1+m2
  (define (compute exp)
    (eval exp) )
)
```

Modules offer the usual syntax tower. In the `F` module, the language for syntaxes also imports `M1` (its function `f1` and syntax `m1` renamed `mone`) therefore, the language for syntaxes is R5RS enriched with `mone`. A specific syntax, `begin-for-syntax`, evaluates its body in the language of syntaxes (something not so dissimilar to `eval-in-abbreviation-world` [9]). Let us focus a little on `begin-for-syntax`. Compare the old writing with plain old macros with the new syntax[1]:

```
(define-macro (foo _)
  (hack)
  '(bar _) )
```

is now written as

```
(define-syntax foo
   (syntax-case
     (_ (begin (begin-for-syntax (hack))
               (bar _) )) ) )
```

Since dependencies are explicit, `require`-ing a module *M* recursively `require`s the modules *M* requires. Compiling a module *M* requires the modules *M* requires for syntax in order to initialize the syntax tower and its first level: the syntax language. Modules contains sequences of code associated with their phase (run-time, expansion-time, etc.) and only the needed part is run when required by a specific phase. Repeatability is ensured since modules' environments are not shared by differing phase: if a module is required at some phase, it will be reinitialized when required at a different phase.

Concerning explicit evaluation, there also exists in MzScheme namespaces to provide global environments for `eval` (the standard `scheme-report-environment` function creates namespaces). They do not seem to be associated with a syntax tower.

3. TAXONOMY

All these modules systems are very different, they have various goals and few common points. Here is an attempt to classify them.

What is a program? Scheme48 and MzScheme specify what is a program with a grammar defining and instantiating modules. ChezScheme proposes a transformation mapping a program using modules into a single S-expression. Bigloo and Gambit compile towards C (or JVM) and leave `ld` build programs.

Do modules support separate compilation? This sieves ChezScheme embeddable modules from the others.

Do modules support interactive debug? Debugging means, most of the time, violating the language (modifying a constant, conditionally aborting computations, etc.): debugging is not constrained by the language. Offering a toplevel for debug as in Scheme48 complexifies the semantics.

Do modules support classes? Classes are not defined by Scheme but all systems offer a variant of them. Bigloo is the only one that combines class definition and exportation.

4. VAGUE FEELINGS, FUTURE QUESTIONS

This Section is highly hypothetic, it only reflects some instantaneous feelings about modules and macros. It also contains some shallow ideas that need much, much, much work to be published :). Of course, readers are not compelled to share these feelings!

Modules do not need to be embeddable, top-level modules with explicit importations and exportations allow for easier separate compilation. I also tend to think that higher-order modules are not needed in an statically untyped language such as Scheme (generic functions are probably sufficient).

Specifying a language or a global environment are two different things that operate at different times with very different goals. Languages must be totally defined in order to expand modules: they extend the compiler with a pre-pass (an expansion pass). Therefore a language may be represented by a transformer that converts expressions using some abbreviations into expressions that do not use these abbreviations. Therefore an abbreviation may be seen as a language transformer that is, creating a new language enriched with a new abbreviation.

Today, the abbreviation protocol fuse all abbreviations into a single transformer. To stage transformations would be benefitful for instance for macros that want to code-walk expressions after transformation to core Scheme (so they are free of implementation-dependent special forms). How to compose abbreviations into passes and how to rank passes is open.

Constituting global environments is rather independent of the compiler. Even if requiring a unique thing, such as `scheme`, bringing both a global environment and a language is, of course, easier for the user, I tend to separate expansion and linking.

Importation language should allow arbitrary computations on sets of names (for instance, managing the whole set of names associated to a class definition, or names obeying a given naming pattern). The importation language should also be able to accompany sets of locations with extra informations required for better compilation. This extra information should not obfuscate importations.

The only operation that can be performed on a compiled module should be to load it (not to visit, import, use or whatever). I therefore favor a mode where a module is compiled

[1] I tend to think that the first one makes easier to understand the two different languages that are involved.

into a single, monolithic, that is, non conditionalized, code. However, compiling a module requires expanding its body. Expansion requires an evaluation that is done with an appropriate syntax tower. Compiling another module requires another appropriate syntax tower.

On the evaluation side, it should be possible to build specialized `eval`-uator(s) for any given language. Different parts of the whole executable may need more than one language for extension. It should also be possible (maybe with first-class environments [10]) to set up the needed sharing.

The synthetised `eval` takes an expression and a global environment as in R5RS. The returned evaluator comes with its own syntax tower, the language of the macros for this evaluator may be obtained. An example of these functions may be as follows:

(`make-eval` *language-expr*)	→ *evaluator*
(*evaluator expression environment*)	→ *value*
(`syntax-eval` *evaluator*)	→ *evaluator*

A *language-expr* is an expression in a language definition language, a naive example might be:

(*base-language macro* `...`)

Finally, language expressions may also be used to specify local languages to use:

(`with-language` *language-expr s-expr*)

Since a language is seen as an expression transformer it may be obtained by loading a module. Finally, repeatibility must be the paramount property of this system (with first-class languages?) to offer real separate compilation.

5. CONCLUSIONS

This paper discusses various points offered by some module systems for Scheme, some problems they solve or not and some ideas about them. As a conclusion, it seems highly hypothetic to add soon a chapter on modules in R^6RS. However, thinking positively, I propose two measures that should be simpler to introduce:

- Documentations should explain the syntax towers they use (for their toplevel, modules, `eval` or `expand` facilities).
- Introduce an `eval` function with an additional third argument specifying the syntax tower to use.

The source of the various experiments may be found via the net at:

`http://youpou.lip6.fr/queinnec/Programs/sws-2002Aug18.tgz`

6. REFERENCES

[1] P. Curtis and J. Rauen. A module system for Scheme. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[2] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in Scheme. *International journal on Lisp and Symbolic Computation*, 5(4):295–326, 1993.

[3] M. Feeley. *Gambit-C, version 3.0 – A portable implementation of Scheme*, May 1998.

[4] M. Flatt. Composable and compilable macros: You want it *When?* In *ICFP '2002 – International Conference on Functional Programming*, Pittsburgh (Pennsylvania, US), Oct. 2002. ACM.

[5] R. Kelsey, W. Clinger, and J. Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[6] R. Kelsey and J. Rees. *The Incomplete Scheme 48 Reference Manual for release 0.57*, 1999. with a chapter by Mike Sperber.

[7] C. Queinnec. Designing MEROON v3. In C. Rathke, J. Kopp, H. Hohl, and H. Bretthauer, editors, *Object-Oriented Programming in Lisp: Languages and Applications. A Report on the ECOOP'93 Workshop*, number 788, Sankt Augustin (Germany), Sept. 1993.

[8] C. Queinnec. *Lisp in Small Pieces*. Cambridge University Press, 1996.

[9] C. Queinnec. Macroexpansion reflective tower. In G. Kiczales, editor, *Proceedings of the Reflection'96 Conference*, pages 93–104, San Francisco (California, USA), Apr. 1996.

[10] C. Queinnec and D. De Roure. Sharing code through first-class environments. In *Proceedings of ICFP'96 — ACM SIGPLAN International Conference on Functional Programming*, pages 251–261, Philadelphia (Pennsylvania, USA), May 1996.

[11] C. Queinnec and J. Padget. A deterministic model for modules and macros. Bath Computing Group Technical Report 90-36, University of Bath, Bath (UK), 1990.

[12] C. Queinnec and J. Padget. Modules, macros and Lisp. In *Eleventh International Conference of the Chilean Computer Science Society*, pages 111–123, Santiago (Chile), Oct. 1991. Plenum Publishing Corporation, New York NY (USA).

[13] M. Serrano. *Bigloo — A "practical Scheme compiler" for Bigloo version 2.5b*, July 2002.

[14] O. Waddell and R. K. Dybvig. Extending the scope of syntactic abstraction. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 203–213, New York, NY, 1999.