EXTENDING THE SCOPE OF SYNTACTIC ABSTRACTION

Oscar Waddell

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science,
Indiana University

August 1999

Accepted by the faculty of the University Graduate School, Indiana University, in partial fulfillment of the requirements for the degree Doctor of Philosophy

R. Kent Dybvig, Ph.D.
(Principal Adviser)

Daniel P. Friedman, Ph.D.

David B. Leake, Ph.D.

Bloomington, Indiana
July 9, 1999

David C. McCarty, Ph.D.

Acknowledgments

I am profoundly grateful to my advisor, Kent Dybvig, for his keen insight, patient support, and sound advice. His active interest, encouragement, and guidance were vital to the completion of this dissertation. This collaboration has benefitted me immensely both as a researcher and as a person.

I wish to thank the other members of my committee as well: Dan Friedman, David Leake, and David McCarty. Not only did they provide perceptive comments and pose interesting questions, they graciously agreed to a midsummer defense.

Although I was not fortunate enough to have known him, I am indebted to Bob Hieb for his pioneering work on `syntax-case`, which is the basis for the developments in Chapters 2 and 3.

It is with great fondness that I recall animated discussions with the other alumni of the LH415 lab: Mike Ashley, Carl Bruggeman, Bob Burger, and John Zuckerman. Thanks to Bob and Mike for their assistance in developing early versions of the inlining algorithm described in Chapter 4. Carl could contribute to any discussion and was always ready to explore new ideas. John's accounts of real-world software development practice were fascinating and I value his insights into the process. Carl and John both made significant contributions to the SWL system, as well.

Thanks go to Pam Larson for ushering me safely through the labyrinth of paperwork that had to be completed. Without her timely reminders and expert guidance, I would have been lost.

Colleagues at other institutions contributed as well. Olivier Danvy provided helpful comments on early drafts of what eventually became Chapters 2 and 4. Suresh Jaganathan and Andrew Wright provided details on their inlining algorithm and the benchmark code used to test it. Exchanges with Matthias Felleisen helped to clarify the semantics of the $\textsc{Unit}_d$ module facility used as an example in Section 3.7. Lou LaLonde posed a seemingly innocuous question about an early prototype of `define-integrable` that ultimately led us to develop fluid keyword binding. My time in Kansas was enhanced by the opportunity to work with Ed Komp. His rigorous analytical thinking and his perspective as a seasoned professional were much appreciated.

The `vi` text editor and the *Chez Scheme* implementation have been trusted companions on this journey, and I thank the creators of these sublime software systems.

The support, advice, and encouragement of friends was invaluable throughout this process. Electronic mail from Greg and Lisa Bloy brightened my day. Trek fests with Susan Fox and George Calger, sorely missed after they graduated, were always a bright spot in my week. Mike Ashley has been a good friend and colleague from the start. His enthusiasm, wit, good nature, dedication, high standards, and hard work are much appreciated. Glenn and Lorie Bear shared with us the adventure of first-time parenting in the midst of dissertation completion and helped in many ways. It has been a joy to watch their son, Robbie, as he grows. George and Annemarie Springer treated us like family.

For love, support, and encouragement, I am deeply grateful to my parents and their spouses, John Waddell, Karen Dovenmuehle, Karen Bradley, Thomas Norton, and Robin Waddell, and to my brother, Andrew.

My wife, Dawn, is a remarkable and talented woman with a kind heart and the patience of Job. For her steadfast love and support, her ready laughter, her sacrifice, her resilience, and her long hours as mother to our children, I am profoundly grateful. Our children, Marina and Dylan, have been a source of purest joy. When late nights of writing took their toll, family life gave me the strength to continue.

This research was supported in part by the National Science Foundation under grants CDA-9312614, CDA-9401021, CCR-9623753, and CCR-9711269.

Abstract

Syntactic abstraction is a powerful tool enabling users to introduce new notation, to abstract repeated patterns in the text of programs, to express high-level program transformations, and even to embed programming languages within one another. Robust, lexically scoped syntactic abstractions can be implemented via hygienic macros. We extend the scope of syntactic abstraction in three ways. First, we enhance the expressive power of the macro system that supports syntactic abstraction. Second, we introduce a module system that enables the use of syntactic abstraction within large software development projects. Finally, we incorporate optimizations that improve the efficiency of both procedural and syntactic abstractions.

Our primary contributions are as follows. We introduce lexically scoped identifier macros and fluid keyword binding and demonstrate significant applications of both. We extend the macro system to collect and preserve information that can be used by the compiler and run-time system. We present the design and implementation of a realistic module system based entirely on the manipulation of lexical scope and the uniform treatment of `module` and `import` forms as internal definitions. The module system supports nested modules, local modules, and mutual recursion, and permits both import and export of macros, modules, and variables. We describe our solutions to several practical problems in the module design and implementation. In addition, we identify and solve an "implicit export" problem with the potential to cripple static analysis. Finally, we present the design, implementation, and evaluation of a procedure inlining algorithm that is highly effective, yet is significantly faster than other inlining algorithms recently described in the literature.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Syntax matters. To appreciate the importance of syntax, one need only consider the popularity of C++ and Java. Key semantic features of these languages—classes and exceptions, threads and automatic storage management—were available in other programming languages long before they were adopted by C++ and Java, but did not see widespread use until paired with the anointed syntax of the C language. Programmers invest considerable time and energy learning to use the syntax and semantics of a language effectively. The popularity of C++ and Java is due, in part, to their ability to leverage substantial investments in C syntax.

Mastery of syntax is important. Syntax is our interface to the machine: clear syntax improves the reliability of this link; concise syntax improves its efficiency. Clear, concise syntax also benefits the other programmers who must read our code. Expressive syntax, used with skill, enhances productivity. The painter's skill lies not only in transferring paint to canvas, however, but also in mixing pigments to achieve desired effects. Likewise, the programmer who would master syntax is not content to draw from a fixed palette. Thus, the ability to formulate new syntactic constructs is essential to complete mastery of syntax.

This dissertation advances the cause of *syntactic abstraction*, a powerful tool enabling users to introduce new notation, to abstract repeated patterns in the text of programs, to express high-level program transformations, and even to embed languages within one another. We extend the scope of syntactic abstraction in three ways. First, we enhance the expressive power of the mechanisms supporting syntactic abstraction. Second, we support modular use of syntactic abstraction. Finally, we incorporate optimizations that simplify the expression of syntactic abstractions while improving their efficiency.

1.1 Syntactic Abstraction

While the syntax of most programming languages is fixed, some languages provide macro systems that can be used to extend the syntax of the language. The Scheme programming language incorporates a powerful macro system that is the subject of this dissertation.

Macros specify source-to-source rewrite rules that are applied locally within programs prior to evaluation. In effect, a macro system implements a domain-specific language for describing program transformers. Macros let us simplify and standardize repeated patterns in a program, embed languages within one another, introduce syntactic forms with new evaluation rules, and increase run-time efficiency via compile-time program transformations. In the remainder of this section, we briefly consider three applications of macros: abstracting common patterns, capturing best practice, and language embedding. Other applications are demonstrated in subsequent chapters.

1.1.1 Abstracting Common Patterns

Macros are often used to abstract common patterns in the text of programs. For example, `let` is a familiar macro that abstracts a direct `lambda` application.

```
(let ([x e1] [y e2])          ((lambda (x y)
  (let ([z (min x y)])   →       ((lambda (z) e3) (min x y)))
    e3))                        e1 e2)
```

The form on the left has the advantage that the variables `x`, `y`, and `z` are textually adjacent to the expressions `e1`, `e2`, and `(min x y)` that compute their values; this is particularly important when `e3` is large. Not only is the `let` expression easier to read, it is also easier to edit, e.g., to modify the expression that computes the value of a particular binding. The `cond` macro offers similar benefits by abstracting the common control structure of nested conditional expressions.

```
(cond                 (if e1
  [e1 e2]                 e2
  [e3 e4]      →          (if e3
  [e5 e6]                     e4
  [else e7])                  (if e5 e6 e7)))
```

The sequence of tests `e1`, `e3`, `e5` is more readily apparent in the form on the left. The macro also reduces the level of indentation in the program and makes it easier to compare the various tests and their alternatives. Limiting the degree of indentation in a program is useful because our eyes are accustomed to reading text on pages that are taller than they are wide.

Syntactic abstractions can be defined in terms of one another. This simplifies the process of developing new syntactic abstractions. For example, the `let*` macro

```
(let* ([x e1] [y e2] [z e3])
  e4)
```

recursively expands into a series of nested `let` expressions.

```
(let ([x e1])
  (let ([y e2])
    (let ([z e3])
      e4)))
```

Similarly, the `or` macro

```
(or e1 e2 e3)
```

can be defined in terms of `let` or the extended `cond` form.

```
(let ([t1 e1])
  (if t1
      t1
      (let ([t2 e2])
        (if t2 t2 e3))))
```

```
(cond
  [e1 => (lambda (x) x)]
  [e2 => (lambda (x) x)]
  [e3 => (lambda (x) x)]
  [else #f])
```

1.1.2 Capturing Best Practice

Macros can be used to implement syntactic abstractions that codify safe practice. For example, the `parameterize` macro is used below to temporarily set the values of the `print-level` and `print-length` parameters to `3` and `70` during the call to `pretty-print` and to restore their original values upon return.

```
(parameterize ([print-level 3] [print-length 70])
  (pretty-print (read)))
```

The macro generates the following code.

```
(let ([p1 print-level] [p2 print-length])
  (let ([v1 3] [v2 70])
    (let ([swap
           (lambda ()
             (let ([t (p1)]) (p1 v1) (set! v1 t))
             (let ([t (p2)]) (p2 v2) (set! v2 t)))])
      (dynamic-wind
        swap
        (lambda () (pretty-print (read)))
        swap))))
```

The `parameterize` abstraction provides several benefits. First, the abstract code is far more readable than the expanded code. Second, it is easy to propagate any improvements in the implementation of the abstraction to the places where it is used. Finally, using `parameterize` results in more robust code than many programmers would otherwise write. For example, only an experienced programmer would think to insert the bindings for `p1` and `p2` that preserve the original values of the parameters. This is important because the variables `print-level` and `print-length` could be assigned during the evaluation of the `parameterize` body, leaving no way to restore their original values. Similarly, the implementor of the abstraction was careful to introduce local variables `v1` and `v2` for temporary storage of old parameter values. This is important since several threads of control could execute this code concurrently. The old and new parameter values are swapped using `dynamic-wind` so that capturing or reinstating continuations leaves the parameter values in a consistent state. Less-experienced programmers might overlook the possibility of non-local exit and reentry. Using `dynamic-wind` also has the benefit of transparently supporting multiple return values.

The benefits just described are not unique to syntactic abstraction. Procedural abstraction could have been used to implement `parameterize` instead. The procedural implementation would expect a data structure listing parameters and their values, since their number is not fixed. Depending on the ambition of the programmer implementing `parameterize`, the syntax of calls might take one of the forms below.

```
(parameterize (list (cons param₁ val₁) ... (cons paramₙ valₙ)) (lambda () exp))
(parameterize (lambda () exp) param₁ val₁ ... paramₙ valₙ)
(parameterize param₁ val₁ ... paramₙ valₙ (lambda () exp))
```

For $n < 5$, the syntactic abstraction is more concise.

```
(parameterize ([param₁ val₁] ... [paramₙ valₙ]) exp)
```

When n is large or `exp` is large, the syntactic abstraction is arguably more readable. In any case, it is also more efficient and easier to implement. Moreover, it reports syntax errors, such as mismatched numbers of parameters and values, at compile time.

The immediate benefits just described combine to produce an important indirect benefit: they encourage the use of the abstraction. An abstraction pays off only when implemented and used. Simplifying implementation and use promotes abstraction.

1.1.3 Language Embedding

Good language design is hard. Dismay at the proliferation of scripting languages with arcane quotation rules, baroque lexical and grammatical structures, idiosyncratic control flow paradigms, and continued abuse of strings as the only data type prompted Shivers to embed a high-level process-control notation within Scheme [57]. His design delivers on the promise of systems programs that are readable, reliable, even elegant.

Encouraged by his success in elevating shell programming, Shivers proposed a universal scripting framework based on the idea of language embedding [58]. He identifies as targets several "little languages" that trade generality for syntax that is more clear and concise. For example, `sed` describes string transformations based on regular expressions, `awk` supports pattern matching, `nroff`, TeX, and LaTeX support typesetting, `make` specializes in dependency-directed compilation, and `find` is a filesystem query language.

While the goals of these little languages are noble, Shivers objects to the standard practice of implementing each one with a separate interpreter. Using `awk` as an example, he demonstrates several benefits of instead implementing a little language by embedding it within an extensible, general-purpose language. A key benefit of this approach is simplicity. Embedding a sub-language within Scheme is much easier than the usual practice of implementing it from the ground up in C. For example, while the GNU project's C implementation of `awk` is about 14,000 lines of code, embedding `awk` within Scheme requires only 933 lines. Another immediate benefit is quality: although implemented in far fewer lines of code, the embedded `awk` is significantly more powerful than its standalone C counterpart. Finally, embedded languages support more reliable and more efficient composition. While standalone components are composed using pipes to communicate byte streams between separate processes, embedded components inhabit the same address space and communicate via shared access to complex data structures.

Language embedding leverages the considerable design and implementation efforts invested in the host language. Constructs for variable binding and scope control can be reused. Data structures and control operators can be reused. In addition, the embedded language inherits qualities of the host language that few developers of standalone interpreters are willing or able to implement. Examples include optimizing native-code compilers, generational garbage collectors, and powerful syntactic abstraction facilities.

Macros are the enabling technology for language embedding. The core functionality of a system like `awk` could be implemented as a library within any general-purpose programming language. Macros implement the notational conveniences that are, after all, the sole motivation for creating a little language instead.

1.2 Extending the Scope of Syntactic Abstraction

This dissertation extends the scope of syntactic abstraction in three ways. First, we enhance the expressive power of the macro system that enables syntactic abstraction. Second, we support modular use of syntactic abstractions. Finally, we incorporate optimizations that improve the run-time efficiency of programs that use syntactic abstraction.

While the components of this dissertation work well together, each can stand on its own merits. We have already hinted at the power of syntactic abstraction. Module systems are important even in languages without macros, and the optimizations we provide also benefit programs that use procedural abstraction. The remainder of this section briefly outlines the individual contributions of Chapters 2–4 before discussing the relationships between the various components. Chapter 5 presents a more detailed summary of our contributions and proposes future work.

1.2.1 Expressiveness

Chapter 2 describes a syntactic abstraction facility based on the powerful lexically scoped macro system originally developed by Dybvig, Hieb, and Bruggeman [29]. We revise and extend previously published descriptions of the macro language [27] and the expansion algorithm [29] as we present an extended language that incorporates two new features: identifier macros and fluid binding of macro keywords.

Identifier macros extend the concrete syntax of macro calls in a way that is simple, yet powerful. In the original system, macro transformations could be applied to an expression only if it was structured as a list with an identifier at its head. This deliberate overlap with the syntax of applications meant that macros could be used to rewrite portions of programs resembling function calls. In the new system, the syntax of macro calls is extended so that it also overlaps the syntax of variable reference and assignment. We provide several examples of significant transformations that are now possible because macros can be used to rewrite apparent variable references and assignments.

Fluid binding of macro keywords permits a form of intentional identifier capture that complements the `datum->syntax-object` mechanism provided by the original system. While the original mechanism provides control over identifier renaming, the new mechanism provides dynamic control over (expand-time) identifier bindings. Together these mechanisms permit control over both phases of the process that determines the (expand-time) meaning of identifiers. We give examples showing applications of both mechanisms.

1.2.2 Modularity

Chapter 3 presents a module system that seamlessly integrates macros and modules. Numerous examples demonstrate the resulting synergy. The module system features a small core language

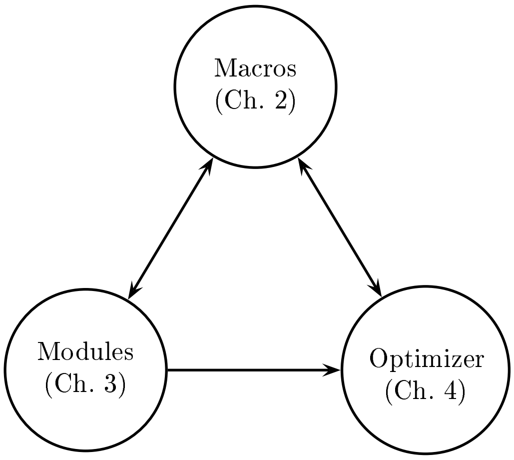

Figure 1: Relationship of components described in the dissertation.

and supports nested modules, local modules, import and export of modules and macros, mutually recursive modules, separate compilation, and general macro transformation, while maintaining lexical scoping for all identifiers. It is implemented by extending the macro system described in Chapter 2. Because it is based entirely on manipulation of static scope, the module system yields efficient compiled code.

1.2.3 Efficiency

Chapter 4 presents the design and implementation of a linear-time optimization pass that is both fast and effective. The optimizer performs procedure inlining, constant propagation, copy propagation, and elimination of useless or unreachable code. The algorithm exploits online transformation, polyvariance, context sensitivity, and demand-driven processing. Performance comparisons show that the algorithm is often more effective, yet substantially faster, than competing algorithms recently proposed in the literature.

1.2.4 Putting the Pieces Together

Figure 1 depicts beneficial relationships between the core components of this dissertation. Bidirectional arrows indicate the synergy between macros and modules and between macros and optimization. The remaining arrow indicates that the module system is also an asset to the optimization pass. An important component not covered in the dissertation, but described elsewhere [63, 64], is a system for visualizing the effects of program transformation and the results of static and dynamic program analyses.

Macros benefit modules. The macro system permits a strategic separation of concerns in the design of our module system. At the language level, we provide only essential functionality: the module system is based entirely on the manipulation of static scope. Using syntactic abstraction we derive, at the source level, a wide range of features found in more elaborate module systems. We reap the benefits—clarity, elegance, and efficiency—of a simple, static core language, while supporting a variety of more elaborate and more dynamic module constructs.

Macros benefit optimization. Optimization inevitably favors certain idioms, and programmers learn to work within those idioms. Macros provide a simple and effective way to harness this domain knowledge. For example, if imperative language features are not supported efficiently, macros can express looping constructs functionally using parameter rebinding to update loop iterands rather than assignment. The macro system also provides enough power to implement domain-specific optimizations that may, for example, apply only in the context of a particular language embedding.

Modules benefit macros. Modules legitimize the use of macros in large-scale software development efforts by providing macros with the same protection and access to auxiliary definitions that variable bindings already enjoy. Modules compound the benefits of macros by permitting selective use of different syntactic abstractions and language embeddings in different parts of a program. The combination of modules and macros delivers on the promise of multi-paradigm programming [12, 32, 47].

Modules benefit optimization. Modules permit precise control over scope and reduce our reliance on global variables as a way to link program components. Global variables are, by nature, an impediment to optimization because it is generally impossible at compile time to determine their run-time bindings. Making more bindings local benefits the optimizer because it limits the scope in which those bindings could be modified. Only when the optimizer can prove that bindings are invariant can it perform constant propagation, copy propagation, or procedure inlining.

Optimization benefits macros. Optimization allows the macro writer to focus on creating a clear and correct expression of the intended semantics for new syntactic abstractions. For example, to bridge the gap between the call-by-name semantics of macros and the call-by-value semantics of the core language, macros often introduce local bindings to prevent duplicate evaluation of their arguments. When the argument happens to be a constant, the local binding is unnecessary. Explicitly testing for this circumstance, however, would complicate the implementation of macros. Instead, the macro writer can rely on the optimizer to perform constant propagation, constant folding, and copy propagation, to eliminate useless or unreachable code, and to inline calls to procedures that are singly referenced. The optimizer can perform more general procedure inlining as well. This eliminates the temptation to overuse macros as integrable procedures.

Visualization benefits optimization. Though not described in this dissertation, visualization is an important component of the system. We have developed a tool that allows the programmer to examine the output of the macro expander and the output of the optimization pass correlated with a display of the original program source. The visualization tool is a tremendous asset to the programmer seeking to understand complex program transformations implemented via macros or performed by the optimizer. We also provide similar tools that display the results of static and dynamic program analyses. These tools depend critically on the ability to correlate source and object code through macro expansion and subsequent optimization.

Syntactic abstraction is a powerful tool for structuring programs. The remainder of this dissertation describes the macro system that supports syntactic abstraction, shows how we can enhance its expressive power, introduces a module system that is seamlessly integrated with macros, and presents an optimization algorithm that, despite low compile-time overhead, can significantly improve the run time of programs using procedural or syntactic abstractions.

Chapter 2

Syntactic Abstraction

Syntactic abstraction can simplify and standardize repeated patterns in the text of programs. Robust, lexically scoped syntactic abstractions can be implemented via hygienic macros. This chapter reviews and extends a powerful hygienic macro system originally developed by Dybvig, Hieb, and Bruggeman [29]. Section 2.1 briefly describes the role of macros in syntactic abstraction. Section 2.2 introduces the language we extend with syntactic abstractions. Section 2.3 illustrates features of the syntactic abstraction facility through a series of examples. Section 2.4 describes the hygienic macro expansion algorithm that supports syntactic abstraction. Section 2.5 shows how the macro expander can collect and preserve information that can be used by the compiler and run-time system. Chapter 3 extends the core of this macro system with support for modular programming.

2.1 Introduction

Macros describe source-to-source transformations that are applied to programs prior to evaluation. Before they are compiled, source programs are translated into a core language by a macro expander that is invoked once for each top-level form in a program. If the expander encounters a macro call, it invokes the associated transformer to expand the macro, then repeats the expansion process for the form returned by the transformer. When the expander encounters a core syntactic form, it constructs the output expression by recursively expanding the subforms.

In traditional Scheme and Lisp macro systems, this translation process can result in unintended variable capture in two ways. First, macros can introduce variable bindings that capture references in subexpressions of the macro call. Second, macros can introduce references that are captured by bindings in the context of the macro call. For example, consider the following rewrite rule for `or`, which introduces a local binding `t` to avoid duplicate evaluation of the expression `e1`.

```
(or e1 e2) → (let ((t e1)) (if t t e2))
```

Given the program

```
(let ((if #f) (t "ok"))
  (or if t))
```

a naive macro expander produces

```
(let ((if #f) (t "ok"))
  (let ((t if))
    (if t t t)))
```

demonstrating both types of inadvertent capture: the binding for `t` introduced by `or` captures the reference to `t` in the subexpression corresponding to `e2`, and the binding for `if` outside the macro call captures the reference to `if` introduced by the macro.

Although unintended capture of the first type can be avoided by special naming conventions or by using `gensym`, these approaches place the burden of preventing capture on the programmer. Furthermore, these techniques cannot prevent unintended capture of the second type. Kohlbecker, Friedman, Felleisen, and Duba first addressed the problem of unintended capture by proposing a *hygiene condition* for macros and an elegant algorithm for enforcing this condition [43]. Their macro system automatically renames bound variables using time stamps to distinguish variables with the same name that were introduced at different times in the transformation process.

Unfortunately, since the hygienic macro-expansion algorithm completely traverses each expression after it is rewritten, the time complexity of the macro expansion process increases from linear to quadratic with respect to the number of expressions present in the source code or introduced during macro expansion. The `syntax-case` system reviewed in this chapter supports hygienic expansion in linear time and provides several other important benefits such as automatic correlation of source and object code [29]. Since the `syntax-case` maintains hygiene not only for variables, but for all identifiers, including keywords, the `or` example above generates the following expansion.

$$
\begin{array}{l}
\texttt{(let ((if}_1\texttt{ \#f) (t}_1\texttt{ "ok"))} \\
\quad\texttt{(let ((t}_2\texttt{ if}_1\texttt{))} \\
\quad\quad\texttt{(if t}_2\texttt{ t}_2\texttt{ t}_1\texttt{)))}
\end{array}
$$

Subscripts indicate the time at which the variables were introduced. For example, $\texttt{if}_1$ and $\texttt{t}_1$ were present in the input to the `or` macro, while $\texttt{t}_2$ was introduced during the expansion of the macro. Because it is not bound locally within the output generated by the macro, the introduced keyword `if` is not renamed.

2.2 A Language for Syntactic Abstraction

This section describes the syntax, scoping rules, and informal semantics of the macro language.

2.2.1 Concrete Syntax

The macro language extends the core language of the Revised[5] Report on Scheme [41]. In the core language, a program consists of a sequence of definitions and expressions.

⟨program⟩ ⟶ ⟨form⟩`*`
⟨form⟩ ⟶ ⟨definition⟩ | ⟨expression⟩

A definition is a variable definition or a sequence of definitions.

⟨definition⟩ ⟶ `(define` ⟨variable⟩ ⟨expression⟩`)`
    | `(begin` ⟨definition⟩`*)`
⟨variable⟩ ⟶ ⟨identifier⟩

An expression is a constant, a quoted datum, a variable reference, an abstraction, an application, a conditional, or an assignment.

⟨expression⟩ ⟶ ⟨constant⟩
    | `(quote` ⟨datum⟩`)`
    | ⟨variable⟩
    | ⟨lambda expression⟩
    | `(`⟨expression⟩ ⟨expression⟩`*)`
    | `(if` ⟨expression⟩ ⟨expression⟩ ⟨expression⟩`)`
    | `(set!` ⟨variable⟩ ⟨expression⟩`)`
⟨lambda expression⟩ ⟶ `(lambda` ⟨formals⟩ ⟨body⟩`)`
⟨body⟩ ⟶ ⟨definition⟩`*` ⟨expression⟩`+`

A `lambda` body consists of a sequence of zero or more definitions followed by one or more expressions. Thus, definitions may appear both at the top level of a program or nested within a `lambda` body.

Support for syntactic abstraction is provided by extending the set of definition forms to include keyword definitions, local keyword bindings, and definitions derived via syntactic abstraction, i.e., macro calls that expand into definitions.

⟨definition⟩ ⟶ `(define` ⟨variable⟩ ⟨expression⟩`)`
    | `(define-syntax` ⟨keyword⟩ ⟨expression⟩`)`
    | `(let-syntax (` ⟨keyword binding⟩`*)` ⟨definition⟩`*)`
    | `(letrec-syntax (` ⟨keyword binding⟩`*)` ⟨definition⟩`*)`
    | `(begin` ⟨definition⟩`*)`
    | ⟨derived definition⟩
⟨keyword⟩ ⟶ ⟨identifier⟩

```
⟨keyword binding⟩ ⟶ (⟨keyword⟩ ⟨expression⟩)
⟨derived definition⟩ ⟶ ⟨macro call⟩
```

In addition, the set of expression forms is extended to include derived expressions.

```
⟨expression⟩ ⟶ ⟨constant⟩
     | (quote ⟨datum⟩)
     | ⟨variable⟩
     | ⟨lambda expression⟩
     | (⟨expression⟩ ⟨expression⟩*)
     | (if ⟨expression⟩ ⟨expression⟩ ⟨expression⟩)
     | (set! ⟨variable⟩ ⟨expression⟩)
     | (let-syntax (⟨keyword binding⟩*) ⟨body⟩)
     | (letrec-syntax (⟨keyword binding⟩*) ⟨body⟩)
     | (fluid-let-syntax (⟨keyword binding⟩*) ⟨body⟩)
     | ⟨derived expression⟩
⟨derived expression⟩ ⟶ ⟨macro call⟩
```

As in the core language, definitions may appear at top level or nested within `lambda` bodies. Definitions may also appear nested within the bodies of derived expressions that expand into `lambda` expressions. For example, the derived expression

```
(let ((x 3)) (define y 1) (+ x y))
```

expands into the following direct `lambda` application.

```
((lambda (x) (define y 1) (+ x y)) 3)
```

Traditionally, the concrete syntax of macro calls overlaps that of procedure application. The resulting ambiguity is resolved by recognizing as a macro call any application having a syntactic keyword (see Section 2.2.2) in the operator position. We extend the concrete syntax of macro calls so that it also overlaps the syntax of variable reference and variable assignment, as indicated by the following grammar.

```
⟨macro call⟩ ⟶ (⟨keyword⟩ ⟨form⟩*)
     | ⟨keyword⟩
     | (set! ⟨keyword⟩ ⟨expression⟩)
```

As before, ambiguity is resolved by checking whether the identifier appearing as a referenced or assigned variable is in fact a syntactic keyword. Generalizing the syntax of macro calls in this way significantly increases the expressive power of the macro language, as shown in Section 2.3.3.

2.2.2 Keyword Bindings

An identifier is a *syntactic keyword* if it has a keyword binding. A keyword binding associates an identifier with a *transformer*, an ordinary procedure that is evaluated and applied during macro expansion. The set of syntactic keywords is not fixed and there are no reserved keywords. New keyword bindings are established using `define-syntax`, `let-syntax`, or `letrec-syntax`. Keyword bindings can be altered temporarily using `fluid-let-syntax`. The syntax of these constructs is summarized below.

```
(define-syntax keyword expression)
(let-syntax ((keyword expression) ...) form0 form1 ...)
(letrec-syntax ((keyword expression) ...) form0 form1 ...)
(fluid-let-syntax ((keyword expression) ...) form0 form1 ...)
```

In each case, *keyword* must be an identifier and *expression* must evaluate to a transformer. Where the ellipsis (...) appears, zero or more occurrences of the immediately preceding form are permitted. The interpretation of $form_0$ $form_1$ ... depends on the binding form and the context in which it appears.

At top level, `define-syntax` establishes globally visible keyword bindings with indefinite extent. As an *internal* definition (within a `lambda` body), `define-syntax` establishes local keyword bindings, visible only within the `lambda` body, whose extent corresponds to the interval of time during which the `lambda` body is expanded. The `let-syntax` and `letrec-syntax` forms establish local keyword bindings whose extent corresponds to the time during which the local binding form is expanded. For `let-syntax` and `letrec-syntax`, the scope of each *keyword* includes each of the forms $form_0$ $form_1$ .... For `letrec-syntax`, the scope of the bindings also includes each *expression*.[1]

Instead of introducing new bindings, the `fluid-let-syntax` form temporarily alters the binding of each *keyword* so that it refers to the associated transformer during the expansion of the body. References to these keywords encountered during the expansion of $form_0$ $form_1$ ..., now refer to these temporary bindings whether or not the references appear within the lexical scope of the `fluid-let-syntax` form. In contrast, `let-syntax` keyword bindings are visible only to references lexically apparent in the body of the `let-syntax` form.

The `fluid-let-syntax` form permits a useful form of intentional capture that resembles dynamic scoping, but is more disciplined. In particular, within the dynamic scope of the construct, the new bindings are visible only to those references that originally referred to the same lexical or top-level keyword rebound by the `fluid-let-syntax` form.

[1] Recursive and mutually recursive keyword bindings can be established via `define-syntax` or `letrec-syntax`. By combining `define-syntax` and `define`, it is also possible to create mutually recursive keyword and variable bindings.

In contexts where an expression is permitted, a `let-syntax` or `letrec-syntax` form may expand into one or more expressions. The resulting expressions are treated as if enclosed in a `begin` form. A `let-syntax` or `letrec-syntax` form may expand into a definition or a sequence of definitions anywhere definitions are permitted. In this case, the resulting definitions are spliced in, replacing the local binding form. The `fluid-let-syntax` form cannot be used as a splicing form containing only definitions.

2.2.3 Macro Transformers

When the expander recognizes a macro call, the corresponding transformer is invoked with a *syntax object* containing information about the context and structure of the call expression. Contextual information is used to preserve lexical scoping and to maintain correlation between source and object code.

Transformers destructure their input using `syntax-case` and construct their output using `syntax`. The syntax of these forms is summarized below.

```
(syntax-case exp (literal ...) clause ...)
(syntax template)
```

The list of literals, `(`*literal* ...`)`, is used to support matching of auxiliary keywords, such as `else` in `cond` and `case` expressions. Each *clause* specifies a pattern to be matched, an optional fender expression, and an output expression. Each *clause* must take one of the following two forms:

```
(pattern output-expression)
(pattern fender output-expression)
```

A *pattern* may contain list structure, vector structure, identifiers, and constants. Each identifier within a pattern is either a *literal*, a pattern variable, or an ellipsis (the identifier "`...`", not to be confused with the meta-language ellipsis, "...". The *fender*, if present, is a Scheme expression that imposes additional constraints on the matching process.

The `syntax-case` form evaluates the expression *exp* to obtain a Scheme object (typically a syntax object) that is matched against the *pattern* in each *clause*. An input form F matches a pattern P if and only if one of the following conditions holds:

- P is a pattern variable,
- P is a literal identifier and F is an identifier with the same binding (see `free-identifier=?` in Section 2.2.4),
- P is of the form `(`P_1 ... P_n`)` and F is a list of n elements such that element i matches P_i, for $1 \leq i \leq n$,

- P is of the form `(`P_1 ... P_{n-1} `.` P_n`)` and F is an improper list where element i matches P_i, for $1 \leq i \leq n$,
- P is of the form `(`P_1 ... P_n P_r `...)` and F is a proper list at least n elements, where element i matches P_i, for $1 \leq i \leq n$, and any remaining elements match P_r,
- P is of the form `#(`P_1 ... P_n`)` and F is a vector of n elements such that element i matches P_i, for $1 \leq i \leq n$,
- P is of the form `#(`P_1 ... P_n P_r `...)` and F is a vector of at least n elements, where element i matches P_i, for $1 \leq i \leq n$, and any remaining elements match P_r, or
- P is any datum that is not a list, vector, or symbol, and F is equal to P in the sense of the `equal?` predicate.

A match is sought by testing the pattern of each clause in order. When a pattern matches the input form, the environment is extended locally with bindings that associate pattern variables with the pieces of the input form they matched. If present, the optional fender is evaluated in the extended environment. If there is no fender or the fender evaluates to a true value, the match succeeds and the corresponding output expression is evaluated in the extended environment. Otherwise, the matching process continues with the next clause. If no clause matches, a syntax error is reported.

The output expression of a `syntax-case` clause typically contains an expression of the form `(syntax` *template*`)`, where *template* can be a pattern variable, an identifier that is not a pattern variable, a pattern datum, a list of subtemplates `(`S_1 ... S_n`)`, an improper list of subtemplates `(`S_1 ... S_{n-1} `.` S_n`)`, a vector of subtemplates `#(`S_1 ... S_n`)`, or the special template `(... ...)`, used to insert a single `...` in the output of a transformer. A subtemplate may be followed by zero or more ellipses (`...`) provided that at least one pattern variable contained within the subtemplate is bound in a pattern that is followed by the same number of ellipses. A `syntax` form returns a Scheme object, much like `quote` or `quasiquote`, but with two important differences: (1) values bound to pattern variables are substituted into the output, and (2) contextual information is retained. Pattern variables can be referenced only within `syntax` forms. Where a subtemplate is followed by an ellipsis, the subtemplate is replicated in the output once for each element of the matching input list with pattern variables bound in turn to corresponding pieces of the matching input sequence. Contextual information is used to support hygienic expansion and to preserve source information.

2.2.4 Predicates and Conversions

In addition to the new core syntax described in Sections 2.2.2 and 2.2.3, the syntax-case system provides several procedures that perform tests or conversions that are useful when writing macros. The procedure `identifier?`, for example, takes a single argument and returns true if its input is a syntax object representing a program identifier. This is often used within fenders to test whether or not a given pattern variable matched an identifier in the input.

Because the macro system transparently renames bound identifiers to prevent inadvertent capture, it is generally not useful to compare identifiers on the basis of their symbolic names except where they are used only as symbolic data. The predicates `free-identifier=?` and `bound-identifier=?` are used to compare two identifiers based on their *intended use* as free references or identifier bindings in a given context. The procedure `free-identifier=?` returns true if and only if the two identifiers refer to the same lexical or top-level binding. A literal identifier within a `syntax-case` pattern matches the input only if the two are equivalent according to `free-identifier=?`. The procedure `bound-identifier=?` returns true if and only if a binding for one identifier would capture references to the other. In general, two identifiers satisfy `bound-identifier=?` only if both are introduced by the same transformer application or, by extension, if both are present in the original input program (but see `datum->syntax-object`).

The procedure `syntax-object->datum` takes a syntax object and returns a Scheme datum with the same structure but with none of the contextual information contained in the original syntax object. The procedure `datum->syntax-object` takes a syntax object representing an identifier (called the *template identifier*) and a Scheme datum; it returns a syntax object with the same structure as the datum and the same contextual information as the template identifier. Identifiers within the resulting syntax object behave as if they were introduced by the same transformer application that introduced the template identifier. In particular, `datum->syntax-object` can be used to construct identifiers that satisfy `bound-identifier=?` even if introduced by different transformer applications. Section 2.3.4 describes macros that use this mechanism to intentionally capture identifier references or bindings.

A transformer can introduce a fixed number of identifiers into its output by naming the identifiers within a `syntax` form. Sometimes it is convenient to introduce a number of identifiers that depends on the particular input to the transformer. In these cases, it is often useful to bind a list of identifiers to a pattern variable that is followed by an ellipsis. If a list of fresh identifiers is required, the procedure `generate-temporaries` can be used to construct a list of as many fresh identifiers as there are elements in the list passed to the procedure.

2.2.5 Relation to R^5RS Macros

The macro language described in this chapter extends the macro language described in the Revised[5] Report on Scheme [41]. While the system described in this chapter permits arbitrary Scheme expressions as macro transformers, for example, the R^5RS system requires transformers to be specified using only `syntax-rules`, a limited variant of `syntax-case` in which each *output-expression* is implicitly wrapped in a `syntax` form. Consequently, none of the procedures described in Section 2.2.4 are provided by the R^5RS macro system. In addition, the pattern language supported by `syntax-case` permits improper lists and vectors in contexts where `syntax-rules` does not. The provision for internal `define-syntax` extends the R^5RS, as does the ability to use `let-syntax` and `letrec-syntax` as splicing constructs containing only definitions. Finally, the concrete syntax of macro calls described in Section 2.2.1 generalizes that of the R^5RS, which allows macro calls only of the form (*keyword subform* ...).

2.3 Examples

This section demonstrates the features of the macro system with a progression of examples that begins with simple textual rewrites, introduces repeated patterns and fenders, then covers recursive macros, identifier macros, and explicit capture. The section concludes with a case study that combines many of these elements with the powerful technique of macros that introduce local macros when expanded.

2.3.1 Simple Macros

The following example combines the elements described in Sections 2.2.2 and 2.2.3 to define a simple macro that provides convenient syntax for `call-with-values`.

```
(define-syntax with-values
  (lambda (x)
    (syntax-case x ()
      ((_ producer consumer)
       (syntax
         (call-with-values (lambda () producer) consumer))))))
```

This definition binds the keyword `with-values` to a transformer that uses `syntax-case` to recognize input of the form (`with-values` *producer consumer*). By convention, the unreferenced pattern variable "_" is used to match the keyword within the pattern, since if the transformer was invoked, the keyword must have matched. The transformer constructs its output using the `syntax` form.

The pattern variables `producer` and `consumer` in the `syntax` form are replaced by the corresponding pieces of the input expression they matched. For example, the expression below

```
(with-values (split < (car ls) (cdr ls))
  (lambda (lo hi)
    (append (sort lo) (cons (car ls) (sort hi)))))
```

expands as follows.

```
(call-with-values
  (lambda () (split < (car ls) (cdr ls)))
  (lambda (lo hi)
    (append (sort lo) (cons (car ls) (sort hi)))))
```

Simple macros like `with-values` do not require the expressive power of `syntax-case` transformers. For these macros, more concise `syntax-rules` transformers can be used instead. The following example shows the definition of `with-values` using `syntax-rules`.

```
(define-syntax with-values
  (syntax-rules ()
    ((_ producer consumer)
     (call-with-values (lambda () producer) consumer))))
```

`syntax-rules` is itself a macro defined in terms of `syntax-case` as follows.

```
(define-syntax syntax-rules
  (lambda (x)
    (syntax-case x ()
      ((_ (lit ...) ((p0 p1 ...) output) ...)
       (syntax
         (lambda (x)
           (syntax-case x (lit ...)
             ((ignored p1 ...) (syntax output)) ...)))))))
```

Just as `(quote e)` may be abbreviated `'e`, the system accepts `#'x` as an abbreviation of `(syntax x)`.[2] The preceding example could therefore also be written as follows.

```
(define-syntax syntax-rules
  (lambda (x)
    (syntax-case x ()
      ((_ (lit ...) ((p0 p1 ...) output) ...)
       #'(lambda (x)
           (syntax-case x (lit ...)
             ((ignored p1 ...) #'output) ...))))))
```

Repetition

The ellipsis (`...`), used earlier in the definition of `syntax-rules`, provides a mechanism for matching repeated elements in the input to a macro and for constructing repeated sequences in the output of a macro. For example, the following definition for `let` uses the ellipsis to match a list of zero or

[2] These transformations are made during lexical analysis and parsing by the `read` procedure and are therefore known as "reader macros".

more bindings of the form `(x v)`, followed by an expression `e0`, followed by zero or more expressions `e1`.

```
(define-syntax let
  (syntax-rules ()
    ((_ ((x v) ...) e0 e1 ...)
     ((lambda (x ...) e0 e1 ...) v ...))))
```

In the output clause, the pattern variables `x` and `v` that were grouped together in the repeated element "`(x v) ...`" in the input pattern, now appear separately. The corresponding pieces of the input pattern are replicated separately where `x` and `v` appear, as shown in the sample expansion below.

```
(let ((x 1) (y 2) (z 3)) (+ x y z))
 → ((lambda (x y z) (+ x y z)) 1 2 3)
```

Multiple Clauses

The definition of `let` given earlier does not support the "named `let`" syntax which permits concise expression of simple loops. To support this alternative syntax, the previous definition can be extended with an additional `syntax-rules` clause, as follows.

```
(define-syntax let
  (syntax-rules ()
    ((_ ((x v) ...) e0 e1 ...)
     ((lambda (x ...) e0 e1 ...) v ...))
    ((_ name ((x v) ...) e0 e1 ...)
     ((rec name (lambda (x ...) e0 e1 ...)) v ...))))
```

With this definition, the named `let` construct expands as shown below.

```
(let fac ((x 10))                ((rec fac
  (if (zero? x)                     (lambda (x)
      1                      →        (if (zero? x)
      (* x (fac (- x 1)))))                1
                                           (* x (fac (- x 1))))))
                                   10)
```

The order of the clauses in the `syntax-rules` (or `syntax-case`) form is important: clauses are matched against the input in the order they appear. The pattern `((x v) ...)` in the first clause is more specific than the pattern variable `name` that appears in the corresponding position in the second clause. If the clauses were reversed, the pattern variable `name` would match any input intended for the pattern `((x v) ...)` in the unnamed `let` case. For example, with the order of the clauses reversed, the expression

```
(let ((id (lambda (x) x)))
  ((id (lambda () 3)))
  1)
```

would expand into the malformed expression

```
((rec ((id (lambda (x) x)))
   (lambda (id) 1))
 (lambda () 3))
```

instead of the intended expression, shown below.

```
((lambda (id)
   ((id (lambda () 3)))
   1)
 (lambda (x) x))
```

Fenders

As the following expansion shows, even the careful ordering of clauses in the previous implementation of `let` is not sufficient to prevent invalid inputs from matching.

```
(let 10 (("oops" 8)) 5) → ((rec 10 (lambda ("oops") 5)) 8)
```

There are two errors in the input: the name of a named-`let` must be an identifier and the bound variable must be an identifier. The following definition of `let` adds fenders that use the predicate `identifier?` to prevent a match in either of these cases.

```
(define-syntax let
  (lambda (x)
    (syntax-case x ()
      ((_ ((x v) ...) e0 e1 ...)
       (andmap identifier? #'(x ...))
       #'((lambda (x ...) e0 e1 ...) v ...))
      ((_ name ((x v) ...) e0 e1 ...)
       (andmap identifier? #'(name x ...))
       #'((rec name (lambda (x ...) e0 e1 ...)) v ...)))))
```

Fenders are supported only by `syntax-case`. This is one way in which `syntax-case` is more expressive than `syntax-rules`. With this definition, the input `(let 10 (("oops" 8)) 5)` is rejected as invalid syntax.

Unfortunately, as the next example demonstrates, the new definition of `let` still admits some malformed inputs.

```
(let ((x 1) (x 2)) x) → ((lambda (x x) x) 1 2)
```

To reject inputs containing duplicate bound identifiers, the following definition of `let` uses the procedure `bound-identifier=?` within the fenders.

```
(define-syntax let
  (lambda (x)
    (define valid-bound-ids?
      (lambda (ls)
        (and (andmap identifier? ls)
             (let f ((ls ls))
               (or (null? ls)
                   (and (andmap
                          (lambda (x) (not (bound-identifier=? x (car ls))))
                          (cdr ls))
                        (f (cdr ls)))))))))
    (syntax-case x ()
      ((_ ((x v) ...) e0 e1 ...)
       (valid-bound-ids? #'(x ...))
       #'((lambda (x ...) e0 e1 ...) v ...))
      ((_ name ((x v) ...) e0 e1 ...)
       (and (identifier? #'name) (valid-bound-ids? #'(x ...)))
       #'((rec name (lambda (x ...) e0 e1 ...)) v ...)))))
```

The locally defined procedure `valid-bound-ids?` must first verify that all elements of the list are, in fact, identifiers, because the subsequent call to `bound-identifier=?` will signal an error if one of its arguments is not an identifier. A user who writes `(let ((x 1) (1 x)) x)` would be rather surprised to receive an error message from `bound-identifier=?` (which is nowhere to be found in the input).

In fact, the core transformers for `lambda` and `letrec` (to which `rec` expands) already check to ensure that the bound identifiers are distinct. The tests added in the fenders we have seen so far are, therefore, not strictly necessary. Section 2.3.3 contains examples where fenders actually are necessary. One motivation for the use of fenders in the preceding examples is better reporting of syntax errors. A novice programmer, for example, may otherwise be surprised by an error message reporting a duplicate identifier in the formals list of a `lambda` expression that does not appear within the input program.[3]

Unfortunately, the desire for meticulous error checking poses a software engineering conundrum: although it is clearly beneficial to the programmer using the macros, it is at odds with the macro writer's desire to reuse the existing error checking code within the core transformers. Fortunately, the macro system described in this chapter is able to correlate the source program with the resulting expanded code. In practice, macro writers can often rely on the error checks performed by the core transformers, knowing that the error messages generated by the core transformers will contain enough source information to point novice users to the offending lines in their program source.

[3] A stack trace might be of some use were it not for the fact that the transformer for `let` has already returned by the time the expander detects the error in the resulting `lambda` expression.

2.3.2 Recursive Macros

While the ellipsis provides a useful way to replicate syntax, it is necessary to express replication of nested structures using macros that expand recursively. For example, the `or` macro defined below expands into nested `if` expressions when there are more than two alternatives.

```
(define-syntax or
  (syntax-rules ()
    ((_) #f)
    ((_ e) e)
    ((_ e others ...) (let ((t e)) (if t t (or others ...))))))
```

The first clause matches the case in which no subexpressions are supplied: `(or)` → `#f`. The second clause matches the case where a single subexpression is supplied: `(or e)` → `e`. The final clause matches an `or` expression with two or more subexpressions and expands recursively into an `or` expression with one less term. To avoid duplicate evaluation of the expression `e` in the guard and consequent branch of the `if` expression, the macro introduces a `let` binding `t`. This is necessary because macros resemble call-by-name procedures. If the `let` binding were omitted, for example, computational effects in the evaluation of `(f)` could be duplicated due to the following expansion.

```
(or (f) (g)) → (if (f) (f) (g))
```

While mutually recursive macros are normally defined using `define-syntax` or `letrec-syntax`, it is also possible to define such macros using `let-syntax` or `fluid-let-syntax`, as shown below.

```
(let-syntax
  ((spin (syntax-rules ()
           ((_ f g e) e)
           ((_ f g e more ...) (list 'spin e (f g f more ...)))))
   (loop (syntax-rules ()
           ((_ g f e) e)
           ((_ g f e more ...) (list 'loop e (g f g more ...))))))
  (spin loop spin 1 2 3 4 5))
(fluid-let-syntax
  ((spin (syntax-rules ()
           ((_ e) e)
           ((_ e more ...) (list 'spin e (loop more ...)))))
   (loop (syntax-rules ()
           ((_ e) e)
           ((_ e more ...) (list 'loop e (spin more ...))))))
  (spin 1 2 3 4 5))
```

Both of these expressions evaluate to `(spin 1 (loop 2 (spin 3 (loop 4 5))))`.

2.3.3 Identifier Macros

Since the concrete syntax for macro calls has been extended to include apparent variable references and variable assignments in addition to the traditional list structure, macro transformers may be invoked with any of these forms. Section 2.3.5 describes macros, generated by `define-integrable`, that exploit this fact to expand differently depending on the form of the macro call.

Identifier macros commonly expand as if they were variable references independent of their position in a structured form. The `identifier-syntax` macro defined below makes it easy to specify transformers for simple identifier macros.

```
(define-syntax identifier-syntax
  (lambda (form)
    (syntax-case form ()
      ((_ exp)
       #'(lambda (form)
           (syntax-case form ()
             (id (identifier? #'id) #'exp)
             ((id . rest) #'(exp . rest))))))))
```

The macro call `(identifier-syntax exp)` expands into a `lambda` expression that destructures its input using a `syntax-case` form with two clauses. The first clause produces the output `exp` and uses a fender to match only when the input is an identifier. The second clause constructs an application of `exp`, and matches when the input is a structured form.

The ability to replace identifier references with arbitrary code extends the expressive power of the macro system significantly. Identifier macros allow the programmer to express many transformations on identifiers that would otherwise require an *ad hoc* code walk. Moreover, unlike symbol macros in Common Lisp and other systems [28, 60], identifier macros are immune to inadvertent capture because the underlying system preserves lexical scoping.

The lazy variant of `let` below is a simple application of identifier macros and `identifier-syntax`. The `lazy-let` construct binds each variable to a promise [41] that computes its value.

```
(define-syntax lazy-let
  (syntax-rules ()
    ((_ ((v e) ...) e1 e2 ...)
     (let ((v (delay e)) ...)
       (let-syntax ((v (identifier-syntax (force v))) ...)
         e1 e2 ...)))))
```

Identifier macros are used to replace each variable reference `v` in the body (`e1 e2 ...`) with code that forces the corresponding promise. Because the underlying system enforces lexical scoping, identifier macros can be used with predictable results. For example, the following evaluates to `3` despite the local bindings for `x` and `force`.

```
(lazy-let ((x 1)
           (f (lambda (x) (+ x 2)))
           (y (let diverge () (diverge))))
  (let-syntax ((y (identifier-syntax x)))
    (let ((x 7) (force #f))
      (f y))))
```

Because the concrete syntax of macro calls also includes apparent variable assignments, it is useful to extend the `identifier-syntax` macro to accept either syntax outlined below.

(identifier-syntax *e*)
(identifier-syntax (*id* e_1) ((set! *id* *val*) e_2))

The first form is identical to that described earlier and inserts e in place of the corresponding macro keyword wherever the keyword occurs as an apparent variable reference. A syntax error is reported if the keyword occurs as an apparent variable assignment. The second form contains two subforms: the first indicates that the keyword should be replaced with e_1 wherever it occurs as an apparent variable reference, and the second indicates that the keyword should be replaced with e_2 wherever it occurs as an apparent variable assignment. When the transformer is invoked on an apparent assignment, the pattern variable *val* is bound to the expression on the right-hand side of the assignment and is visible within the output template e_2.

To support macro calls with the concrete syntax of variable assignments, the macro expander permits extended keyword bindings in which transformers are tagged to indicate whether or not they accept the extended concrete syntax. Tagging the transformers is useful for backward compatibility, because established practice when specifying `syntax-case` patterns is to disregard the keyword position by using the pattern variable "_".

The core-language `set!` transformer now consults the expand-time store (see Section 2.4) to determine whether the identifier to be assigned represents a variable or a keyword. If the identifier represents a variable, the usual assignment is constructed. If the identifier represents a keyword, the expander checks whether the transformer accepts the extended concrete syntax. If so, the transformer is applied to the input initially passed to the core `set!` transformer. Otherwise a syntax error is reported. An implementation of the more general `identifier-syntax` macro is given below.

```
(define-syntax identifier-syntax
  (lambda (x)
    (syntax-case x (set!)
      ((_ e)
       #'(lambda (x)
           (syntax-case x ()
             (id (identifier? #'id) #'e)
             ((_ x (... ...)) #'(e x (... ...))))))
      ((_ (id exp1) ((set! var val) exp2))
```

```
          (and (identifier? #'id) (identifier? #'var))
          #'(cons 'macro!
                  (lambda (x)
                    (syntax-case x (set!)
                      ((set! var val) #'exp2)
                      ((id x (... ...)) #'(exp1 x (... ...)))
                      (id (identifier? #'id) #'exp1))))))))
```

The more general version of identifier-syntax can be used to provide convenient variable-like syntax for structure accessors and mutators. For example, an identifier macro is used below to rewrite all references and assignments to ivar as vector-ref and vector-set! operations on the formal parameter self.

```
(lambda (self x y z)
  (let-syntax ((ivar (identifier-syntax
                       (var (vector-ref self 1))
                       ((set! var val) (vector-set! self 1 val)))))
    (let ((old ivar))
      (set! ivar 4)
      (+ old ivar x y z))))
```

The macro-expanded form of this expression is shown below.

```
(lambda (self x y z)
  (let ((old (vector-ref self 1)))
    (vector-set! self 1 4)
    (+ old (vector-ref self 1) x y z)))
```

More elaborate examples are presented in Sections 3.4.8 and 3.7.

It is important to note that a keyword is also a valid output for identifier-syntax. This provides the ability to alias keywords. For example, in the following expression, bind and seq are introduced as local names for let and begin.

```
(let-syntax ((bind (identifier-syntax let)) (seq (identifier-syntax begin)))
  (bind ((x 'hello) (y 'world))
    (seq (display x)
         (write-char #\space)
         (display y)
         (newline))))
```

In fact, the general implementation of identifier-syntax makes it possible to define syntactic aliases that transparently replace references, assignments, and calls to one identifier with references, assignments, and calls to another. These aliases work for both variables and keywords.

```
(define-syntax define-alias
  (syntax-rules ()
    ((_ x y)
     (define-syntax x
       (identifier-syntax
         (var y)
         ((set! var val) (set! y val)))))))
```

Given the implementation of `define-alias` above, the expression `(define-alias a b)` defines `a` as an identifier macro that expands into references, calls, or assignments to `b` depending on its use. The following macro can be used to introduce one or more aliases within a limited scope.

```
(define-syntax alias
  (syntax-rules ()
    ((_ ((x t) ...) e0 e1 ...)
     (let ()
       (define-alias x t) ...
       e0 e1 ...))))
```

Syntactic aliases can be used to manipulate scope in interesting ways. For example, using the `alias` form just introduced, it is possible to implement the `irreflexive-letrec` form (proposed by Friedman [private communication]) which is similar to `letrec` except that each bound identifier is visible in all other right-hand expressions but its own.

```
(define-syntax irreflexive-letrec
  (lambda (x)
    (syntax-case x ()
      ((_ ((x v) ...) e0 e1 ...)
       (with-syntax (((t ...) (generate-temporaries #'(x ...))))
         #'(alias ((t x) ...)
             (letrec ((x (alias ((x t)) v)) ...)
               e0 e1 ...)))))))
```

When this macro expands, it introduces temporary names for the bound identifiers by wrapping an `alias` form around the `letrec`. On the right-hand side of each `letrec` binding another `alias` is used to locally rename the bound identifier to the corresponding temporary name, which in turn is a name for the binding, if any, of the original identifier outside the `letrec` form. For example, the following expression evaluates to the list `(0 1)`.

```
(let ((x 0))
  (irreflexive-letrec ((x (lambda () x)) (y (lambda () (+ (x) 1))))
    (list (x) (y))))
```

This expression expands (roughly) as follows.

```
(let ((x0 0))
  (let ()
    (define-alias t1 x0)
    (define-alias t2 y)
    (letrec ((x1 (let ()
                   (define-alias x1 t1)
                   (lambda () x1)))
             (y2 (let ()
                   (define-alias y2 t2)
                   (lambda () (+ (x1) 1)))))
      (list (x1) (y2)))))
 →
```

```
(let ((x0 0))
  (let ()
    (letrec ((x1 (lambda () x0)) (y1 (lambda () (+ (x1) 1))))
      (list (x1) (y1)))))
```

2.3.4 Intentional Capture

While automatic protection against unintended capture is clearly beneficial, certain program transformations cannot be expressed in hygienic macro systems unless a controlled form of *intentional* capture is permitted. The macro system described in this chapter supports intentional capture in two ways. First, the procedure `datum->syntax-object` can be used to introduce *implicit* identifiers that behave as if they were present in the input to a macro. Second, the `fluid-let-syntax` form can be used to temporarily rebind keywords during the expansion of its body.

Implicit Identifiers

Some macros require the ability to introduce identifier bindings or references that capture references or bindings in the macro input. For example, suppose we want to define a `loop` macro that binds the variable `break` to an exit procedure within the loop body. We might try the following definition.

```
(define-syntax loop
  (syntax-rules ()
    ((loop body)
     (call/cc (lambda (break) (let f () body (f)))))))
```

This does not work. The introduced binding for `break` fails to capture references to `break` within `body` due to the hygienic expansion mechanism. One way to see this is by analogy with β-reduction in the λ-calculus.[4]

$$(\lambda x.e_1)e_2 \xrightarrow{\beta} e_1'[e_2/x] \qquad \text{where } e_1 \xrightarrow{\alpha} e_1'$$

In β-reduction, the actual parameter e_2 is substituted for formal parameter x in the expression e_1' that results when α-conversion is used to consistently rename bound variables in the body of the λ-expression with fresh identifiers. Treating the expansion of the `loop` macro as a λ-calculus β-reduction we have the following.

```
(loop (break 'out))
 ⟶ ((lambda (body) (call/cc (lambda (break) (let f () body (f))))) (break 'out))
 β⟶ (call/cc (lambda (break1) (let f1 () body (f1))))[(break 'out)/body]
 ⟶ (call/cc (lambda (break1) (let f1 () (break 'out) (f1))))
```

[4]The analogy with λ-calculus is quite natural given the call-by-name semantics of macro expansion.

Because the expression `(break 'out)` is part of the input to the macro, modeled here as an argument to the call-by-name procedure, it is substituted into the body after α-conversion, and therefore does not capture the binding for $\texttt{break}_1$.

In fact, the expander uses a slightly different mechanism (described in Section 2.4) for renaming. When expanding a macro, the expander first generates a fresh mark and applies this mark to identifiers in the input and output of the macro transformer. Because identical marks cancel when they meet, identifiers in the output are marked only if they were introduced by the macro transformer. When the core binding constructs rename their bound identifiers, the marks are used to recognize and rename references to bound identifiers. An identifier is renamed only if it has the same name and marks as an identifier bound by a core binding construct. In the expansion of `loop` macro given earlier, the identifier `break` introduced by the macro retained the mark that was applied to the macro output, while the `break` in the input did not. When the `lambda` expression is expanded and the marked `break` is renamed, the unmarked `break` will not be renamed.

To permit a controlled form of capture, the `syntax-case` system allows macros to create *implicit identifiers* that behave as if they were present in the input to the macro. Implicit identifiers are constructed using the procedure `datum->syntax-object`. Given a template identifier and a Scheme datum, `datum->syntax-object` returns a syntax object with the same structure as the datum but with each symbol replaced by the identifier that would have resulted if that symbol had appeared with the template in the input form.

To permit the intended capture of `break`, the `loop` macro can be rewritten as follows.

```
(define-syntax loop
  (lambda (x)
    (syntax-case x ()
      ((kwd body)
       (with-syntax ((brk (datum->syntax-object #'kwd 'break)))
         #'(call/cc (lambda (brk) (let f () body (f)))))))))
```

The `with-syntax` form is used to introduce a new pattern variable `brk` bound to an implicit identifier constructed by `datum->syntax-object`. The `with-syntax` macro is defined in terms of `syntax-case`. When the transformer constructs the macro output, the implicit identifier is substituted for the pattern variable `brk`, and the input subform that matched the `body` template is substituted for `body`.

The implicit identifier behaves just as the identifier `break`, if any, introduced in the expansion that introduced the identifier bound to the pattern variable `kwd` used as the template identifier in the call to `datum->syntax-object`. In particular, the implicit identifier acquires the same marks (and substitutions) as the template identifier. Because the template identifier was marked by the

```
(define-syntax include
  (lambda (form)
    (define read-file
      (lambda (fn k)
        (let ((p (open-input-file fn)))
          (datum->syntax-object k
            (let loop ((x (read p)))
              (if (eof-object? x)
                  (begin (close-input-port p) '())
                  (cons x (loop (read p)))))))))
    (syntax-case form ()
      ((k filename)
       (let ((fn (syntax-object->datum #'filename)))
         (with-syntax (((exp ...) (read-file fn #'k)))
           #'(begin exp ...)))))))
```

Figure 2: An expand-time form of `load`.

expander, the implicit identifier is also marked and therefore appears to have been present in the input to the macro. When the expander reapplies the mark to the macro output, the mark on the implicit identifier is canceled. The same marking and unmarking will apply to occurrences of `break` in the input subform bound to the pattern variable `body`. Because the identifiers have the same name and marks, the `lambda` binding for `break` will capture references to `break` within `body`. Given the input `(loop (break 'out))`, the macro now expands as follows.

$$\texttt{(call/cc (lambda (break}_1\texttt{) (let f}_2\texttt{ () (break}_1\texttt{ 'out) (f}_2\texttt{))))}$$

The `loop` example shows that macros can introduce bindings that capture identifier references in their input. In fact, a macro can also introduce identifier references that capture bindings in the surrounding context of the macro call. The `include` macro defined in Figure 2 does both. The `include` macro uses `datum->syntax-object` to place the expressions read from the file named by `filename` into the proper lexical context so that identifier references and definitions within those expressions are scoped where the `include` form appears. For example, if the file `f-def.ss` contains the expression `(define f (lambda () x))`, then the following expression evaluates to `"okay"`.

```
(let ((x "okay"))
  (include "f-def.ss")
  (f))
```

The existing binding for `x` captures the reference to `x` introduced in the definition of `f`, and this introduced definition captures the existing reference to `f`.

Often `datum->syntax-object` is used to introduce compound identifiers whose names are constructed from one or more constituent identifiers in the input to the macro. This common usage is

```
(define-syntax define-simple-record
  (lambda (x)
    (syntax-case x ()
      ((_ name (field ...))
       (with-syntax ((maker (compound-id #'_ "make-" #'name))
                     ((acc-id ...)
                      (map (lambda (fld) (compound-id #'_ #'name "-" fld))
                           #'(field ...)))
                     ((accessor ...)
                      (map (lambda (fld)
                             (with-syntax ((fld fld)) #'(lambda (field ...) fld)))
                           #'(field ...))))
         #'(begin
             (define maker
               (lambda (field ...)
                 (lambda (f)
                   (f field ...))))
             (define acc-id
               (lambda (obj)
                 (obj accessor))) ...))))))
```

Figure 3: A simple record facility.

abstracted by the user-defined procedure `compound-id` shown below.

```
(define compound-id
  (lambda (template . ids)
    (define id->string
      (lambda (x)
        (cond
          [(string? x) x]
          [(identifier? x) (id->string (syntax-object->datum x))]
          [(symbol? x) (symbol->string x)])))
    (datum->syntax-object
      template
      (string->symbol
        (apply string-append (map id->string ids))))))
```

Given a template identifier and zero or more strings, symbols, or identifiers, `compound-id` returns an identifier scoped where template appears but with a name formed by concatenating the names of the constituents. The simple record facility defined in Figure 2.3.4 uses `compound-id` to construct the names of the record constructor and accessors. For example, the derived definition `(define-simple-record point (x y))` expands into the following sequence of definitions.

```
(begin
  (define make-point
    (lambda (x y)
      (lambda (f)
        (f x y))))
  (define point-x
```

```
    (lambda (obj)
      (obj (lambda (x y) x))))
  (define point-y
    (lambda (obj)
      (obj (lambda (x y) y)))))
```

Because `compound-id` uses `datum->syntax-object`, definitions for the constructed identifiers capture references that appear within the scope of the record definition, as intended.

Keyword Rebinding

Suppose that we would like to define a `while` construct that repeatedly tests a condition and evaluates a sequence of expressions until the condition no longer holds. Using the simple `loop` macro developed earlier, the `while` construct can be defined as follows.

```
(define-syntax while
  (syntax-rules ()
    ((_ test e0 e1 ...)
     (loop (if (not test) (break) (begin e0 e1 ...))))))
```

This works as expected, except that, unfortunately, `break` is not visible within the sequence of expressions `e0 e1 ...`. To make `break` visible within subforms of the `while` construct, we could arrange for explicit capture using `datum->syntax-object`, as shown below.

```
(define-syntax while
  (lambda (x)
    (syntax-case x ()
      [(k test e0 e1 ...)
       (with-syntax ((loop (datum->syntax-object #'k 'loop)))
         #'(loop (if (not test) (break) (begin e0 e1 ...))))])))
```

Unfortunately, we would have to arrange for the explicit capture of `break` yet again in order to define a `break`-aware `for`-loop construct in terms of `while`.

The following implementation of the `loop` macro presents another alternative. This definition uses `fluid-let-syntax` to temporarily rebind the keyword `break` to an identifier macro that expands as a reference to the reified continuation `k`.

```
(define-syntax loop
  (syntax-rules ()
    ((_ body)
     (call/cc
       (lambda (k)
         (fluid-let-syntax ((break (identifier-syntax k)))
           (let f () body (f))))))))
```

The identifier `break` temporarily rebound here refers to the binding of `break`, if any, visible at top level (globally). A free reference to `break` in the input to the macro also resolves to the top-level `break`.

Because the top-level `break` is temporarily rebound during the expansion of `(let f () body (f))`, references to `break` in the macro input are rewritten as references to the continuation `k`, as shown by the following expansion.

```
(loop (break 'out))
  → (call/cc (lambda (k) (let f () (k 'out) (f))))
```

It is important to realize that `fluid-let-syntax` respects lexical scoping. In fact, its name is derived from the notion of "fluid binding" in which the binding of a lexical variable is modified on entry to a dynamic context and restored on exit. Fluid binding provides much of the convenience of dynamic scoping while retaining many benefits of static scoping. In particular, fluid binding limits the visibility of dynamic effects to code within a prescribed static scope.

Just as `fluid-let` [26] temporarily rebinds variables at run time, `fluid-let-syntax` temporarily rebinds keywords at expand time. During the expansion of the `fluid-let-syntax` body, the visible lexical (or top-level) binding for each keyword is temporarily replaced by a new association between the keyword and the corresponding transformer. This effectively captures references to the keyword that resolve to the same lexical (or top-level) binding whether the references occur in the text of the body or are introduced during its expansion. In contrast, `let-syntax` captures only those references that occur within the text of its body.

Drawing on the analogy between the evaluation of `let` and `fluid-let` and the expansion of `let-syntax` and `fluid-let-syntax`, the following examples show how lexically scoped fluid binding differs from dynamic scoping.

```
(let ([x 1])
  (let ([f (lambda () x)])
    (let ([x 2])
      (let ([g (lambda () x)])
        (fluid-let ([x 100]) (+ (f) (g)))))))              ⇒  101

(let-syntax ((x (identifier-syntax 1)))
  (let-syntax ((f (syntax-rules () ((_) x))))
    (let-syntax ((x (identifier-syntax 2)))
      (let-syntax ((g (syntax-rules () ((_) x))))
        (fluid-let-syntax ((x (identifier-syntax 100)))
          (+ (f) (g)))))))                                 →  (+ 1 100)
```

Just as the first expression evaluates to `101`, the second expands into `(+ 1 100)`. In both cases, the fluid binding construct alters the binding of `x` in scope, leaving the shadowed outer binding for `x` unchanged. If `fluid-let` instead implemented dynamic scoping, the call `(f)` would return 100 and the first expression would evaluate to `200`. Similarly, if `fluid-let-syntax` were a dynamic scoping construct, the second expression would expand to `(+ 100 100)`.

Because `fluid-let-syntax` provides fluid binding that respects lexical scoping, the `loop` macro implementation on page 33 has certain advantages over the implementation on page 30 that uses `datum->syntax-object`. First, the identifier `break` is captured within the body of the `loop` only if it refers to the same lexical binding apparent where `loop` was defined (in this case, the top-level binding). For example, because it refers to the `let` binding rather than the binding in scope where `loop` was defined, `break` is not captured when the following expression is expanded.

```
(let ((break (lambda (x) x)))        (let ((break (lambda (x) x)))
  (loop (break 'out)))          →      (call/cc
                                         (lambda (k)
                                           (let f () (break 'out) (f)))))
```

In this case, `break` resolves not to the implicit continuation binding, but to the lexically apparent binding.

The second advantage of the `loop` definition on page 33 is that the temporary rebinding of `break` persists throughout the process of expanding the loop body. In particular, the capture of `break` is not limited to the single transcription step described by the `loop` macro transformer, but instead extends to all transcriptions applied while translating the `loop` body into the core language. As a result, there is no need to arrange for the explicit capture of `break` each time we derive new syntax in terms of `loop` or another construct already derived from `loop`. For example, we can define a `while` construct directly in terms of `loop` and a `for` construct directly in terms of `while` as follows.

```
(define-syntax while
  (syntax-rules ()
    ((_ test e0 e1 ...)
     (loop (if (not test) (break) (begin e0 e1 ...))))))

(define-syntax for
  (syntax-rules ()
    ((_ (initial test update) e0 e1 ...)
     (begin
       initial
       (while test e0 e1 ... update)))))
```

Both constructs naturally support the use of `break` without referring to details of the `loop` macro implementation.

2.3.5 Case Study: User-defined Integrable Procedures

Macros are often used as substitutes for integrable procedures. While this practice is quite common even among users of primitive macro systems, it can cause serious problems. First, if the macro system does not preserve lexical scoping, unintended variable capture can cause hard-to-find errors. Second, because macros are call-by-name, the macro writer must be careful to avoid duplicate evaluation of actual parameters. Finally, users must remember which procedures are actually defined as macros to avoid attempting to use them as first-class values. The first problem is solved by using a lexically scoped macro system such as the one described in this chapter. The second problem is solved by the technique of `let`-insertion [11]. The final problem is solved by using macros that expand differently depending on whether or not they are referenced in the operator position of an application.

In this section, we develop a user-level mechanism for procedure integration that satisfies a key requirement: integrable procedures are virtually indistinguishable from ordinary procedures. Two differences are observable. First, at top level, an integrable procedure must be defined before it is referenced within compiled code. This is a familiar consequence of staged translation. Because programs are fully macro-expanded prior to compilation, defining a macro `f` has no effect on references to `f` within previously compiled code. Second, unlike ordinary variable bindings, integrable procedure bindings are not mutable by `set!`.

Although using macros to define integrable procedures is often frowned upon, this case study is included here for two reasons. First, most arguments against this practice are based on the historical inability to bridge the semantic gap between macros and procedures. Our ability to close this gap makes this practice more acceptable. Second, this example exercises several features of the macro system. It shows the use of macros that define other macros and macros that expand into local macros, and it shows an important application of `fluid-let-syntax`. We begin by reviewing briefly the solutions to the problems described earlier. Next, we address the issue of recursively defined integrable procedures.

Preserving Lexical Scoping

Lexical scoping must be preserved if integrable procedures are to behave just as ordinary procedures. For example, it should be possible to refer to free variables within an integrable procedure even where those free variables are shadowed at the call site, as shown by the subscripts in the following expansion.

```
(let ((x 3))
  (define-integrable scale             (let ((x₁ 3))
    (lambda (y) (* x y)))      →         (let ((x₂ 5))
  (let ((x 5))                             (* x₁ x₂)))
    (scale x)))
```

This would be virtually impossible were it not for the fact that the underlying macro system preserves lexical scoping.

The define-integrable form introduced above abstracts the process of defining integrable procedures as macros. A natural implementation is given below.

```
(define-syntax define-integrable
  (syntax-rules (lambda)
    [(_ name (lambda (x ...) e0 e1 ...))
     (define-syntax name
       (syntax-rules ()
         [(_ x ...) (begin e0 e1 ...)]))]))
```

When invoked, this implementation generates a macro definition, as illustrated by the following expansion for the integrable procedure scale.

```
(define-integrable scale  →  (define-syntax scale
  (lambda (y) (* x y)))         (syntax-rules () [(_ y) (* x y)]))
```

Avoiding Duplicate Evaluation

Unfortunately, the simple implementation of define-integrable just given defines macros that may duplicate evaluation of their arguments, as shown below.

```
(define-integrable square (lambda (x) (* x x)))
  →  (define-syntax square
        (syntax-rules () [(_ x) (* x x)]))

(square (begin (display "hello") 3))
  →  (* (begin (display "hello") 3) (begin (display "hello") 3))
```

To obtain the desired applicative-order call-by-value semantics, integrable procedures can instead be defined as identifier macros that insert copies of the corresponding lambda expressions wherever the macros are referenced.

```
(define-syntax define-integrable
  (syntax-rules (lambda)
    [(_ name (lambda formals e0 e1 ...))
     (define-syntax name (identifier-syntax (lambda formals e0 e1 ...)))]))
```

Using this implementation of define-integrable, we obtain a direct lambda application (equivalent to let) wherever an integrable procedure is directly called.

```
(lambda (y)
  (define-integrable square (lambda (x) (* x x)))
  (square (begin (display "hello") y)))
  →  (lambda (y)
        (define-syntax square (identifier-syntax (lambda (x) (* x x))))
        (square (begin (display "hello") y)))
  →  (lambda (y)
        ((lambda (x) (* x x)) (begin (display "hello") y)))
  →  (lambda (y)
        (let ((x (begin (display "hello") y)))
```

```
             (* x x)))
```

This `let`-insertion technique is part of the folklore of macro use and has also been used by the partial evaluation community to avoid duplicating computations during specialization [11].

Defining integrable procedures as identifier macros also means that an integrable procedure can be used as a first-class value.

```
(lambda (ls)
  (define-integrable square (lambda (x) (* x x)))
  (map square ls))
  →  (lambda (ls)
       (define-syntax square (identifier-syntax (lambda (x) (* x x))))
       (map square ls))
  →  (lambda (ls)
       (map (lambda (x) (* x x)) ls))
```

Maintaining Pointer Equality

While the second implementation of `define-integrable` is closer to the semantics of `define` than the first, it does not preserve pointer equality. For example, the expression on the left evaluates to `#t`, but the expression on the right might evaluate to `#f`.

```
(let ()                               (let ()
  (define f (lambda (x) x))             (define-integrable g (lambda (x) x))
  (eq? f f))                            (eq? g g))
```

When the call `(eq? g g)` is expanded, each reference to `g` inserts a copy of the `lambda` expression. In the resulting call, `(eq? (lambda (x) x) (lambda (x) x))`, each `lambda` expression may evaluate to a different closure, so `eq?` may return `#f`. If the `lambda` expression has free variables, this approach may also be less efficient since a closure may be allocated at each site where the integrable procedure is referenced rather than being allocated once at the definition site.

The following implementation of `define-integrable` solves these problems by defining integrable procedures as macros that expand selectively based on the context in which they are used.

```
(define-syntax define-integrable
  (syntax-rules (lambda)
    [(_ name (lambda formals e0 e1 ...))
     (begin
       (define residual (lambda formals e0 e1 ...))
       (define-syntax name
         (lambda (x)
           (syntax-case x ()
             [(_ . args)
              #'((lambda formals e0 e1 ...) . args)]
             [else
              (identifier? x)
```

```
                #'residual])))]))
```

The transformer for an integrable procedure now uses `syntax-case` instead of `identifier-syntax`. The first clause of the `syntax-case` form handles the case where an integrable procedure is called directly, and simply replaces the keyword in the operator position of the call with a copy of the `lambda` expression. The second clause of the `syntax-case` form rewrites references that do not appear in operator position as references to the variable `residual`.

Using this implementation of `define-integrable`, only direct calls to integrable procedures are expanded inline, as shown below.

```
(let ()
  (define-integrable g (lambda (x) x))
  (eq? g (g g)))
 →  (let ()
      (define residual (lambda (x) x))
      (define-syntax g
        (lambda (x)
          (syntax-case x ()
            [(_ . args)
             #'((lambda (x) x) . args)]
            [else
             (identifier? x)
             #'residual])))
      (eq? g (g g)))
 →  (let ()
      (define residual (lambda (x) x))
      (eq? residual ((lambda (x) x) residual)))
```

Since non-call references to `g` are rewritten as references to the variable `residual`, this expression evaluates to `#t` as intended. Because a binding for the identifier `residual` is introduced in the expansion of the `define-integrable` macro, the identifier `residual` is automatically renamed. This prevents name conflicts among the temporaries introduced when two or more integrable procedures are defined in the same scope.[5]

Recursion

One problem remains. The preceding implementation of `define-integrable` does not handle recursive definitions well. In particular, the macro expander will not terminate if it encounters a call to a recursive integrable procedure. This problem is solved by the following implementation.

[5] Although it handled internal definitions correctly, the original `syntax-case` implementation did not rename introduced identifiers defined at top level. This inconsistency was eliminated while implementing the module system described in Chapter 3.

```
(define-syntax define-integrable
  (syntax-rules (lambda)
    [(_ name (lambda formals e0 e1 ...))
     (begin
       (define residual (rec name (lambda formals e0 e1 ...)))
       (define-syntax name
         (lambda (x)
           (syntax-case x ()
             [(_ . args)
              #'(fluid-let-syntax ((name (identifier-syntax residual)))
                  ((lambda formals e0 e1 ...) . args))]
             [else
              (identifier? x)
              #'residual]))))]))
```

There are two changes in this implementation. First, the `rec` form is used to construct a local recursive definition for `name` and the result is bound to the variable `residual`. Because the `rec` binding shadows the keyword binding for `name`, recursive calls in the `lambda` body are not expanded. The second change is that a local `fluid-let-syntax` binding is introduced at each expanded call site. This `fluid-let-syntax` form temporarily rebinds `name` to an identifier macro that rewrites any references to `name` encountered while expanding the call as references to `residual`.

The use of `fluid-let-syntax` here is crucial. While `let-syntax` could prevent indefinite expansion of *directly* recursive integrable procedures, it could not prevent indefinite expansion of *mutually* recursive integrable procedures. Consider the following mutually recursive definitions.

```
(define-integrable even
  (lambda (x)
    (or (zero? x) (odd (- x 1)))))

(define-integrable odd
  (lambda (x)
    (not (even x))))
```

Suppose we expand the call `(odd y)`, producing the following.

```
(fluid-let-syntax ((odd (identifier-syntax residual_o)))
  ((lambda (x) (not (even x))) y))
```

The body of the `fluid-let-syntax` construct is expanded in an environment that maps `odd` to an identifier macro that always expands as a reference to $\texttt{residual}_o$. This binding remains in effect until the body of the `fluid-let-syntax` is fully expanded. If `let-syntax` were used instead, this binding would be visible only to lexically apparent references—and there are none—within the body. When the call `(even x)` is expanded we have.

```
((lambda (x)
   (not (fluid-let-syntax ((even (identifier-syntax residual_e)))
          ((lambda (x)
              (or (zero? x) (odd (- x 1))))
```

```
              x))))
   y)
```

Because the fluid binding for `odd` is still in effect, `(odd (- x 1))` becomes `(residual`$_o$ `(- x 1))`. If `let-syntax` were used instead, the temporary binding for `odd` would no longer be visible, and the integrable version of `odd` would expand again, causing the expander ultimately to diverge. In the final output, no recursive references remain.

```
((lambda (x)
   (not ((lambda (x)
           (or (zero? x) (residual_o (- x 1))))
         x)))
 y)
```

For the first time, we have a macro-based implementation of integrable procedures that preserves lexical scoping, avoids duplicate evaluation, maintains pointer equality, and even supports mutual recursion. In fact, bindings established by the final version of `define-integrable` are indistinguishable from those established via `define`, with two exceptions. First, at top level, an integrable procedure must be defined before its first use outside of a macro (or integrable procedure). Second, bindings established by `define` can be mutated using `set!` while those established via `define-integrable` cannot. On balance, these differences are relatively minor irritations.

2.4 Algorithm

The algebra given in this section originally appeared in the article by Dybvig, et al., describing the hygienic macro expansion algorithm [29]. We make minor corrections here and extend the algebra to support the new features of the macro system. The expander is assumed to be part of a standard evaluation process where the value of a program e is obtained by $eval(expand(e, r_{init}))$. The evaluator is invoked only on fully expanded (core-language) programs and does not invoke the expander.

Symbols naming core syntactic forms, such as `quote` and `lambda`, are bound to Special in the initial expand-time store r_{init}; all other symbols are bound initially to Variable. Although it resembles an environment in use, we refer to r as a store in order to prevent confusion between r and the substitution environment. Much as a conventional interpreter maintains an environment that maps variables to locations and a store that maps locations to values, the expander maintains a substitution environment mapping identifiers to labels and a store mapping labels to expand-time values.

Several restrictions have been imposed on the input language to simplify the presentation of the

$$
\begin{aligned}
e &\in \mathit{Exp} \cup \mathit{ConcreteExp} \cup \mathit{CoreExp} \\
c &\in \mathit{Const} \\
s &\in \mathit{Sym} \\
m &\in \mathit{Mark} \\
i &\in \mathit{Identifier} = \{e \mid e \in \mathit{Exp} \text{ and } \mathit{strip}(e) \in \mathit{Sym}\} \\
r &\in \mathit{Store} = \mathit{Sym} \to (\mathit{Transformer} \cup \{\mathsf{Variable},\ \mathsf{PVariable},\ \mathsf{Special}\}) \\
t &\in \mathit{Transformer} = \mathit{Exp} \to \mathit{Exp}
\end{aligned}
$$

$$
\begin{array}{rcll}
\mathit{Exp} & ::= & e & e \in \mathit{ConcreteExp} \\
 & \mid & \mathit{mark}(e, m) & e \in \mathit{Exp} \\
 & \mid & \mathit{subst}(e, i, s) & e \in \mathit{Exp} \\
 & \mid & \mathit{cons}(e_1, e_2) & e_1, e_2 \in \mathit{Exp} \\
 & & & \\
\mathit{ConcreteExp} & ::= & c & \\
 & \mid & s & \\
 & \mid & \mathit{cons}(e_1, e_2) & e_1, e_2 \in \mathit{ConcreteExp} \\
 & & & \\
\mathit{CoreExp} & ::= & \mathit{variable}(s) & \\
 & \mid & \mathit{application}(e_1, e_2) & e_1, e_2 \in \mathit{CoreExp} \\
 & \mid & \mathit{symbolic\text{-}data}(e) & e \in \mathit{Exp} \\
 & \mid & \mathit{function}(s, e) & e \in \mathit{CoreExp} \\
 & & & \\
\mathit{ParsedExp} & ::= & \mathit{variable}(s) & \\
 & \mid & \mathit{application}(e_1, e_2) & e_1, e_2 \in \mathit{Exp} \\
 & \mid & \mathit{symbolic\text{-}data}(e) & e \in \mathit{ConcreteExp} \\
 & \mid & \mathit{function}(s, e) & e \in \mathit{Exp} \\
 & \mid & \mathit{macro\text{-}call}(i, e) & e \in \mathit{Exp} \\
 & \mid & \mathit{pfunction}(s, e) & e \in \mathit{Exp} \\
 & \mid & \mathit{syntax\text{-}data}(i) & \\
 & \mid & \mathit{syntax\text{-}binding}(i, e_1, e_2) & e_1, e_2 \in \mathit{Exp} \\
 & \mid & \mathit{fluid\text{-}syntax\text{-}binding}(i, e_1, e_2) & e_1, e_2 \in \mathit{Exp} \\
 & \mid & \mathit{rec\text{-}syntax\text{-}binding}(i, e_1, e_2) & e_1, e_2 \in \mathit{Exp}
\end{array}
$$

Figure 4: Expression domains.

$$
\begin{array}{rcl}
\textit{expand} & : & \textit{Exp} \times \textit{Store} \rightarrow \textit{CoreExp}
\end{array}
$$

$$
\begin{array}{rcl}
\textit{expand}(e,r) & = & \text{case } \textit{parse}(e,r) \text{ of:} \\
\textit{variable}(i) & \rightarrow & \textit{variable}(\textit{resolve}(i)), \\
\textit{application}(e_1,e_2) & \rightarrow & \textit{application}(\textit{expand}(e_1,r), \textit{expand}(e_2,r)), \\
\textit{symbolic-data}(e) & \rightarrow & \textit{symbolic-data}(\textit{strip}(e)), \\
\textit{syntax-data}(e) & \rightarrow & \textit{symbolic-data}(e), \\
\textit{function}(i,e) & \rightarrow & \textit{function}(s, \textit{expand}(\textit{subst}(e,i,s), r')) \\
 & & \text{where } r' = r[s := \mathsf{Variable}] \text{ and } s \text{ is fresh,} \\
\textit{pfunction}(i,e) & \rightarrow & \textit{function}(s, \textit{expand}(\textit{subst}(e,i,s), r')) \\
 & & \text{where } r' = r[s := \mathsf{PVariable}] \text{ and } s \text{ is fresh,} \\
\textit{macro-call}(i,e) & \rightarrow & \textit{expand}(\textit{mark}(t(\textit{mark}(e,m)),m),r) \\
 & & \text{where } t = r(\textit{resolve}(i)) \text{ and } m \text{ is fresh,} \\
\textit{syntax-binding}(i,e_1,e_2) & \rightarrow & \textit{expand}(\textit{subst}(e_2,i,s), r[s := t]) \\
 & & \text{where } t = \textit{eval}(\textit{expand}(e_1,r)) \text{ and } s \text{ is fresh,} \\
\textit{rec-syntax-binding}(i,e_1,e_2) & \rightarrow & \textit{expand}(\textit{subst}(e_2,i,s), r[s := t]) \\
 & & \text{where } t = \textit{eval}(\textit{expand}(\textit{subst}(e_1,i,s),r)), \text{ and} \\
 & & \qquad s \text{ is fresh} \\
\textit{fluid-syntax-binding}(i,e_1,e_2) & \rightarrow & \textit{expand}(e_2, r[s := t]) \\
 & & \text{where } t = \textit{eval}(\textit{expand}(e_1,r)), \text{ and } s = \textit{resolve}(i),
\end{array}
$$

Figure 5: A macro-expansion algorithm that respects lexical scoping.

$$
\begin{array}{rcl}
\textit{parse} & : & \textit{Exp} \times \textit{Store} \rightarrow \textit{ParsedExp}
\end{array}
$$

$$
\begin{array}{rcll}
\textit{parse}([\![c]\!],r) & = & \textit{symbolic-data}(c) & \\
\textit{parse}([\![i]\!],r) & = & \textit{variable}(i) & \text{if } r(\textit{resolve}(i)) = \mathsf{Variable} \\
\textit{parse}([\![i]\!],r) & = & \textit{macro-call}(i,[\![i]\!]) & \text{if } r(\textit{resolve}(i)) \in \textit{Transformer} \\
\textit{parse}([\![\texttt{(}e_1\ e_2\texttt{)}]\!],r) & = & \textit{application}(e_1,e_2) & \text{if } e_1 \notin \textit{Identifier} \\
\textit{parse}([\![\texttt{(}i\ e\texttt{)}]\!],r) & = & \textit{application}(i,e) & \text{if } r(\textit{resolve}(i)) = \mathsf{Variable} \\
\textit{parse}([\![\texttt{(}i\ e\texttt{)}]\!],r) & = & \textit{macro-call}(i,[\![\texttt{(}i\ e\texttt{)}]\!]) & \text{if } r(\textit{resolve}(i)) \in \textit{Transformer} \\
\textit{parse}([\![\texttt{(quote } e\texttt{)}]\!],r) & = & \textit{symbolic-data}(e) & \text{if } r(\textit{resolve}([\![\texttt{quote}]\!])) = \mathsf{Special} \\
\textit{parse}([\![\texttt{(lambda (}i\texttt{) } e\texttt{)}]\!],r) & = & \textit{function}(i,e) & \text{if } r(\textit{resolve}([\![\texttt{lambda}]\!])) = \mathsf{Special} \\
\textit{parse}([\![\texttt{(plambda (}i\texttt{) } e\texttt{)}]\!],r) & = & \textit{pfunction}(i,e) & \text{if } r(\textit{resolve}([\![\texttt{plambda}]\!])) = \mathsf{Special} \\
\textit{parse}([\![\texttt{(syntax } i\texttt{)}]\!],r) & = & \textit{syntax-data}(i) & \text{if } r(\textit{resolve}([\![\texttt{syntax}]\!])) = \mathsf{Special}, \\
 & & & \text{and } r(\textit{resolve}(i)) \neq \mathsf{PVariable} \\
\textit{parse}([\![\texttt{(syntax } i\texttt{)}]\!],r) & = & \textit{variable}(i) & \text{if } r(\textit{resolve}([\![\texttt{syntax}]\!])) = \mathsf{Special}, \\
 & & & \text{and } r(\textit{resolve}(i)) = \mathsf{PVariable}
\end{array}
$$

$$
\begin{array}{l}
\textit{parse}([\![\texttt{(let-syntax ((}i\ e_1\texttt{)) } e_2\texttt{)}]\!],r) = \textit{syntax-binding}(i,e_1,e_2) \\
\qquad \text{if } r(\textit{resolve}([\![\texttt{let-syntax}]\!])) = \mathsf{Special} \\
\textit{parse}([\![\texttt{(letrec-syntax ((}i\ e_1\texttt{)) } e_2\texttt{)}]\!],r) = \textit{rec-syntax-binding}(i,e_1,e_2) \\
\qquad \text{if } r(\textit{resolve}([\![\texttt{letrec-syntax}]\!])) = \mathsf{Special} \\
\textit{parse}([\![\texttt{(fluid-let-syntax ((}i\ e_1\texttt{)) } e_2\texttt{)}]\!],r) = \textit{fluid-syntax-binding}(i,e_1,e_2) \\
\qquad \text{if } r(\textit{resolve}([\![\texttt{fluid-let-syntax}]\!])) = \mathsf{Special}
\end{array}
$$

Figure 6: A macro-expansion parser.

$$
\begin{aligned}
\mathit{mark} &: \mathit{Exp} \times \mathit{Mark} \rightarrow \mathit{Exp} \\
\mathit{mark}(c, m) &= c \\
\mathit{mark}(\mathit{cons}(e_1, e_2), m) &= \mathit{cons}(\mathit{mark}(e_1, m), \mathit{mark}(e_2, m)) \\
\\
\mathit{marksof} &: \mathit{Identifier} \rightarrow \mathit{MarkSet} \\
\mathit{marksof}(s) &= \emptyset \\
\mathit{marksof}(\mathit{mark}(i, m)) &= \mathit{marksof}(i) \uplus \{m\}, \text{ where } A \uplus B = (A \cup B) - (A \cap B) \\
\mathit{marksof}(\mathit{subst}(i_1, i_2, s)) &= \mathit{marksof}(i_1) \\
\\
\mathit{subst} &: \mathit{Exp} \times \mathit{Identifier} \times \mathit{Sym} \rightarrow \mathit{Exp} \\
\mathit{subst}(c, i, s) &= c \\
\mathit{subst}(\mathit{cons}(e_1, e_2), i, s) &= \mathit{cons}(\mathit{subst}(e_1, i, s), \mathit{subst}(e_2, i, s)) \\
\\
\mathit{resolve} &: \mathit{Identifier} \rightarrow \mathit{Sym} \\
\mathit{resolve}(s) &= s \\
\mathit{resolve}(\mathit{mark}(i, m)) &= \mathit{resolve}(i) \\
\mathit{resolve}(\mathit{subst}(i_1, i_2, s)) &= \begin{cases} s & \text{if } \mathit{marksof}(i_1) = \mathit{marksof}(i_2) \\ & \text{and } \mathit{resolve}(i_1) = \mathit{resolve}(i_2) \\ \mathit{resolve}(i_1) & \text{otherwise} \end{cases} \\
\\
\mathit{expose} &: \mathit{Exp} \rightarrow \mathit{Exp} \\
\mathit{expose}(i) &= i \\
\mathit{expose}(\mathit{mark}(\mathit{cons}(e_1, e_2), m)) &= \mathit{cons}(\mathit{mark}(e_1, m), \mathit{mark}(e_2, m)) \\
\mathit{expose}(\mathit{subst}(\mathit{cons}(e_1, e_2), i, s)) &= \mathit{cons}(\mathit{subst}(e_1, i, s), \mathit{subst}(e_2, i, s)) \\
\\
\mathit{strip} &: \mathit{Exp} \rightarrow \mathit{ConcreteExp} \\
\mathit{strip}(s) &= s \\
\mathit{strip}(c) &= c \\
\mathit{strip}(\mathit{cons}(e_1, e_2)) &= \mathit{cons}(\mathit{strip}(e_1), \mathit{strip}(e_2)) \\
\mathit{strip}(\mathit{mark}(e, m)) &= \mathit{strip}(e) \\
\mathit{strip}(\mathit{subst}(e, i, s)) &= \mathit{strip}(e)
\end{aligned}
$$

Figure 7: Auxiliary functions.

algorithm. Function application is restricted to a single argument. Consequently, binding forms are restricted to a single binding. Generalization to multiple arguments and multiple bindings is straightforward. Conditional branch and variable assignment constructs are omitted since their treatment is also straightforward. For simplicity, `define-syntax` and internal definitions are omitted.[6] A discussion of pattern matching is also beyond the scope of this chapter. We therefore present `plambda`, which binds a single pattern variable within its body, in place of `syntax-case`, which can be defined in terms of `plambda` and *expose*, which is defined in Figure 7. Similarly, we restrict the subform of a `syntax` expression to be a single identifier. If the identifier is a pattern variable, the `syntax` expression yields the value of the pattern variable; otherwise the expression evaluates to symbolic data representing that identifier.

The concrete syntax of expressions ($e \in \mathit{ConcreteExp}$) is defined in Figure 4 as a data type consisting of an unspecified set of constants ($c \in \mathit{Const}$), symbols ($s \in \mathit{Sym}$), and structures built in the usual way by pairing (if $e_1, e_2 \in \mathit{ConcreteExp}$, then $\mathit{cons}(e_1, e_2) \in \mathit{ConcreteExp}$). *Const* includes such traditional Lisp constants as booleans, numbers, and the empty list.

To maintain lexical scoping during translation, the macro system consistently renames bound variables (α-conversion). The renaming process is fully automated by the system and is transparent to the user. A lazy substitution mechanism is used to support renaming. The substitution mechanism used is lazy for two reasons. First, applying substitutions on demand is far more efficient than repeatedly traversing subexpressions to apply substitutions eagerly. In fact, lazy substitution maintains lexical scoping with constant[7] overhead on macro expansion. Second, by delaying substitution until it can determine the role of an input expression, the system avoids traversing (potentially cyclic) structured constants, and it avoids renaming identifiers used as symbolic data. For example, in the body of the following `lambda` expression, the two occurrences of `x` have different roles: the first refers to the formal parameter and must be renamed, while the second is used as symbolic data and must retain its original name.

```
(lambda (x)
  (define-syntax f
    (syntax-rules () ((_ e) (quote e))))
  (eq? x (f x)))
```

As this example illustrates, the role of an input expression cannot be determined until macro expansion of surrounding forms is complete.

To support lazy substitution, the algorithm abandons the traditional identification of variables

[6] Section 3.5 describes handling of internal definitions.
[7] Assuming constant-time environment lookup.

and symbols. Both variables and symbols are now represented by a new kind of object, the *identifier*, which retains its symbolic name and its binding name until the role of the identifier can be determined. Identifiers ($i \in \mathit{Identifier}$) are a subset of the *Exp* domain. An identifier is a symbol that has been subjected to zero or more marking and substitution operations. That is, an identifier is a symbol s, a marked identifier $\mathit{mark}(i, m)$, or a substitution $\mathit{subst}(i_1, i_2, s)$. The *mark* and *subst* constructors are defined in Figure 7. Marks ($m \in \mathit{Mark}$) can be any countably infinite set.[8] The intent of $\mathit{subst}(e, i, s)$ is to replace the binding name of the identifier i in the expression e with the symbol s. In Chapter 3 this substitution machinery is used to control the visibility of identifiers defined within modules.

In the implementation of the macro system, the input and output of macro transformers is represented by syntax objects that, in addition to the usual list-structured program expression, contain a *wrap* that holds the substitutions and marks that apply to that expression. Representing the input and output of macro transformers as syntax objects enables the system to perform α-conversion lazily and transparently. In addition, this representation allows the system to correlate program source and object code through arbitrary user-defined transformations. Macro transformers destructure their input using a pattern-matching facility (`syntax-case`) that exposes list structure within a syntax object and propagates the wrap to the exposed subforms. Syntax objects are destructured only as far as is necessary for pattern matching.

In the algebra, macro transformers ($t \in \mathit{Transformer}$) operate on elements of the *Exp* domain, which extends the usual list-structured source expressions, found in *ConcreteExp*, with substitutions and marks. In the expansion algorithm shown in Figure 5, substitutions are introduced by the core binding forms. For example, when processing $\mathit{function}(i, e)$, corresponding to `(lambda (`i`)` e`)`, *expand* introduces a substitution renaming i to a fresh symbol s within the body e and extends the store to map s to Variable. The substitution is layered on the expression representing the `lambda` body before the body is processed recursively by the expander. Substitutions are propagated to subforms of the body by the function *expose*, defined in Figure 7, which is implicitly applied to the input expression by the parser, defined in Figure 6.

To make the algebra more readable, the parser is defined using a pattern-matching syntax in which *expose* is implicitly applied to the input and the result is matched against the various patterns (set off in $[\![\,]\!]$). Matching of concrete syntax is specified by patterns with the usual print representation for lists. For example, the concrete syntax $\mathit{cons}(e_1, \mathit{cons}(e_2, \texttt{()}))$ is matched by a meta-language

[8] To support separate compilation, our implementation represents marks as strings rather than integers and compares marks using `eq?` since the object-file loader maintains the object identity of strings.

pattern of the form (e_1 e_2). The syntactic categories of meta-language variables used within patterns impose additional constraints on matching. For example, since $i \in$ *Identifier*, a pattern containing i is matched only if the input contains an identifier in the corresponding position.

The *expose* function reveals only the surface list structure of an expression by pushing substitutions and marks down into the constituents of pairs constructed by *cons*. A simple implementation of *expose* is given below.[9]

```
(define expose
  (lambda (e)
    (let expose ([e e] [wrap (lambda (x) x)])
      (cond
        [(or (symbol? e) (constant? e)) (wrap e)]
        [(pair? e) (cons (wrap (car e)) (wrap (cdr e)))]
        [(mark? e)
         (expose (mark-e e) (lambda (x) (wrap (make-mark x (mark-m e)))))]
        [(subst? e)
         (expose (subst-e e)
           (lambda (x) (wrap (make-subst x (subst-i e) (subst-s e)))))]))))
```

Unfortunately, this implementation is rather inefficient because the work of the `wrap` function is duplicated for each level of the list structure that the parser descends. To avoid this overhead, the actual implementation imposes the additional constraint that the expression component of a wrapped expression cannot be another wrapped expression. This property is maintained by representing wrapped expressions with syntax objects, as described earlier, and making the `mark` and `subst` operations combine marks and substitutions when applied to syntax objects. A simple implementation maintaining this property is given below.

```
(define-record syntax-object (exp wrap))
(define-record wrap (marks substs))
(define empty-wrap (lambda () (make-wrap '() '())))

(define expose
  (lambda (e)
    (cond
      [(or (symbol? e) (constant? e) (pair? e)) e]
      [(syntax-object? e)
       (let ([exp (syntax-object-exp e)])
         (if (or (symbol? exp) (constant? exp))
             e
             (cons (make-syntax-object (car exp) (syntax-object-wrap e))
                   (make-syntax-object (cdr exp) (syntax-object-wrap e)))))]
      [else (error 'expose "invalid syntax ~s" e)])))
```

[9] The procedures `cons`, `make-mark`, and `make-subst` correspond to the algebraic constructors *cons*, *mark*, and *subst*. The predicates `pair?`, `mark?`, and `subst?` test whether their input was constructed by `cons`, `make-mark`, or `make-subst`. The accessors `mark-e` and `mark-m` return the e and m components of a construction $mark(e, m)$ and similarly `subst-e`, `subst-i`, and `subst-s` return the e, i, and s components of a construction $subst(e, i, s)$.

```
(define extend-wrap
  (lambda (e extend)
    (if (syntax-object? e)
        (make-syntax-object
          (syntax-object-expression e)
          (extend (syntax-object-wrap e)))
        (make-syntax-object e (extend (empty-wrap))))))

(define mark
  (lambda (e m)
    (extend-wrap e (lambda (wrap) (add-mark m wrap)))))

(define subst
  (lambda (e i s)
    (extend-wrap e (lambda (wrap) (add-subst i s wrap)))))

(define add-mark
  (lambda (m wrap)
    (make-wrap
      (let ((mks (wrap-marks wrap)))
        (if (and (pair? mks) (eq? m (car mks)))
            (cdr mks)
            (cons m mks)))
      (map (lambda (s) (cons (mark (car s) m) (cdr s)))
           (wrap-substs wrap)))))

(define add-subst
  (lambda (i s wrap)
    (make-wrap (wrap-marks wrap) (cons (cons i s) (wrap-subst wrap)))))
```

A syntax object maintains a wrap that collects the marks and substitutions that apply to the expression. Given this representation, the `expose` function can push all the pending marks and substitutions down onto the constituents of the syntax-object expression in constant time. Similarly, the `mark` and `subst` operations have direct access to any pending marks or substitutions and can extend these sets efficiently.

During translation, identifier references are resolved through a two-step process. In effect, identifier references are resolved via a two-level map consisting of a substitution environment and a store. First, the binding name of the identifier is determined using *resolve*, defined in Figure 7, to apply any pending substitutions to the identifier. Substitutions are introduced by core binding forms such as `lambda`, `let-syntax`, and `letrec-syntax`. Substitutions are keyed with marks to prevent substitutions for identifiers introduced in the expansion of a macro from affecting identifiers not introduced by the macro. Marks are introduced when the expander invokes a macro transformer (see Figure 5). A fresh mark is applied to the input of the transformer, and the same mark is applied to the output of the transformer. Because identical marks cancel when they meet (see *marksof*, defined in Figure 7), marks adhere only to expressions introduced in the expansion of a macro. When processing an expression of the form $subst(i_1, i_2, s)$, *resolve* renames identifier i_1 to s only if the set of marks for i_1 is identical to the set of marks associated with the identifier i_2 and both i_1 and i_2 resolve to the same symbol.

In the second step of the identifier resolution process, the parser consults r, the expand-time store, to find the binding of the symbol returned by *resolve*. The store maps symbols to expand-time bindings that describe the roles of the corresponding identifiers in the input program. The parser uses the store to resolve ambiguity in the concrete syntax of the program. For example, in the concrete syntax of programs, symbols represent both variables and keywords. If the substitution environment and store map a symbol to a macro transformer ($t \in \mathit{Transformer}$), then the symbol represents a syntactic keyword. If the symbol maps to *variable*, then it represents a program variable. The store is also used to detect attempts to reference run-time bindings within code that is evaluated during macro expansion, *i.e.*, transformer expressions. The store is extended only by the expander and consulted by the parser and expander.

Two identifiers i_1 and i_2 satisfy `free-identifier=?` if and only if $\mathit{resolve}(i_1) = \mathit{resolve}(i_2)$. Two identifiers i_1 and i_2 satisfy `bound-identifier=?` if and only if $\mathit{resolve}(\mathit{subst}(i_1, i_2, s)) = s$ where s is a fresh symbol.

Within programs, macro transformers are simply `lambda` expressions whose chief distinction is that they may be evaluated and applied at translation time. Syntax objects commonly appear in the expanded code of these transformers. For example, a syntax object representing the identifier `lambda` appears within the expanded code of the transformer for the following `let` macro.

```
(define-syntax let
  (syntax-rules ()
    ((_ ((x v) ...) e0 e1 ...)
     ((lambda (x ...) e0 e1 ...) v ...))))
```

This syntax object contains the substitution environment and marks in effect where `let` was defined. Therefore `let` expands into a reference to the binding for `lambda` visible in the scope where `let` was defined. The need to represent identifiers, which may contain marks and substitutions, within the expanded code of transformers, means that $\mathit{symbolic\text{-}data}(e) \in \mathit{CoreExp}$ implies $e \in \mathit{Exp}$ (rather than $\mathit{ConcreteExp}$).

2.4.1 Intentional Capture

The macro system supports intentional capture via `fluid-let-syntax` and `datum->syntax-object`, described in Sections 2.2.2, 2.2.4, and 2.3.4. The form of capture provided by `fluid-let-syntax`

$$\begin{aligned}
\mathit{imp\text{-}id} &: \mathit{Identifier} \times \mathit{Sym} \rightarrow \mathit{Identifier} \\
\mathit{imp\text{-}id}(s_1, s_2) &= s_2 \\
\mathit{imp\text{-}id}(\mathit{mark}(i, m), s) &= \mathit{mark}(\mathit{imp\text{-}id}(i, s), m) \\
\mathit{imp\text{-}id}(\mathit{subst}(i_1, i_2, s_1), s_2) &= \mathit{subst}(\mathit{imp\text{-}id}(i_1, s_2), i_2, s_1)
\end{aligned}$$

Figure 8: Construction of implicit identifiers.

complements that provided by `datum->syntax-object`. A straightforward generalization of the function *imp-id* defined in Figure 8, `datum->syntax-object` allows the user to manipulate the mapping from identifiers to binding names (*resolve*). Its counterpart, `fluid-let-syntax`, allows the user to manipulate the mapping from binding names to expand-time values (r). Together they permit the user some measure of control over the two-level mapping that determines the meaning of identifiers at expand-time.

Manipulating the substitution environment

Given a template identifier i and a symbol s, the function *imp-id* returns a new identifier containing the marks and substitutions of i but having s as its symbolic name. In effect, the procedure `datum->syntax-object` applies *imp-id* to all symbols contained within the datum passed as its second argument. While a literal implementation of this semantics would be inefficient, the lazy substitution mechanism makes an alternative constant-time implementation possible. Using the syntax-object representation from page 48, for example, `datum->syntax-object` can be implemented efficiently as follows.

```
(define datum->syntax-object
  (lambda (id datum)
    (if (identifier? id)
        (make-syntax-object datum (syntax-object-wrap id))
        (error 'datum->syntax-object "~s is not an identifier" id))))

(define identifier?
  (lambda (x)
    (and (syntax-object? x) (symbol? (syntax-object-expression x)))))

(define syntax-object->datum syntax-object-exp)
```

The substitution environment can be manipulated by combining `syntax-object->datum` and `datum->syntax-object`. The former extracts the syntax-object expression, discarding the wrap, and the latter can be used to apply a new wrap to the extracted datum.[10] When `expose` is eventually applied to the object returned by `datum->syntax-object`, the new substitutions are applied.

[10] These procedures permit a limited form of reflection on syntax objects.

One consequence of supporting `datum->syntax-object` is that syntax-object wraps must retain some apparently useless substitutions. For example, while a substitution renaming `a` to `b` has no effect when applied to the symbol `x`, that substitution must be retained in the wrap of the syntax object encapsulating `x` in case `datum->syntax-object` is used to apply the wrap of this syntax object to a datum containing the symbol `a`. While this may increase the size of syntax objects, it is not a problem in practice because the structures are shared. Substitutions that fail due to mismatched marks can safely be discarded.

Manipulating the expand-time store

For each of the core binding forms except `fluid-let-syntax`, the expander introduces substitutions renaming the bound identifiers. The substitutions are introduced in the handling of *function*, *pfunction*, *syntax-binding*, and *rec-syntax-binding* in Figure 5. Consider the rule for `let-syntax` given below.

$$\begin{array}{lcl} \textit{syntax-binding}(i, e_1, e_2) & \rightarrow & \textit{expand}(\textit{subst}(e_2, i, s), r[s := t]) \\ & & \text{where } t = \textit{eval}(\textit{expand}(e_1, r)) \text{ and } s \text{ is fresh} \end{array}$$

This rule uses *subst* to rename the bound identifier to a fresh symbol s and then extends the store r so that it maps s to the transformer t obtained by evaluating the expanded code of e_1. In effect, `let-syntax` binds a fresh "location" that is accessible only through a renamed identifier that is visible only within the body. Contrast this with the following rule for `fluid-let-syntax`.

$$\begin{array}{lcl} \textit{fluid-syntax-binding}(i, e_1, e_2) & \rightarrow & \textit{expand}(e_2, r[s := t]) \\ & & \text{where } t = \textit{eval}(\textit{expand}(e_1, r)), \text{ and } s = \textit{resolve}(i) \end{array}$$

In this rule, i is not renamed and the store is extended to map the symbol returned by *resolve*(i) to the new transformer t. Thus, `fluid-let-syntax` rebinds the existing "location" s, which is accessible via the identifier i wherever it appears in the program. Because the store is threaded through the expander, the store can be "updated" functionally. The update applies only during the recursive call to expand, i.e., during the expansion of the body. Fluid binding follows naturally from the usual lookup on r. Lexical scoping is preserved because s is obtained via *resolve*(i), which respects existing α-substitutions for i. In particular, the only identifiers affected by the temporary binding are those that refer to the same lexical variable that i refers to. It is interesting to note that, because they do not perform α-conversion, traditional macro systems effectively implement dynamic scoping.

2.5 Collecting Information

As it translates an input program into the core language, the macro expander can collect several pieces of useful information. First, the expander can annotate core-language terms with source information extracted from its input. Second, the expander can distinguish between global and local definitions and references. Third, the expander can construct links connecting each lexical variable reference to the corresponding formal parameter declaration. Finally, the expander can record conservative information about the use of variables.

To record this additional information, the expander constructs annotated records representing the abstract syntax of the core-language program. This provides a natural interface between the expander, which necessarily constructs syntactically well-formed terms, and the rest of the compiler, which expects syntactically well-formed terms. It is straightforward to provide a translator that converts the record structures into the usual list-structured representation expected by some other clients of the expander.

2.5.1 Source Information

To preserve source information, the algorithm described in Section 2.4 is extended to operate on annotated terms in an extended term structure Exp' and produce annotated terms in an extended structure $\mathit{CoreExp}'$ defined below.

$$
\begin{array}{rcll}
a & \in & \mathit{Annotation} & \text{source annotations} \\[1ex]
\mathit{Exp}' & ::= & e & e \in \mathit{Exp} \\
 & | & \mathit{annotation}(e, a) & e \in \mathit{Exp}' \text{ and } a \in \mathit{Annotation} \\[1ex]
\mathit{CoreExp}' & ::= & e & e \in \mathit{CoreExp} \\
 & | & \mathit{annotation}(e, a) & e \in \mathit{CoreExp}' \text{ and } a \in \mathit{Annotation} \\[1ex]
\mathit{ParsedExp}' & ::= & e & e \in \mathit{ParsedExp} \\
 & | & \mathit{annotation}(e, a) & e \in \mathit{ParsedExp}' \text{ and } a \in \mathit{Annotation}
\end{array}
$$

When the expander or parser encounters an annotated expression, it processes the expression as before and wraps the result with the annotation on the input expression.

$$\begin{aligned}
\mathit{expand}' &: \mathit{Exp}' \times \mathit{Store} \rightarrow \mathit{CoreExp}' \\
\mathit{expand}'(e,r) &= \mathit{annotation}(\mathit{expand}'(e_1,r),a), \text{ if } \mathit{parse}'(e,r) = \mathit{annotation}(e_1,a)
\end{aligned}$$

otherwise, exactly as *expand* with all occurrences of *expand* replaced by *expand′*

$$\begin{aligned}
\mathit{parse}' &: \mathit{Exp}' \times \mathit{Store} \rightarrow \mathit{ParsedExp}' \\
\mathit{parse}'(e,r) &= \begin{cases} \mathit{annotation}(\mathit{parse}(e_1,r),a) & \text{if } e = \mathit{annotation}(e_1,a) \\ \mathit{parse}(e,r) & \text{otherwise} \end{cases}
\end{aligned}$$

Source information is carefully maintained by subsequent passes of the compiler. More details are given elsewhere [63, 64].

Annotations are layered on the input by an alternative implementation of the `read` procedure that tracks the current file name and the current position within that file. The modified `read` procedure annotates each identifier or constant and each pair (*cons*) corresponding to the start of an S-expression. For example, given the input string `"(1 2)"`, the modified `read` returns the following.

$$\mathit{annotation}(\mathit{cons}(\mathit{annotation}(1,a_1),\mathit{cons}(\mathit{annotation}(2,a_2),())),a_0)$$

Annotating each *cons* would substantially increase the overhead of collecting source information without materially improving the quality of the information retained after macro expansion.

Since the macro system is careful to preserve the object identity of constants, it is important to preserve this property even when collecting source information. For example, collecting source information should not affect the behavior of the following macro.

```
(define-syntax test-eq?
  (syntax-rules ()
    [(_ exp) (eq? exp exp)]))
```

```
(test-eq? (list 1 2))    →  (eq? (list 1 2) (list 1 2))    ⇒  #f
(test-eq? '(1 2 3 4))    →  (eq? '(1 2 3 4) '(1 2 3 4))    ⇒  #t
```

Because the constant argument to `test-eq?` is not copied, the expression `(test-eq? '(1 2 3 4))` evaluates to `#t`.

Annotations within structured constants are removed by *strip*. To detect and preserve cycles and shared structure within complex constants, *strip* updates a mutable field within the annotation records it encounters. This field is examined when *strip* encounters an annotation. If a structured object is found, it is returned. Otherwise, *strip* processes the annotated expression, passing the annotation record in as the parent. When allocating an object, such as a pair, *strip* installs that object in the mutable field of the parent object, if any, before recursively processing the fields of the child object and updating the corresponding fields of the newly allocated object. The order of these operations prevents the algorithm from looping indefinitely within a cycle.

The current implementation records four pieces of information about each annotated expression: the byte offset of its starting position within the source file, the filesystem path to the file, the length of the file, and a checksum. The byte offset can be used to quickly retrieve the text of an expression. The last three pieces of information are factored into a shared structure allocated once per file. This permits efficient use of memory and simplifies the task of supporting other kinds of source information, such as location within an edit buffer of an interactive environment. The file length and checksum can be used by the system to distinguish like-named files that differ and to verify that offsets into the file still refer to the same text that they referred to at the time the file was compiled. This is important when providing feedback to the user in terms of the original source. The file length provides a fast way to detect certain file modifications; the checksum provides a robust way to detect file modifications and is used only when file lengths are equal.[11] Both are deliberately independent of filesystem time stamps so that source information can still be exploited in cases where a file has been temporarily modified and then restored.

2.5.2 Conservative Analysis

The expander maintains a substitution environment and an expand-time store that are used to recognize and rename bound identifiers and to determine whether an identifier represents a variable, a keyword, a pattern variable, or a core form. The parser in Figure 6 recognizes an identifier as a variable reference when the identifier resolves to a symbol that is bound to Variable in the expand-time store. All symbols not bound to Special are bound to Variable in the initial store r_{init} and, when processing a `lambda` expression, the expander extends the store to map the renamed formal parameter to Variable.

In an implementation of the macro expander, it is useful to bind all non-Special symbols in the initial store to *Global* and, when processing a `lambda` expression, extend the store so that it maps the renamed formal parameter to a mutable *lexvar* structure. This enables the expander to distinguish references and assignments to global variables from references and assignments to local variables. This information is collected naturally when the parser consults the expand-time store. Because the expander produces record structures as its output, the collected information is easily made available to subsequent passes of the compiler.

When the expander encounters a reference or assignment to a bound variable, it stores a pointer

[11] When the size of the file on disk exceeds the recorded file length, we could compute the checksum up to the original file length to recognize cases in which the file has merely been extended without altering the original content.

to the corresponding *lexvar* structure in the abstract syntax record it constructs. The expander stores a pointer to the same *lexvar* structure in the output record representing the `lambda` expression binding that variable. Subsequent passes of the compiler can exploit this representation to implement fast compile-time environment lookup. This representation also permits the expander to collect conservative information about the use of variables. For example, the expander can increment a reference count field within the *lexvar* structure each time it residualizes a reference to the corresponding variable. Similarly, an "assigned" flag in the *lexvar* structure can be set when the expander residualizes an assignment to the variable. This information is necessarily conservative because the expander is not sensitive to control flow: syntactically apparent variable assignments may, in fact, never be executed when the program is evaluated.

2.5.3 Application

The information collected by the expander can be used by the compiler, by the development environment, and by the run-time system. Source information can be used to provide helpful error messages that direct the user to the precise location of the error in the source code. Correlating source and object code is particularly useful because extensive macro rewrites can obscure this relationship. It could be quite difficult for a user to find the source of program errors given only the expanded code as a guide. Source correlation also makes it possible to support source-level debugging including single-steppers and interactive breakpoints. Though not described here, we have implemented visualization tools that present a source-level view of static and dynamic program properties using the source-correlation framework outlined in this section [63, 64].

Many development environments now display source programs using different fonts and colors for different tokens. A powerful macro facility makes it difficult to display programs accurately because it is not clear until macro expansion is complete whether a particular identifier represents a variable reference or symbolic data. It is possible, however, to invoke the expander on completed expressions and traverse the resulting record structures to determine how to display the corresponding source expressions.

The *lexvar* structures introduced by the expander also have a range of applications that spans the compiler and development environment. For example, we have built simple browsers that support hyper-text navigation of source using the links in the record output of the expander to jump from variable references to their bindings and back. Interpreters can use these links to support constant-time variable lookup. This is useful even in the context of a compiler since many analyses can be cast in the framework of abstract interpretation [18]. The conservative variable usage information collected by the expander can be refined by subsequent flow analysis. It may appear frivolous to collect conservative information here when more accurate information can be obtained via flow analysis, but the expander collects its information at very little cost and the results can improve the run time of expensive analyses.

Chapter 3

Module System

Module systems and lexically scoped syntactic abstraction (macro) facilities have been well-studied in isolation [17, 22, 29, 30, 31, 43, 44, 45, 46, 52, 61]. In the past several years there has been increasing interest in combining modules and lexically scoped macros [19, 50, 51, 53, 56]. This chapter describes the first fully implemented system that allows arbitrary composition of module and macro forms, supports separate compilation, and supports fully general macro transformations while maintaining lexical scoping for all macros. Section 3.1 provides an overview of the module system. Section 3.2 describes our design goals. Section 3.3 describes our extended language that supports both modules and syntactic abstraction. Section 3.4 demonstrates the synergy between modules and syntactic abstraction with a series of short examples. Section 3.5 describes the implementation. Section 3.6 addresses practical concerns such as separate compilation and support for static program analysis in the presence of powerful macro systems. Section 3.7 presents extended examples showing how mechanisms for dynamic linking and object-oriented programming can be implemented using macros and modules. Section 3.8 discusses related work.

3.1 Introduction

A *module system* provides control over the scopes of bindings. A module is a named lexical scope encapsulating a set of identifier bindings. The *exported* bindings of a module can be made visible in another scope by *importing* the module. A binding is otherwise visible only within the module in which it is defined. A key benefit of modular structure is that it assists both humans and machines in understanding programs by limiting amount of code that must be understood. To use a module, a programmer needs to understand its *interface*, the set of exported bindings, but need not understand the details of its implementation. When writing a module, the programmer defines the interface that controls which bindings are visible outside the module. This structure permits the programmer considerable flexibility in the implementation and benefits the compiler as well. For example, knowing that a particular procedure is not accessible outside the module, the compiler may select a more efficient calling convention for that procedure.

A *syntactic abstraction* facility can be used to simplify and standardize repeated patterns in programs, to introduce syntactic forms with new evaluation rules, and to embed one language within another. A key benefit of syntactic abstraction is that the semantics of new constructs are declaratively specified by translation into familiar constructs in the source language. Syntactic abstractions are implemented as *macros* describing local source-to-source transformations that are applied to a program at compile time. Because macros are expanded at compile time, their use generally incurs no run-time overhead.

Combining module systems and lexically scoped macro facilities expands the capabilities of both. The system described in this chapter extends the small core language and powerful syntactic abstraction mechanisms of the `syntax-case` system described in Chapter 2 with support for modules. A module is a named scope that encapsulates a set of identifier bindings. Importing from a module makes these identifier bindings visible in the importing context. Modules control visibility of bindings and can be viewed as extending lexical scoping to allow more precise control over where bindings are or are not visible. Modules export identifier bindings, i.e., variable bindings, keyword bindings, or module name bindings. The primitive module and import forms are simple, yet can be extended via syntactic abstraction. They may appear at top level, within local scopes, or nested within other modules. A program containing module and import forms can be understood statically in terms of a straightforward translation to core language forms.

3.2 Design Considerations

This section briefly examines the requirements that shaped the design of the module system.

- A module should hide the details of its implementation.
- The module system should be useful for structuring large (or small) programs.
- The module system should facilitate code reuse.
- It should be possible to compile modules separately.

- The module system should be a compatible extension of Scheme.
- Modules should be expressive.
- Macros and modules should be seamlessly integrated.
- The system must permit efficient implementation.

The ability to hide implementation details makes modules useful for structuring large systems. Large systems are often composed of independent modules each of which may have been written by different programmers. To coordinate development, the authors and users of a module agree on an externally visible interface to the functionality of the module, and this contract is then enforced by the scoping rules of the module system. This prevents users from building dependencies on implementation details into their code, and it allows the authors to revise the implementation of a module without breaking user code.

Separate compilation originated as a way to reduce compilation time by recompiling only the modules that changed during development. Reducing compile time can improve programmer productivity. This is particularly important in developing large systems where compile time might otherwise be excessive. The ability to link separately compiled modules allows the software development process to be distributed even if the source code is not. Distributing modules as object code, rather than source code, reinforces the abstraction barriers imposed by the module system. Just as scoping limits access to module internals, compilation limits scrutiny of those internals.

Support for separate compilation and information hiding are common goals of most module systems. Several additional goals arise when designing a module facility for an incrementally compiled Scheme system. In particular, because it is being grafted onto a language with a long history, the module system should be a compatible extension of Scheme. Existing programs should remain valid in the extended language so that existing code can be maintained and reused. A solution that cripples `eval`, for example, is not an option.

Scheme is hailed as a highly expressive language. Any proposed module system should enhance this expressive character and not detract from it. Few new core constructs should be required. Others should be derived as syntactic abstractions. Indeed, because syntactic abstraction is a powerful and often-used tool, modules and macros should be seamlessly integrated. Modules can help structure the use of macros and vice versa. Modules must be able to export macros defined within them, and macros must be able to expand into module definitions and imports. In particular, it should be possible to use a macro in the same way whether inside or outside a module.

To survive in a production environment, the module system must admit efficient implementation. The module facility itself must be implemented efficiently and the compiler must generate efficient code for modular programs. In fact, one motivation for developing a module system is to provide tools for structuring programs in ways that assist compile-time analysis. Complex analysis should not be necessary simply to maintain the level of performance obtained without modules; rather it should improve performance. As part of a portable macro system the module facility has potential for wide-spread use. Unless efficient code can be generated for modular programs, however, the module system will not realize this potential.

3.3 Language

This section describes the syntax, scoping rules, and semantics of the module language.

3.3.1 Syntax

The module language extends the core language described in Section 2.2. To support modules, the existing set of definitions is extended with three new definition forms: `module`, `import`, and `import-only`.

```
⟨definition⟩ ⟶ (define ⟨variable⟩ ⟨expression⟩)
     | (define-syntax ⟨keyword⟩ ⟨expression⟩)
     | (begin ⟨definition⟩*)
     | ⟨derived definition⟩
     | (module ⟨module name⟩ ⟨interface⟩ ⟨definition⟩* ⟨expression⟩*)
     | (import ⟨module name⟩)
     | (import-only ⟨module name⟩)
⟨interface⟩ ⟶ (⟨identifier⟩*)
⟨module name⟩ ⟶ ⟨identifier⟩
```

A module consists of a (possibly empty) set of definitions and a (possibly empty) sequence of initialization expressions. The initialization expressions of any modules within a body are evaluated immediately after the last definition form in the body. Each identifier listed in a module's interface must be defined within that module. Because `module`, `import`, and `import-only` are definitions, they may appear wherever other definitions may appear: at the top level of a program, nested within a `module` or `lambda` body, or nested within derived forms that expand into `module` or `lambda` forms.

Although the language does not provide primitive constructs for separating interfaces from implementations, importing only selected identifiers from a module, or renaming imported identifiers, Section 3.4 demonstrates how such constructs can be obtained via syntactic abstraction.

3.3.2 Scope

Except where shadowed, identifiers defined at top level are visible throughout the program, while identifiers defined within a body are visible only within that body.

A `module` definition introduces a named lexical scope. Module names occupy the same namespace as other identifiers and follow the same scoping rules. Unless exported, identifiers defined within a module are visible only within that module. Identifiers exported from a module are visible within the module and where the module is imported. A module is imported via an `import` of the module's name. An identifier made visible by `import` is scoped as if its definition appears where the `import` form appears. The following example illustrates these scoping rules.

```
(let ((x 1))
  (module M (x setter)
    (define-syntax x (identifier-syntax z))
    (define setter (lambda (x) (set! z x)))
    (define z 5))
  (let ((y x) (z 0))
    (import M)
    (setter 3)
    (list x y z)))  ⇒  (3 1 0)
```

The inner `let` expression binds `y` to the value of the `x` bound by the outer `let`. The import of `M` makes the definitions of `x` and `setter` visible within the inner `let`. In the expression `(list x y z)`, `x` refers to the identifier macro exported from `M`, while `y` and `z` refer to the bindings established by the inner `let`. The identifier macro `x` (defined using the syntactic abstraction `identifier-syntax`) expands the reference to `x` into a reference to the variable `z` defined within the module.

3.3.3 Semantics

The semantics of a program containing modules can be completely understood in terms of a static translation into core forms. Before it is compiled, a source program is translated into a core language program containing no syntactic abstractions, syntactic definitions, module definitions, or import forms. Translation is performed by a *syntax expander* that processes the forms in the source program via recursive descent. The semantics of the core language is defined in the Revised^5 Report on Scheme.

As described in Chapter 2, a `define-syntax` form associates a keyword with a transformer in an expand-time environment and store. When the expander encounters a keyword, it invokes the associated transformer and reprocesses the resulting form. A `module` form associates a module name with an interface. When the expander encounters an `import` form, it extracts the corresponding module interface from the expand-time store and makes the exported bindings visible in the scope where the `import` form appears.

Definitions within a `lambda` or `module` body are processed from left to right so that a module's definition and import may appear within the same sequence of definitions. Expressions appearing within a body and the right-hand sides of variable definitions, however, are translated only after the entire set of definitions has been processed, allowing full mutual recursion among variable and syntactic definitions. Processing definitions from left to right has the added benefit that derived definitions, such as `define-record`, may be defined and used within the same body, provided they are defined before use.

Module and import forms affect only the visibility of identifiers in the source program, not their denotations. In particular, variables are bound to locations whether defined within or outside of a module, and `import` does not introduce new locations.

Local variables are renamed as necessary to preserve the scoping relationships established by both modules and syntactic abstractions. Thus, the module program in Section 3.3.2 is equivalent to the following program in which identifiers have been consistently renamed as indicated by subscripts.

```
(let_0 ([x_1 1])
  (define-syntax_0 x_2 (identifier-syntax_0 z_2))
  (define_0 setter_2 (lambda_0 (x_3) (set!_0 z_2 x_3)))
  (define_0 z_2 5)
  (let_0 ([y_4 x_1] [z_4 0])
    (setter_2 3)
    (list_0 x_2 y_4 z_4)))
```

Both programs are equivalent to the core-language program below in which syntactic abstractions have been expanded.

```
((lambda_0 (x_1)
  (define_0 setter_2 (lambda_0 (x_3) (set!_0 z_2 x_3)))
  (define_0 z_2 5)
  ((lambda_0 (y_4 z_4)
     (setter_2 3)
     (list_0 z_2 y_4 z_4))
   x_1
   0))
 1)
```

The mechanisms that perform expansion and renaming are described in Sections 2.4 and 3.5.

Although definitions within a `lambda` or `module` body are processed from left to right by the expander, the order in which variable definitions are evaluated is left unspecified as in the core language. Initialization expressions appearing within a `module` body are evaluated in sequence after the evaluation of the variable definitions.

3.4 Examples

The examples in this section show that many useful module constructs can be derived from the primitive `module` and `import` forms via syntactic abstraction.

3.4.1 Qualified Reference

It is often convenient to refer to one export of a module without importing all of its exports. While our system does not provide an explicit construct for this purpose, a qualified reference macro can easily be defined and used as follows.

```
(define-syntax from
  (syntax-rules ()
    ((_ M id) (let () (import M) id))))

(let ((x 10))
  (module m1 (x) (define x 1))
  (module m2 (x) (define x 2))
  (list (from m1 x) (from m2 x)))  ⇒  (1 2)
```

Because the module system supports nested modules, it is useful to provide a fully qualified reference form that supplies the path to the identifier of interest. The `from` macro defined above can be generalized to support these fully qualified paths very simply with the addition of one line.

```
(define-syntax from
  (syntax-rules ()
    ((_ M id) (let () (import M) id))
    ((_ M more ...) (let () (import M) (from more ...)))))

(let ()
  (module m1 (a m2)
    (define a 1)
    (module m2 (a m3)
      (define a 2)
      (module m3 (a)
        (define a 3))))
  (list (from m1 a) (from m1 m2 a) (from m1 m2 m3 a)))  ⇒  (1 2 3)
```

3.4.2 Anonymous Modules

Often it is convenient to avoid naming a module that is imported only in the block where it is defined. The following macro supports both named and *anonymous* modules. Identifiers exported by an anonymous module are visible where the module definition appears; non-exported identifiers are visible only within that module. An anonymous module is the equivalent of the `local` construct in Standard ML [33].

```
(define-syntax module*
  (syntax-rules ()
    [(_ (id ...) form ...)
     (begin
       (module* tmp (id ...) form ...)
       (import tmp))]
    [(_ name (id ...) form ...)
     (module name (id ...) form ...)]))
```

An anonymous module definition expands into a sequence containing the definition and immediate import of an equivalent module named `tmp`. Because the macro expander automatically renames introduced identifiers, the name `tmp` is visible only to the `import` form introduced by the macro.

3.4.3 Selective Import and Renaming

While the `import` construct of the core language does not directly support renaming of imported bindings or selective import of specific bindings, the following macro does provide these capabilities.

```
(define-syntax import*
  (syntax-rules ()
    [(_ M (new old))
     (module* (new)
       (define-alias new tmp)
       (module* (tmp)
         (import M)
         (define-alias tmp old)))]
    [(_ M id) (module* (id) (import M))]
    [(_ M spec ...) (begin (import* M spec) ...)]))
```

To selectively import an identifier from module `M`, the macro expands into an anonymous module that first imports all exports of `M` then re-exports only the selected identifier. To rename on import, the macro expands into an anonymous module that instead exports an alias bound to the new name. The alias is simply an identifier macro that expands into a reference to the old name visible where the alias is defined (see page 27).

If the macro placed the definition of `new` in the same scope as the import of `M`, and `new` is also present in the interface of `M`, a naming conflict would arise. To prevent this, the macro instead places the import within a nested anonymous module and links `old` and `new` by means of an alias for the introduced identifier `tmp`.

The macro expands recursively to handle multiple import specifications. Thus the following construct imports x, y (renamed as z), and z (renamed as y) from module m.

```
(import* m x (y z) (z y))
```

3.4.4 Multiple Views on Modules

It can be useful to present several alternative views on a single module. For example, when clients of a module are trusted to different degrees, we may be willing to export more bindings to some than to others. Because a module can re-export imported bindings, it is easy to provide several views on a module. In the following example, the interfaces of modules A and B present different subsets of the interface of module C.

```
(module C (a b c d)
  (define a 1)
  (define b 2)
  (define c 3)
  (define d 4))

(module A (a b) (import C))
(module B (a b c) (import C))
```

The pattern above is easily generalized using the following syntactic abstraction.

```
(views ([A (a b)]
        [B (a b c)])
  (define a 1)
  (define b 2)
  (define c 3)
  (define d 4))
```

Because modules can be exported from other modules, this abstraction is easily implemented by the following macro.[1]

```
(define-syntax views
  (syntax-rules ()
    [(_ ([m (id ...)] ...) form ...)
     (module* (m ...)
       (module tmp (id ... ...) form ...)
       (module m (id ...) (import tmp)) ...)]))
```

The ability to re-export imported bindings, also makes it quite easy to combine several modules into a compound, as M3 does below.[2]

[1] While it would not be difficult to eliminate duplicate identifiers from the interface of the module tmp, it is not necessary in the current implementation of the module system.

[2] The identifiers A.y and B.y are in no way special. They are simply descriptive names for the exported identifiers that would otherwise clash.

```
(module M1 (x y)
  (define x 1) (define y 2))
(module M2 (y z)
  (define y 3) (define z 4))
(module M3 (x A.y B.y z)
  (import* M1 x (A.y y))
  (import* M2 (B.y y) z))
```

3.4.5 Mutually Recursive Modules

Mutually recursive modules can be defined in several ways. In the following program, `A` and `B` are mutually recursive modules exported by an anonymous module whose local scope is used to statically link the two. For example, the free variable `y` within module `A` refers to the binding for `y` provided by the import of `B` in the enclosing module.

```
(module* (A B)
  (module A (x) (define x (lambda () y)))
  (module B (y) (define y (lambda () x)))
  (import A)
  (import B))
```

The following syntactic abstraction generalizes this pattern to permit the definition of multiple mutually recursive modules.

```
(define-syntax rec-modules
  (syntax-rules ()
    ((_ (module N (id ...) form ...) ...)
     (module* (N ...)
       (module N (id ...) form ...) ...
       (import N) ...))))
```

Using this macro makes mutually recursive module definitions more readable, as shown below.

```
(rec-modules
  (module A (x) (define x (lambda () y)))
  (module B (y) (define y (lambda () z)))
  (module C (z) (define z (lambda () x))))
```

Recursive modules residing in different source files can be combined using the `include` macro. For example, if a program consists of two modules, `m1` and `m2`, residing in files `m1.ss` and `m2.ss`, they can be combined as follows:

```
(module* (m1 m2)
  (include "m1.ss")
  (include "m2.ss")
  (import m1)
  (import m2))
```

As shown below, the `rec-modules` macro can be adapted to support modules located in different source files.

```
(define-syntax rec-modules
  (syntax-rules ()
    [(_ include (m file) ...)
     (module* (m ...)
       (include file) ...
       (import m) ...)]))
```

Using this version of `rec-modules`, the example given earlier can be written as follows.

```
(rec-modules include
  (m1 "m1.ss")
  (m2 "m2.ss"))
```

This model for linking recursive modules requires the modules to appear or be included in the same compilation unit. Section 3.4.8 develops a framework for separately compiled, mutually recursive mdoules.

3.4.6 Separating Interface from Implementation

To allow interfaces to be separated from implementations, the following macros support the definition and use of named interfaces.

```
(define-syntax define-interface
  (syntax-rules ()
    [(_ name (export ...))
     (define-syntax name
       (lambda (x)
         (syntax-case x ()
           [(_ n defs)
            (with-implicit (n export ...)
              #'(module n (export ...) .
                  defs))])))]))

(define-syntax define-module
  (syntax-rules ()
    [(_ name interface defn ...)
     (interface name (defn ...))]))
```

`define-interface` creates an interface macro that, given a module name and a list of definitions, expands into a module definition with a concrete interface. These macros can be used as follows.

```
(define-interface simple (a b))
(define-module M simple
  (define-syntax a (identifier-syntax 1))
  (define b (lambda () c))
  (define c 2))
(let () (import M) (list a (b)))  ⇒  (1 2)
```

The `with-implicit` macro is used in the definition of `define-interface` to ensure that the introduced `export` identifiers are visible in the same scope as the name of the module in the `define-module` form. The `with-implicit` macro is implemented in terms of `datum->syntax-object` (see Sections 2.2.4, 2.3.4, and 2.4.1) as shown below. The reader syntax `#'`⟨form⟩ expands into `(syntax` ⟨form⟩`)`.

```
(define-syntax with-implicit
  (lambda (x)
    (syntax-case x ()
      ((_ (tid id ...) . body)
       (andmap identifier? #'(tid id ...))
       #'(with-syntax ((id (datum->syntax-object #'tid 'id)) ...) . body)))))
```

3.4.7 Compound Interfaces

It is sometimes convenient to combine several interfaces into a compound interface as shown below.

```
(define-interface one (a b))
(define-interface two (c d))
(define-interface both
  (compound-interface one two))
```

The separate interface abstraction defined in Section 3.4.6 can be extended to support compound interfaces by introducing a limited reflection mechanism. When expanded, the `define-interface` macro below defines a macro representing the interface as before.

```
(define-syntax define-interface
  (syntax-rules (compound-interface)
    [(_ name (compound-interface i0 i1 ...))
     (d-i-help name (i0 i1 ...) ())]
    [(_ name (export ...))
     (define-syntax name
       (lambda (x)
         (syntax-case x (expand-exports)
           [(_ n defs)
            (with-implicit (n export ...)
              #'(module n (export ...) . defs))]
           [(_ (expand-exports i-name mac))
            (with-implicit (i-name export ...)
              #'(mac i-name export ...))])))]))
```

For example, the definition `(define-interface one (a b))` expands into the following macro definition.

```
(define-syntax one
  (lambda (x)
    (syntax-case x (expand-exports)
      [(_ n defs)
       (with-implicit (n a b)
```

```
      #'(module n (a b) . defs))]
    [(_ (expand-exports i-name mac))
     (with-implicit (i-name a b)
       #'(mac i-name a b))])))
```

The second clause of this macro provides a simple mechanism for reflecting on the interface. When invoked with the `expand-exports` auxiliary keyword, the name of the interface `i-name`, and the name of a macro `mac`, this interface macro now expands into the macro call `(mac i-name a b)`. By analogy with continuation-passing style (CPS), the macro `mac` supplied as an argument to the interface macro is a continuation that takes as input the set of exports in the interface.

To construct a compound interface, `define-interface` calls upon the `d-i-help` macro, defined below, to collect the exports of the constituent interfaces.[3]

```
(define-syntax d-i-help
  (syntax-rules ()
    [(_ name () (export ...))
     (define-interface name (export ...))]
    [(_ name (i0 i1 ...) (e ...))
     (begin
       (define-syntax tmp
         (syntax-rules ()
           [(_ name expt (... ...))
            (d-i-help name (i1 ...)
              (e ... expt (... ...)))]))
       (i0 (expand-exports name tmp)))]))
```

The `d-i-help` macro takes the interface name, a list of constituent interfaces, and a list of exports collected so far. When it has finished processing all constituent interfaces, `d-i-help` simply calls `define-interface` with the list of exports collected. In the recursive case, `d-i-help` uses the reflective facility of the first constituent interface, `i0`. The continuation macro, `tmp`, extends the list of exports collected so far and invokes `d-i-help` recursively to process the remaining interfaces.

It is not always convenient to define a compound interface explicitly. The following version of `define-module` allows a compound interface to be specified directly.

```
(define-syntax define-module
  (syntax-rules (compound-interface)
    [(_ name (compound-interface i ...) defn ...)
     (begin
       (define-interface tmp
         (compound-interface i ...))
       (define-module name tmp defn ...))]
    [(_ name interface defn ...)
     (interface name (defn ...))]))
```

[3] The pattern `(... ...)` produces a single ellipsis in the output of the macro. Thus the input pattern specification of the `tmp` macro introduced in the expansion of `d-i-help` is `(_ name expt ...)`.

```
(define-module M (compound-interface one two)
  (define a 1)
  (define b 2)
  (define c 3)
  (define d 4))
(let () (import M) (list a b c d))  ⇒  (1 2 3 4)
```

3.4.8 Satisfying Interfaces Incrementally

The abstract module facility defined below allows a module interface to be satisfied incrementally. This permits flexibility in the separation between the interface and implementation, and it supports separate compilation of mutually recursive modules.

```
(define-syntax abstract-module
  (syntax-rules ()
    ((_ name (ex ...) (mac ...) defn ...)
     (module name (ex ... mac ...)
       (declare ex) ...
       defn ...))))

(define-syntax implement
  (syntax-rules ()
    ((_ name form ...)
     (module* () (import name) form ...))))
```

The following example illustrates the use of abstract modules to define mutually recursive modules. We first define two abstract modules, `E` and `O`.

```
(abstract-module E (even?) ())

(abstract-module O (odd?) (pred)
  (define-syntax pred
    (syntax-rules () ((_ exp) (- exp 1)))))
```

We then define implementations of these modules, each of which imports from the other.

```
(implement E
  (import O)
  (satisfy even?
    (lambda (x)
      (or (zero? x) (odd? (pred x))))))

(implement O
  (import E)
  (satisfy odd?
    (lambda (x) (not (even? x)))))
```

`declare` and `satisfy` may simply be `define` and `set!`, although a single-assignment semantics may be more appropriate for `satisfy`. If the core language does not provide a single-assignment construct, a similar effect can be obtained via macros. For example, the following module provides a `satisfy` form that issues a warning if an attempt is made to install a binding more than once.

```
(module (declare satisfy)
  (define secret "secret")
  (define-syntax declare (syntax-rules () ((_ x) (define x secret))))
  (define-syntax satisfy
    (syntax-rules ()
      ((_ x v)
       (begin
         (unless (eq? x secret)
           (warning #f "duplicate satisfy for ~s" 'x))
         (set! x v))))))
```

The `declare` macro binds a given variable to the unique constant string `"secret"`, visible only within the module. Before installing a new value, the `satisfy` macro checks whether the variable is bound to this unique constant and issues a warning if not.

The following implementation of `declare` and `satisfy` exploits identifier macros to detect both duplicate installation of the same variable and references to a variable that has not yet been installed.

```
(module (declare satisfy)
  (define secret "secret")
  (define-syntax declare
    (syntax-rules ()
      ((_ x)
       (module (x)
         (define tmp secret)
         (define-syntax x
           (identifier-syntax
             (id
              (begin
                (when (eq? tmp secret)
                  (error #f "attempt to reference unsatisfied variable ~s" 'x))
                tmp))
             ((set! id val)
              (begin
                (unless (eq? tmp secret) (error #f "duplicate satisfy for ~s" 'x))
                (set! tmp val)))))))))
  (define-syntax satisfy (identifier-syntax set!)))
```

In this implementation, `(declare x)` expands into a module exporting an identifier macro `x` that regulates access to the hidden variable `tmp` initially bound to the string constant `"secret"`. References and assignments to `x` are replaced with references and assignments to `tmp` guarded by run-time checks for this constant. A reference to `x` fails when the corresponding `tmp` is still bound to the constant. An assignment to `x` fails when `tmp` is not bound to the constant. Note that unlike the previous implementation of `declare` and `satisfy`, this implementation protects declared bindings from mutation even by `set!`.

3.5 Implementation

The module system is implemented by extending the `syntax-case` macro system described in Chapter 2. The module system uses the α-conversion machinery described in Section 2.4 to control the visibility of identifiers. In Chapter 2, α-conversion was performed by the function *resolve* as it processed the products of *subst* constructor applications. It is useful to think of the *subst* constructions layered upon an identifier as a substitution environment that maps symbols to labels in the expand-time store. In the implementation, the substitution environment is represented as a sequence of sub-environments called *ribs*, each of which corresponds to a single lexical contour. When searching the substitution environment, the ribs corresponding to the innermost contours are consulted first, in accordance with lexical scoping.

To support internal definitions and modules, the rib structures contained in substitution environments are constructed to permit incremental extension and an interface structure (described below) is added to the set of expand-time bindings to which the store maps.

When processing a `lambda` expression or `module` form, the expander creates an extensible rib in the substitution environment. This rib is part of the wrap pushed down on the form (and propagated to subforms) by the pattern matcher. Definitions within the `lambda` body or `module` form extend this rib with substitutions renaming the identifiers they bind. Internal definitions extend the store with expand-time bindings such as local macro transformers and module interfaces.

To determine the set of internal definitions, the system partially expands each subform in the body of the `lambda` or `module` form until it finds the first non-definition. When it encounters a macro call, the system expands the macro and processes any definitions in the resulting output. For `define`, `define-syntax`, and `module` forms, the system extends the substitution rib of the local scope with a fresh label for the defined variable, keyword, or module name. For a `define` form, the system defers expansion of the expression on the right-hand side and extends the store with a mapping from the fresh label to a expand-time structure representing the lexical variable. For `define-syntax` forms, the transformer expression is expanded and evaluated to obtain a procedure that is installed in the store as the binding for the fresh label. For `module` forms, the store is extended with a mapping from the fresh label to an interface structure listing the exports of the module. The sequencing construct, `begin`, is treated as a splicing form: its subforms are simply added in its place to the list of subforms to be processed.

The system processes `module` forms recursively to collect the set of internal definitions. Definitions inside and outside the module are identical except that the substitution rib extended by definitions inside the module is present only in the wrap pushed down on syntax objects representing expressions within the module. Because internal definitions extend a local substitution rib introduced by the module, substitutions for the defined identifiers are not visible outside the module. Bindings in the enclosing scope are visible within the module because the substitution environments inside and outside the module share a common tail.

Definitions within the module extend the store of the enclosing `lambda` or top-level `module` form. These bindings are accessible only where the substitutions introduced by definitions within the module are in effect. The `import` form makes available the substitutions introduced for identifiers exported from the module. An exported macro can introduce valid references to other bindings in the module where it is defined even when those bindings are not visible or are shadowed where the macro is used. This is possible because syntax objects inserted by the macro close over the substitution environment in effect where they are constructed. In particular, the module rib extended by internal definitions is present in the wrap of each syntax object constructed while processing the module.

When the expander encounters an `import` form, it looks up the module name in the substitution environment and then looks up the resulting label in the store to obtain the module interface. The module's interface structure contains syntax objects naming the exports. These syntax objects contain the module's substitution rib within their wraps. To import a module, the system extends the substitution rib of the enclosing lexical scope with substitutions mapping each export to its resolved name.

After processing the internal definitions of a `lambda` or `module` form, the system expands the expressions on the right-hand sides of variable definitions and expands any non-definition forms following the definitions. For `lambda` it then wraps the body in a `letrec` form binding variables defined locally. For top-level `module` forms it produces a series of top-level definitions for the renamed identifiers. To aid static analysis we improve on this expansion for top-level `module` forms in Section 3.6.2.

The expander collects the initialization expressions, if any, within a module and within any nested modules. These expressions are moved so that they are evaluated immediately after all definitions within the module and all definitions in the enclosing contour, but before any expressions of the enclosing contour. The following example shows the effect of this processing.

```
(let ()
  (define op (open-output-string))
  (module M (a)
    (define a 3)
    (module init1 () (print #\a))
    (define b 2)
```

```
    (module init2 () (print #\b))
    (define print (lambda (c) (write-char c op)))
    (set! a 1)
    (write (list a b) op))
  (write-char #\c op)
  (get-output-string op))   ⇒  "ab(1 2)c"
```

The initialization expressions of modules `init1` and `init2` are evaluated after `print` is defined. As indicated by the sequence of characters in the output string, the initialization expressions of the various modules are evaluated after the definitions, in the order that they appear within the program text and before the expressions in the body of the enclosing `let`. In fact, the preceding code is equivalent to the following code in which subscripts are attached to identifiers renamed during the processing of the `module` form.

```
(let ()
  (define op (open-output-string))
  (define a_0 3)
  (define b_1 2)
  (define print_2 (lambda (c) (write-char c op)))
  (print_2 #\a)
  (print_2 #\b)
  (set! a_0 1)
  (write (list a_0 b_1) op)
  (write-char #\c op)
  (get-output-string op))
```

3.6 Practical Considerations

This section discusses extensions to the language and translation mechanism to support separate compilation, static program analysis, efficient import of top-level modules, and isolated scopes.

3.6.1 Separate Compilation

The implementation described in Section 3.5 supports both internal and top-level modules. For internal modules, the new names generated by the expander must be locally unique, *i.e.*, not otherwise visible within the same top-level expression. For top-level modules within a single compilation unit, the names must be unique within the compilation unit. When multiple compilation units may be linked together, the names must be unique across compilation units.

Our system supports both incremental compilation and linking of compilation units compiled by different runs of the compiler. Since the compiler may be invoked simultaneously on the same or different physical hardware, global name generation factors in the host's network interface identifier, process identifier (PID), and compiler invocation time. Some operating systems provide a system call that generates unique names in much the same fashion.

If a module `m2` imports from a module `m1`, the compiler must have the interface for `m1` in order to compile `m2`. When a source file containing `m1` is compiled, the resulting object file contains the compiled code for `m1`, its interface, and compiled code for any macro transformers defined by `m1`. Before compiling `m2`, it is necessary to first `load` or `visit` `m1`'s object file to install its interface and transformers. While both `load` and `visit` install the interface and transformers, `load` also loads the compiled code and evaluates the initialization expressions.

3.6.2 Static Program Analysis

In the presence of separate compilation, global variables are subject to arbitrary manipulation by compilation units that may be unavailable for scrutiny by the compiler. One benefit of modular program structure is that it may improve the results of static analysis. Variables not exported from a module are effectively local and may therefore present opportunities for analysis and optimization. Moreover, when a module imports from the `scheme` module, the compiler may determine that an identifier such as `cons` refers to the corresponding Scheme primitive.[4] Recognizing primitives improves the quality of analysis and optimization.

The ability to export macros complicates static analysis, however, because an exported macro may expand into references or assignments to nonexported identifiers. It is tempting to believe that a straightforward analysis of the transformer expression can determine the set of these *implicit* exports. Even with the constrained `syntax-rules` transformers, however, such an analysis is necessarily conservative. For example, consider the following definition for module `M`.

```
(module M (a b)
  (import scheme)
  (define-syntax a
    (syntax-rules ()
      [(_ e0 e1 ...) (e0 (quote c) e1 ...)]))
  (define b (lambda () c))
  (define c 3))
```

It appears that `c` is not referenced outside the definition of `b`. The only other occurrence of `c` is within the macro `a`, where it is apparently used as symbolic data. When processing the transformer expression associated with `a`, the hypothetical analysis can easily determine that `quote` refers not to some malicious macro that will assign or reference `c`, but to the familiar and quite harmless `quote`

[4]Lacking a module system, Scheme compilers often provide switches that allow programmers to declare wholesale that, if not bound locally, all such identifiers in fact refer to system primitives.

form imported from the `scheme` module. The hypothetical analysis would conclude that (1) `c` is unassigned and (2) `c` is referenced only within procedure `b`.

As the following program demonstrates, however, code outside the module may not only reference but also assign `c`.

```
(let ()
  (define-syntax f
    (syntax-rules ()
      [(_ (any id)) id]
      [(_ (any id) value) (set! id value)]))
  (import M)
  (let ([original (a f)])
    (a f 4)
    (list original (b))))  ⇒  (3 4)
```

Here `(a f)` expands to `(f (quote c))`, which further expands to the reference `c`. Similarly, `(a f 4)` expands to `(f (quote c) 4)`, which further expands to the assignment `(set! c 4)`. Therefore, because the compiler does not know how the macro `a` will be used in separately compiled code, it must conservatively assume that the variable `c` is assigned and that its value escapes.

Even conservative analysis is impossible if more general transformers are available that can synthesize new identifiers from existing identifiers, *e.g.*, using `datum->syntax-object` (see Section 2.3.4). For example, consider the following module definition.

```
(module foo (access)
  (define a 123)
  (define-syntax b (identifier-syntax 456))
  (define-syntax access
    (lambda (x)
      (syntax-case x ()
        [(_ exp)
         (datum->syntax-object #'b
           (syntax-object->datum #'exp))]))))
```

The `access` macro returns its argument expression intact but for one key difference: the wrap on the input is replaced with the wrap from the identifier `b` bound inside the module.[5] Consequently, expanding `(access exp)` has the effect of expanding `exp` within the module scope. For example, `(access b)` constructs a reference to the private syntax binding `b` and `(access (set! a 3))` constructs an assignment to the private variable binding `a`.

In fact, the `access` macro need not be defined within the module that is violated. A module boundary can be breached if any syntax object constructed within the module may arrive at a call to the procedure `datum->syntax-object`. For instance, the `reveal` macro below is virtually identical

[5]The choice of `b` here is arbitrary. Each syntax object constructed within the module carries in its wrap sufficient information to access any module binding.

to the `access` macro defined earlier except that it transplants the wrap from an identifier in its input onto `exp`.

```
(define-syntax reveal
  (lambda (x)
    (syntax-case x ()
      ((_ (any n ...) exp)
       (datum->syntax-object #'any
         (syntax-object->datum #'exp))))))
```

Now consider the module defined below and observe that the macro `a` contains no occurrences of the identifier `b`. In fact, `b` is not even referenced within the module.

```
(module foo (a)
  (import scheme)
  (define-syntax a
    (syntax-rules ()
      [(_ e0 e1 ...)
       (e0 (quote 3) e1 ...)]))
  (define b 1))
```

The `reveal` macro provides arbitrary access to bindings within the `foo` module, including `b`, as illustrated below.

```
(let ()
  (import foo)
  (a reveal
    (let ((old b))
      (set! b 5)
      (list old b))))  ⇒  (1 5)
```

One partial solution to the problem of implicit exports is to disallow assignments outside the module to unexported variables. Unfortunately this introduces a difference in the semantics of macros inside and outside modules: outside the module a macro cannot expand into an assignment to an unexported variable, while inside it can. In particular, the programmer must remember which macros can and cannot be used outside the module (directly or as implicit exports). This limits the expressive power of exported macros and does not address the problem of escaping values.

Another partial solution is to introduce a `define-constant` form that establishes immutable bindings. This does not inherently limit the expressive power of exported macros but leaves the problem of escaping values.

We have chosen instead to require that implicit exports be explicitly declared in the interface for top-level modules[6] using the following syntax.

⟨interface⟩ ⟶ `(`⟨export⟩`*)`
⟨export⟩ ⟶ ⟨identifier⟩ | `(`⟨identifier⟩ ⟨export⟩`*)`

[6]The declarations are permitted but not required for internal modules.

Declarations of implicit exports are propagated to enclosing interfaces where an identifier is exported. For example, the module `m` defined below exports only `a` and `c`, but the export of `a` causes the implicit export of `b` and `c` from module `n`.

```
(module m ((a b) c)
  (define-syntax a (identifier-syntax b))
  (module n ((b c) (d e))
    (define-syntax b (identifier-syntax c))
    (define c (lambda (x) (d x)))
    (define-syntax d (identifier-syntax e))
    (define e (lambda (x) (* x 2))))
  (import n)
  (define c (lambda (x) (d (+ x 1)))))
```

Because `e` is not exported to the top level, explicitly or implicitly, it is bound locally in the output of the expander. Thus the preceding program is equivalent to the following.[7]

```
(begin
  (cte-install! m_8 'interface
    '#(interface #<import-token> (#<syntax-object a_0> #<syntax-object c_6>)))
  (cte-install! a_0 'transformer (identifier-syntax b_1))
  (cte-install! b_1 'transformer (identifier-syntax c_2))
  (letrec ([e_3 (lambda (x_4) (* x_4 2))])
    (set! c_2 (lambda (x_5) (e_3 x_5)))
    (set! c_6 (lambda (x_7) (e_3 (+ x_7 1))))))
```

Because `e` is not assigned, it can be integrated by a later pass of the compiler.

As described in Section 3.5, the expander processes a module in two stages. First, it partially expands each form in order to determine, and introduce substitutions for, the set of defined identifiers. During this process the expander also records the set of implicit exports declared for each exported identifier. The expander uses this information to partition the definitions into a set of local variable bindings, a set of (explicitly or implicitly) exported variable bindings, and a set of (explicitly or implicitly) exported macro and module bindings.

The partitioning process essentially computes the transitive closure of the implicit export relation, E_i, where $(\mathtt{x}, \mathtt{y}) \in E_i$ if `y` is listed among the implicit exports of `x`, and $(\mathtt{z}, \mathtt{z}) \in E_i$ if `z` is exported. The partitioning process begins with a worklist containing all explicit and implicit exports declared in the interface of the top-level module and a list of the variable, macro, and module definitions encountered within the top-level module. An identifier is removed from the worklist by moving the corresponding definition from the incoming definition list to the appropriate list of exported variable or macro and module definitions, then adding to the worklist any implicit exports recorded earlier for that definition. If a module is exported, the identifiers in the interface of that module are

[7] `(cte-install!` *sym type exp*`)` establishes a compile time binding with the specified *type* that associates *sym* with the value obtained by evaluating *exp*. In essence, this extends the expand-time store, r_{init}, described in Section 2.4.

also added to the worklist. When the worklist is empty, the incoming definition list contains only the unexported local bindings and the other lists contain all of the implicitly or explicitly exported bindings.

After partitioning the bindings and generating unique names for all exported bindings as described in Section 3.6.1, the right-hand sides of variable definitions are expanded along with any initialization expressions of the module. The expander then constructs a `letrec` binding for the unexported local variables. The `letrec` body is a sequence of global assignments to the unique names for the implicitly or explicitly exported variables followed by any initialization expressions.

Automatically propagating the list of implicit exports associated with a macro when it is exported ensures that macros and modules can be used freely in defining new syntactic abstractions without imposing restrictions on the use of these new abstractions at top level. For example, the macro defined below expands into an anonymous module that exports a macro with implicit export `tmp`.

```
(define-syntax define-parameter
  (syntax-rules ()
    [(_ name init-value)
     (module* ((name tmp))
       (define tmp init-value)
       (define-syntax name
         (syntax-rules ()
           [(_) tmp]
           [(_ v) (set! tmp v)])))]))
```

Because implicit exports are propagated, a parameter defined via this macro can be exported without explicitly exporting `tmp`. This is important for two reasons. First, `tmp` is part of the parameter implementation. If a programmer using the `define-parameter` macro had to be aware of `tmp`, the benefit of the abstraction would be lost. Second, abstractions can be used repeatedly. For example, the following module defines and exports two parameters.

```
(module format-text (left-margin right-margin fmt)
  (define-parameter left-margin 5)
  (define-parameter right-margin 75)
  (define fmt (lambda () ...)))
```

Each of these parameters is implemented with a variable named `tmp`. Because the identifier `tmp` is introduced during the expansion of the `define-parameter` macro, the two `tmp` variables will of course be uniquely renamed. If implicit exports were not propagated automatically, then it would be necessary to manually annotate the source program with this information. Moreover, the implementation of the `format-text` module might involve other parameters or other variables named `tmp`, none of which is intended for export. At the source level, however, there is no way to indicate that some of the identifiers named `tmp` are to be (implicitly or explicitly) exported while others are not.

3.6.3 Efficient Import of Large Modules

Rapid compilation can boost programmer productivity. To that end it is important that using modules in a program does not noticeably increase compile time. Because modules support programming in the large, it is particularly important that compile time scale gracefully with program size. Most of the additional overhead in the implementation of the module system consists of applying additional substitutions. During macro expansion, these substitutions must be applied to each identifier in a program.

Excluding `import`, each of the definition forms introduces a single substitution. So the number of substitutions to be applied when resolving an identifier is the same as the number of such definitions in the program. An `import` form, however, may introduce literally hundreds of substitutions. For example, the oft-imported `scheme` module exports nearly 600 bindings. If all of these substitutions were added to the local substitution rib wherever the `scheme` module (or any such large module) was imported, compile time would noticeably increase.

To support efficient import of large top-level modules, the substitution mechanism is extended to handle a new substitution type, the *import token*. When a top-level module is expanded, an import token (a uniquely generated symbol) is associated with the resulting module interface. When the module definition is loaded or evaluated, the import token is associated with a hash table that maps the symbolic names of explicitly or implicitly exported module identifiers to their generated names. The hash table implements the substitution mapping for that module. When the module is imported, only the import token is added to the local substitution rib. If an import token is encountered while searching the rib for an applicable substitution, the corresponding hash table is consulted. If a substitution is found in the hash table, and the set of marks on the exported identifier matches the set of marks at this point in the substitution rib, the substitution is applied. With this mechanism in place, importing from a large module has no noticeable effect on compile time.

Another potential source of inefficiency is in the test for duplicate identifier definitions within a module. The expander collects a list of the identifiers defined and the module interfaces imported within a module. To signal an error when an identifier is multiply defined, the expander must check each element of this list against all the others. There is no need to check for conflicts within individual elements of the list: a lone identifier conflicts with nothing and a module interface cannot contain duplicate identifiers because an error would have been signalled when the module was defined. When checking whether a given identifier is already defined in an imported interface, the expander searches for that identifier in the hash table associated with the interface. When checking for overlap between two interfaces, the expander searches for each identifier in the shorter interface using the hash table associated with the larger interface. This ensures that the expander never iterates through the list of identifiers in the largest interface.

3.6.4 Isolated Scopes

Expressions within a module can reference identifiers bound outside of the module. For example, the following expression evaluates to 8.

```
(let ((x 3))
  (module m (plusx)
    (define plusx (lambda (y) (+ x y))))
  (import m)
  (let ((x 4)) (plusx 5)))
```

The system supports a variant of `import`, called `import-only`, that creates an *isolated scope*, in which the only visible identifiers are those exported by the imported module. This is useful for static verification that a module does not access identifiers that are not explicitly provided by the import. Isolation of scope is, however, independent of the `module` form. For example, the following expression generates an expansion-time error.

```
(let ((x 3))
  (module m ())
  (import-only m)
  x)
```

By using `import-only` to implement the qualified reference macro of Section 3.4, we ensure that it introduces a reference to the intended export and not to some local binding.

```
(define-syntax from
  (syntax-rules ()
    ((_ M id) (let () (import-only M) id))))
```

When a module is imported via `import-only`, subsequent imports are not possible unless this module exports the name of at least one module and `import` or `import-only`. To create an isolated scope containing the exports of more than one module without making `import` or `import-only` visible, it is necessary to create a single module that contains the exports of each of the other modules.

```
(module m2 (y) (define y 'y))
(module m1 (x) (define x 'x))
(module compound-module (x y cons)
  (import m1)
  (import m2)
  (import scheme))
(let ((x 3))
  (import-only compound-module)
  (cons x y))                      ⇒  (x . y)
```

Isolated scopes are implemented by extending the import-token mechanism described in Section 3.6.3 with a distinguished `fail` token. When a module is imported via `import` or `import-only`, the expander makes the exported bindings of the module visible in the same way by extending the local substitution environment. For `import-only` the expander inserts a `fail` token just after the newly added substitutions. The `fail` token is keyed with the set of marks that apply to the `import-only` identifier. If the `fail` token is encountered when searching for an applicable substitution and the set of current marks is identical to the set of marks associated with the `fail` token, the substitution process is aborted and a compile time error is signalled. If the marks do not match, the substitution process is unaffected. This ensures that an `import-only` form introduced in the expansion of a macro affects only the identifiers introduced in that expansion.

3.6.5 Interface Reflection

When importing a subset of the identifiers exported by a module it is often convenient to specify only the set of identifiers to be excluded. Similarly, when renaming imported identifiers it is often convenient to specify a prefix to be used in renaming all identifiers. Providing a limited form of reflection on the module interface allows these and other forms of import to be expressed as macros.

We assume the syntax `(expand-with-exports macro module-name)` invokes the given macro with the exports of the named module as its arguments. In Section 3.4.7 we saw how to derive a similar mechanism for interface reflection at the source level via syntactic abstraction. Using the `expand-with-exports` facility, we can implement prefix import as follows.

```
(define-syntax import-prefixed
  (lambda (x)
    (syntax-case x ()
      [(_ mname prefix)
       (with-syntax ([dots #'(... ...)])
         #'(begin
             (define-syntax do-prefix
               (lambda (x)
                 (syntax-case x ()
                   [(key id dots)
                    (with-syntax ([(newid dots)
                                   (map (lambda (x) (compound-id x prefix x))
                                        #'(id dots))])
                      #'(module* (newid dots)
                          (import mname)
                          (define-alias newid id)
                          dots))])))
             (expand-with-exports do-prefix mname)))])))
```

When expanded, `import-prefixed` defines a `do-prefix` macro and invokes the `expand-with-exports` macro with `do-prefix` as the "continuation." The identifier `do-import` is visible only in the output of the `import-prefixed` macro. The `do-import` macro is invoked with the identifiers exported from the named module as arguments. This macro generates a prefixed identifier for each export using the `compound-id` procedure, defined below, and generates an anonymous module exporting the new identifiers. The named module is imported within the anonymous module where syntactic aliases are defined to translate each occurrence of the new prefixed identifiers into the corresponding export of the named module. The aliases are then exported from the anonymous module.

```
(define compound-id
  (lambda (template . rest)
    (datum->syntax-object template
      (string->symbol
        (apply string-append
          (map (rec cvt
                 (lambda (x)
                   (cond
                     [(string? x) x]
                     [(symbol? x) (symbol->string x)]
                     [else (cvt (syntax-object->datum x))])))
               rest))))))
```

To implement exclusive import, we use `expand-with-exports` to obtain a list of the identifiers exported by a given module, remove any identifiers to be excluded, and construct an anonymous module with the pruned list of identifiers as its interface. The anonymous module simply imports from the named module. The implementation is given below.

```
(define-syntax import-excluding
  (lambda (x)
    (syntax-case x ()
      [(_ mname excl-id ...)
       (with-syntax ([dots #'(... ...)])
         #'(begin
             (define-syntax datum
               (syntax-rules () [(_ x) (syntax-object->datum #'x)]))
             (define-syntax do-prefix
               (lambda (x)
                 (syntax-case x ()
                   [(key id dots)
                    (with-syntax ([iface
                                   (let ([excluded (datum (excl-id ...))])
                                     (let rem ([ids #'(id dots)] [keep '()])
                                       (syntax-case ids ()
                                         [() keep]
                                         [(x more dots)
                                          (memq (datum x) excluded)
                                          (rem #'(more dots) keep)]
                                         [(x more dots)
```

```
                                      (rem #'(more dots) (cons #'x keep))]))))])
                         #'(module* iface (import mname)))])))
                (expand-with-exports do-prefix mname)))])))
```

3.7 Extended Examples

This section demonstrates the expressive power of the system by implementing extensions that support dynamic linking and object-oriented programming.

3.7.1 Dynamic Linking

Flatt and Felleisen propose a system for dynamically linking separately compiled program *units* [31]. As shown in Figure 9, the unit extension can be implemented entirely at the source level using identifier macros and features of the module system described in this chapter.

Background

Units are first-class values supporting just two operations: linking and invocation. A unit is created by a `unit` form with the following syntax.

```
(unit (import variable ...) (export variable ...)
  definition ...
  expression ...)
```

The `import` and `export` clauses list the variables imported and exported by the unit. The body of a unit consists of a set of zero or more definitions followed by zero or more initialization expressions.

A new unit can be obtained by linking two or more constituent units using a `compound-unit` form with the following syntax.

```
(compound-unit
  (import variable ...)
  (link [tag₁ (expr₁ linkspec₁ ...)]
        ...
        [tagₙ (exprₙ linkspecₙ ...)])
  (export (tag variable) ...))
```

The `import` clause lists the variables imported by the compound unit. Each entry in the `link` clause associates a unique $\texttt{tag}_i$ with the unit obtained by evaluating the corresponding $\texttt{expr}_i$. The list of `linkspec`s following each `expr` specifies where the corresponding import of that unit is to be found. If `linkspec` is a variable, it refers to an import listed in the `import` clause. Otherwise `linkspec` is of the form `(tag variable)`, indicating a link to the `variable` exported by the unit associated with the given `tag`. The `export` clause re-exports each listed `variable` exported by the constituent unit with the corresponding `tag`.

A unit is invoked via an `invoke-unit` form with the following syntax.

(invoke-unit expr x_0 ... x_n)

If present, the optional variables x_0 ... x_n are exported to the unit obtained by evaluating `expr`. The invoked unit must import as many variables as are exported.

Implementing unit

The `unit` macro expands into a `lambda` expression that creates explicit locations for the exported variables when invoked. The body of the `lambda` expression is a call to the `make-unit` macro which returns a procedure that takes a continuation expecting two values: a finite function mapping export names to export locations and a procedure parameterized over the imported locations. The import procedure evaluates the definitions within the unit in a syntactic environment in which the import variable names are bound to identifier macros that either dereference or mutate the contents of the locations passed in as imports. For example, here is a simple `unit` expression

```
(unit (import a b) (export x y)
  (define x (lambda () (+ a y)))
  (define y 5)
  (display "initialized")
  (newline))
```

followed by its expansion.

```
(lambda ()
  (let ([g1 'unbound] [g2 'unbound])
    (let ([g3 (locative g1)] [g4 (locative g2)])
      (lambda (k)
        (k (lambda (name)
             (case name
               [(x) g3]
               [(y) g4]
               [else (error #f "~s is not exported" name)]))
           (lambda (a b)
             (letrec ([x (lambda () (+ ((a)) y))] [y 5])
               (g3 (locative x))
               (g4 (locative y))
               (display "initialized")
               (newline)
               (void)))))))))
```

Locations for the exported variables `x` and `y` are allocated during the evaluation of the `letrec` form when the import procedure `(lambda (a b) ...)` is invoked. To permit mutually recursive linking, indirect locations `g1` and `g2` are created outside the unit access procedure `(lambda (k) ...)`. Variables `g3` and `g4` are bound to locative procedures (described below) that are returned by the finite function `(lambda (name) ...)` implementing the export variable lookup table. Locatives providing read and write access to variables `x` and `y` are installed by the calls `(g3 (locative x))` and `(g4 (locative y))`.

A locative is a variable arity procedure that closes over a lexical variable. When invoked with no arguments, the procedure returns the value of that variable. When invoked with a single argument, the procedure assigns the argument value to that variable. The `locative` macro is defined below.

```
(define-syntax locative
  (syntax-rules ()
    [(_ var)
     (case-lambda
       [() var]
       [(val) (set! var val)])]))
```

Since the locative closes over a lexical variable, ordinary references and assignments to that variable are semantically consistent with zero- and one-argument calls to the locative no matter where it is invoked.

The `unit` implementation uses locatives to represent imported and exported variables so that assignments to an exported variable will be reflected within a unit importing that variable and similarly so that assignments to an imported variable will be reflected within the unit that exported the variable. This representation has the added advantage that exported variables may be bound not only by `define`, but also via derived definitions, such as `define-structure`.

An extra level of indirection is introduced to support mutually recursive linking. Units are linked by invoking their import procedures with locatives for the imported variables as arguments. The locative for an exported variable `x` is not created until the import procedure of the defining unit is invoked (since the exported variables are not bound until that time). When linking mutually recursive units, however, this is too late. Say we want to link two units, `A` and `B`. To invoke the import procedure of unit `A`, we first need the locations created when the import procedure of unit `B` is invoked, but to invoke unit `B`, we first need the locations created when `A` is invoked. An extra level of indirection makes one set of locations available to both units at link time. When the unit import procedures are finally invoked, the export locatives are installed in the link locations.

The indirection is transparent to the user: imported variables are accessed via the usual syntax for variable reference and assignment. In fact, each import variable `iv ...` is bound to an indirect locative when the `lambda` expression representing the import procedure (Figure 10, line 14) is invoked.

```
(module Unit ((unit make-unit) (compound-unit make-unit) invoke-unit)
  (import Unit-help)
  (define-syntax unit
    (lambda (x)
      (syntax-case x (import export)
        [(_ (import iv ...) (export ev ...) form ...)
         (with-temporaries ([(tmp ...) (ev ...)]
                            [(ev-loc ...) (ev ...)]
                            [(iv-loc ...) (iv ...)])
           #'(lambda ()
               (let ([tmp 'unbound] ...)
                 (let ([ev-loc (locative tmp)] ...)
                   (make-unit ((ev ev-loc) ...) (iv ...)
                     (let-syntax ([iv-loc (identifier-syntax iv)] ...
                                  [iv (locative-identifier-syntax (iv))] ...)
                       (let ()
                         (define-syntax ensure-locative
                           (syntax-rules (iv ...)
                             [(_ iv) (iv-loc)] ...
                             [(_ id) (locative id)]))
                         (module () (ev-loc (ensure-locative ev)) ...)
                         form ... (void))))))))])))
  (define-syntax compound-unit
    (lambda (x)
      (syntax-case x (import link export)
        [(_ (import iv ...)
            (link (ltag (expr linkage ...)) ...)
            (export (etag ev) ...))
         (with-temporaries ([(ulocs ...) (ltag ...)] [(uimp ...) (ltag ...)])
           #'(lambda ()
               (mvlet* ([(ulocs uimp) (expr)] ...)
                 (define-syntax ltag
                   (syntax-rules () [(_ id) (ulocs 'id)])) ...
                 (make-unit ((ev (etag ev)) ...) (iv ...)
                   (define-syntax ensure-locative
                     (lambda (x)
                       (syntax-case x (iv ... ltag ...)
                         [(_ iv) #'iv] ...
                         [(_ (ltag id)) #'(ltag id)] ...
                         [(_ id) (identifier? #'id) #'(wrap (locative id))]
                         [(_ exp) #'(let ([t exp]) (wrap (locative t)))])))
                   (uimp (ensure-locative linkage) ...) ...))))])))
  (define-syntax invoke-unit
    (syntax-rules ()
      [(_ expr var ...)
       (mvlet* ([(ulocs uimp) (expr)])
         (uimp (wrap (locative var)) ...))])))
```

Figure 9: An implementation of the $\textsc{Unit}_d$ module facility of Flatt and Felleisen.

```
(module Unit-help (wrap make-unit with-temporaries mvlet*
                   locative locative-identifier-syntax)
  (define-syntax wrap
    (syntax-rules ()
      [(_ exp) (let ((p exp)) (lambda () p))]))
  (define-syntax make-unit
    (syntax-rules ()
      [(_ ((ev ev-loc) ...) (iv ...) form ...)
       (lambda (k)
         (k (lambda (name)
              (case name
                [(ev) ev-loc ] ...
                [else (error #f "~s is not exported" name)]))
            (lambda (iv ...) form ...)))]))
  (define-syntax with-temporaries
    (syntax-rules ()
      [(_ ([tmps ls] ...) e0 e1 ...)
       (with-syntax ([tmps (generate-temporaries #'ls)] ...) e0 e1 ...)]))
  (define-syntax mvlet*
    (syntax-rules ()
      [(_ () body1 body2 ...) (let () body1 body2 ...)]
      [(_ ((fmls1 expr1) (fmls2 expr2) ...) body1 body2 ...)
       (expr1
         (lambda fmls1 (mvlet* ((fmls2 expr2) ...) body1 body2 ...)))]))
  (define-syntax locative
    (syntax-rules ()
      [(_ var)
       (case-lambda
         [() var]
         [(val) (set! var val)])]))
  (define-syntax locative-identifier-syntax
    (syntax-rules ()
      [(_ loc)
       (identifier-syntax
         (_ (loc))
         ((set! _ val) (loc val)))])))
```

Figure 10: Auxiliary definitions used in the implementation of $\textsc{Unit}_d$.

The `let-syntax` on line 14 of Figure 9 shadows each `iv` with a binding for an identifier macro that performs the necessary indirection on the outer `iv`. Because the `let-syntax` binds the same identifiers present in the input to the `unit` macro, these bindings capture existing references to the imported variables in the body of the unit. The `let-syntax` form also binds a fresh identifier `iv-loc` to an identifier macro that expands into a reference to the corresponding `iv` bound by the `lambda`. An intermediate stage in the expansion of the simple unit expression on page 85 is shown below.

```
(lambda ()
  (let ([g1 'unbound] [g2 'unbound])
    (let ([g3 (locative g1)] [g4 (locative g2)])
      (lambda (k)
        (k (lambda (name)
             (case name
               [(x) g3]
               [(y) g4]
               [else (error #f "~s is not exported" name)]))
           (lambda (a b)
             (let-syntax ([g5 (identifier-syntax a)]
                          [g6 (identifier-syntax b)]
                          [a (locative-identifier-syntax (a))]
                          [b (locative-identifier-syntax (a))])
               (let ()
                 (define-syntax ensure-locative
                   (syntax-rules (a b)
                     ((_ a) (g5))
                     ((_ b) (g6))
                     ((_ id) (locative id))))
                 (module () (g3 (ensure-locative x)) (g4 (ensure-locative y)))
                 (define x (lambda () (+ a y)))
                 (define y 5)
                 (display "initialized")
                 (newline))))))))
```

The double indirect imposes additional overhead on each reference or assignment to an imported variable.[8] Imported variables are referenced and assigned by first invoking the link location to obtain the locative for the exported variable and then applying that locative to the appropriate argument. For example, note the expansion of the reference to import variable `a` in the definition of `x` from page 85.

```
(define x (lambda () (+ a y)))  →  (define x (lambda () (+ ((a)) y)))
```

Locally defined variables exported from a unit, such as `y` in this example, are referenced and assigned directly (not through locatives) within that unit.

[8]In private communication, Felleisen indicates that imported variables are intended to be immutable. We chose to implement a more general form in order to dispel any concerns about the power of our approach. Implementing the restriction presents no difficulty, may eliminate one level of indirection, and may present software engineering benefits as well.

An anonymous module containing only initialization expressions is used in line 21 of Figure 9 to install the exported values in the link locations as described above. The expander automatically moves the initialization expressions of any modules within a body so that they are evaluated immediately following the definitions in that body. In the `unit` implementation, this ensures that the code produced by line 21 is evaluated after any definitions (but before any expressions) of the `unit` body in line 22. In the sample expansion on page 85, for example, the calls `(g3 (locative x))` and `(g4 (locative y))` were inserted just after `x` and `y` were bound but before the call to `display`.

Locatives for exported variables are introduced selectively by the `ensure-locative` macro on line 21 of Figure 9. This macro uses the auxiliary keywords list of the `syntax-rules` form to recognize and avoid creating locatives for imported variables that are re-exported. Instead, the corresponding import variable is invoked (following a single indirect) to obtain the actual locative encapsulating the (actual) import variable. For an export variable defined locally within the unit, a fresh locative is created by the expansion of line 20. In either case, the resulting locatives are installed in the link locations by the calls generated at line 21.

Implementing compound-unit

In line 31 of Figure 9, the `compound-unit` macro invokes the access procedure (returned by `make-unit`, in line 9 of Figure 10) of each constituent unit `expr` and binds the return values to `ulocs` and `uimp`. Each of the `ulocs` is bound to a finite function representing the export variable lookup table of the corresponding unit. Each `uimp` is bound to the import procedure of the corresponding unit. A local `ltag` macro is defined for each link tag. An `ltag` macro expands into code that invokes the appropriate `ulocs` procedure to obtain the desired export. The `make-unit` macro is used on line 34 to construct a unit access procedure that returns two values: a finite function mapping export names to export locations and an import procedure that invokes the import procedures of the constituent units with the specified import locations. For example, here is a simple `compound-unit` expression

```
(compound-unit
  (import default-error)
  (link (PB (phone-book default-error))
        (GUI (gui (PB insert!) (PB lookup)))
        (MAIN (main (PB insert!) (GUI make-window))))
  (export (PB lookup)))
```

followed by its expansion.

```
(lambda ()
  (mvlet* ([(g1 g2) (phone-book)]
           [(g3 g4) (gui)]
           [(g5 g6) (main)])
```

```
(lambda (k)
  (k (lambda (name)
       (case name
         [(lookup) (g1 'lookup)]
         [else (error #f "~s is not exported" name)]))
     (lambda (default-error)
       (g2 default-error)
       (g4 (g1 'insert!) (g1 'lookup))
       (g6 (g1 'insert!) (g3 'make-window))
       (void)))))
```

Note in the call to `g6` (the import procedure returned by `main`) that `insert!` is obtained from the `ulocs` function `g1` returned by the call to `phone-book`, as directed by the link specification `(PB insert!)`, while `make-window` is obtained from the `ulocs` function `g3` returned by the call to `gui`, as directed by the link specification `(GUI make-window)`.[9] To improve readability, the `mvlet*` macro is not expanded in this example.

The `ensure-locative` macro defined in lines 35–41 of Figure 9 is used in line 42 to ensure that indirect locatives are introduced for any `linkage` that is not among the import variables (`iv ...`) and is not a link tag expression (`(ltag id) ...`). Locatives are not required in these cases because the import variables already refer to locations, and the link tag expressions expand into calls that return export variable locations. In cases where a locative is introduced, the `wrap` macro is used to add an extra level of indirection around the locative to ensure that imported variables are uniformly represented with a double indirect as expected by the code that references and assigns import variables.

Implementing invoke-unit

The `invoke-unit` macro simply evaluates the unit expression to obtain the export variable lookup function and the import procedure. The import procedure is then invoked with locatives for any variables supplied by the `invoke-unit` expression. Once again, the `wrap` macro is used to add the extra level of indirection around the locatives so that imported variables can be accessed uniformly.

3.7.2 Object-oriented Programming

This section describes a static system for object-oriented programming implemented using the features of both the macro system and the module system.

[9] No additional code is required to handle `etags`, eg. `PB`, since each `etag` in the export clause must match one of the `ltag` macros.

Background

An object encapsulates data and a set of operations, called methods, on that data. What distinguishes object-oriented programming from the more conventional use of abstract data types is the additional provision that an object may inherit attributes of another object. Inheritance is often a useful way to structure code sharing, and this is the basis for much of the popularity that object-oriented programming enjoys.

In addition to inheritance, an object system typically provides some mechanism for late binding. Thus an operation may be invoked generically on many different kinds of objects that arrive at a particular call site. Because the dispatch is implicit, new variants of existing objects may be added without revisiting the code that manipulates these objects via their external interface.

This section describes a simple single-inheritance object system. The data members and methods supported by an object are described in a *class* definition. A new class is derived from a single base class, and it inherits the data members (instance variables) and methods of the base class. For example, a new class `<point>` is derived from the `<base>` class below.

```
(define-class (<point> x y) (<base>)
  (ivars (x x) (y y))
  [get-x (method () x)]
  [get-y (method () y)]
  [delta
   (method (pt k)
     (k (abs (- (send self get-x) (send pt get-x)))
        (abs (- (send self get-y) (send pt get-y)))))])
```

Just as a single `lambda` expression may be evaluated to obtain many distinct closures, a class may be instantiated to obtain many distinct *instances*. The `x` and `y` appearing after `<point>` declare parameters that must be supplied when this class is instantiated. The scope of these parameters includes the base class clause and the `ivars` clause. The base class clause specifies the class from which the new class is derived and supplies any actual parameters required to instantiate the base class. The `ivars` clause establishes initial bindings for the instance variables. In this example, the instance variables `x` and `y` are simply bound to the values of the instantiation parameters. Object methods are invoked via the `send` syntax. In the `delta` method, for example, the expression `(send pt get-x)` invokes the `get-x` method of the instance `pt` passed in as an argument. Within each method body, the special variable `self` is bound to the instance on which the method was invoked.

Inheritance is demonstrated by the two class definitions below.

```
(define-class (<cartesian-point> x y) (<point> x y)
  (ivars)
  [distance
   (method (pt)
     (send self delta pt
       (lambda (dx dy)
         (sqrt (+ (* dx dx) (* dy dy))))))])
(define-class (<manhattan-point> x y) (<point> x y)
  (ivars)
  [distance
   (method (pt)
     (send self delta pt +))])
```

Both classes inherit from the `<point>` class defined earlier. No new data members are required, so the instantiation parameters are simply passed to the base class. The new classes support `distance` methods defined in terms of the inherited `delta` method of `<point>`. In the following expression, both classes are instantiated via the `make` syntax, and the two different `distance` methods are invoked.

```
(let ([p1 (make <manhattan-point> 0 0)]
      [p2 (make <cartesian-point> 3 4)])
  (list (send p1 distance p2)
        (send p2 distance p1)))   ⇒  (7 5)
```

Implementation

Figure 11 shows the implementation of `define-class`. A class is represented by a like-named module that exports a predefined set of macros (`$layout`, `$maker`, and `$build-dispatch`), a module (`$methods`), a procedure (`$dispatch`), and a method (`isa?`). Each of the exported macros is defined in terms of the corresponding macro exported by the module representing the base class. The exports of the base-class module are imported with the prefix "`base.`" using the `import-prefixed` macro defined in Section 3.6.5.[10]

The `$maker` macro constructs a `lambda` expression with the instantiation parameters `fmls` as its formal parameters. It applies the `lambda` expression constructed by the `$maker` of the base class to the actual parameters designated for the base class. The values (`val ...`) of the instance variables are prepended to the list of instance variable values passed to `base.$maker`. This macro is used in the definition of the `make` macro shown below.

[10]For simplicity, we assume `expand-with-exports`, used in the implementation of `import-prefixed`, is a core transformer. While this is not strictly necessary, it allows us to use the familiar `module` and `module*` forms in the implementation of the object system. In fact, the issue could be avoided entirely using the `import*` macro of Section 3.4.3, at the expense of a few additional lines of code.

```
(define-syntax define-class
  (lambda (x)
    (syntax-case x ()
      [(dc (class . fmls) (base actual ...) (ivars (id val) ...) (mname meth) ...)
       (with-implicit
         (class $build-dispatch $dispatch $layout $maker $methods isa? send-base
                base.$maker base.$layout base.$build-dispatch)
         (with-syntax ([dots #'(... ...)])
           #'(module class (($maker base.$maker) ($layout base.$layout)
                            ($build-dispatch base.$build-dispatch mname ...)
                            $methods $dispatch isa?)
               (import-prefixed base "base.")
               (define-syntax $layout
                 (syntax-rules ()
                   [(_ (x dots) body) (base.$layout (id ... x dots) body)]))
               (define-syntax $maker
                 (syntax-rules ()
                   [(_ dispatch (x dots))
                    (lambda fmls
                      ((base.$maker dispatch (val ... x dots)) actual ...))]))
               (module $methods (mname ...)
                 (define-syntax send-base
                   (lambda (x)
                     (syntax-case x ()
                       [(_ meth-name arg dots)
                        (with-implicit (_ base $methods self)
                          #'(let ()
                              (import base)
                              ((from $methods meth-name) self arg dots)))])))
                 (define mname ($layout () meth))
                 ...)
               (define isa?
                 (method (dis) (or (eq? $dispatch dis) (base.isa? self dis))))
               (import $methods)
               (define $dispatch ($build-dispatch ()))
               (define-syntax $build-dispatch
                 (syntax-rules ()
                   [(_ ((mn mth) dots))
                    (base.$build-dispatch
                      ((mname mname) ... (isa? isa?) (mn mth) dots))]))))))])))
```

Figure 11: The `define-class` macro.

```
(define-syntax make
  (lambda (x)
    (syntax-case x ()
      [(_ class arg ...)
       (with-implicit (class $maker $dispatch)
         #'(let () (import class) (($maker $dispatch ()) arg ...)))])))
```

The `$layout` macro resolves identifiers representing instance variables to locations within an instance structure. The `$layout` macro of each class simply invokes the `$layout` macro of its base class with the set of instance variable identifiers (`id ...`) prepended to the list of instance variables passed in. A call to the `$layout` macro is inserted around the right-hand side of each method definition. The methods of a class are defined within a `$methods` module that is both exported by and imported within the class module.

The `$build-dispatch` macro accumulates an association list correlating method names with references to the actual methods imported from the nested `$methods` module. The `$dispatch` procedure is constructed by invoking the locally defined `$build-dispatch` macro with an empty list of additional methods. This procedure is stored in each instance and serves both as the tag identifying instances of this class and as the dispatching mechanism that implements late binding for methods. The `isa?` method returns true if `$dispatch` is identical to the dispatch routine passed in as an argument or if the `isa?` method from the base class returns true. This method is used by the `isa?` macro defined below.

```
(define-syntax isa?
  (lambda (x)
    (syntax-case x (class)
      [(_ inst cls)
       (with-implicit (cls $dispatch)
         #'(let ([i inst])
             (and (instance? i) (send inst isa? (from cls $dispatch)))))])))
```

Methods are defined with the help of the `method` macro, shown below.

```
(define-syntax method
  (syntax-rules ()
    [(_ fmls . body)
     (lambda (tmp . fmls)
       (fluid-let-syntax ((self (identifier-syntax tmp))) . body))]))
```

This macro inserts an implicit formal parameter `tmp` that will be bound to the instance on which the method is invoked. The identifier `self` is bound to an identifier macro that expands into a reference to the introduced parameter `tmp` wherever `self` appears in the expansion of `body`. Using the simple read-only form of `identifier-syntax` ensures that `self` cannot be assigned.

The interesting details of the implementation are found in the definition of the `<base>` module in

```
(module <base> ($maker $layout $dispatch isa? $build-dispatch)
  (define-syntax $maker
    (syntax-rules ()
      [(_ dispatch (x ...)) (lambda () (vector dispatch x ...))]))
  (define-syntax $layout
    (lambda (x)
      (define build-bind
        (lambda (ids)
          (let lp ([ids ids] [i 1] [bind '()])
            (if (null? ids)
                (reverse bind)
                (lp (cdr ids)
                    (+ i 1)
                    (with-syntax ([id (car ids)] [idx i] [bind bind])
                      #'((id (identifier-syntax
                               (id (vector-ref self idx))
                               ((set! id val) (vector-set! self idx val))))
                         . bind)))))))
      (syntax-case x ()
        [(_ (x ...) body)
         (with-syntax ([bind (build-bind #'(x ...))])
           #'(fluid-let-syntax bind body))])))
  (define $dispatch ($build-dispatch ()))
  (define isa? (method (dis) #f))
  (define-syntax $build-dispatch
    (syntax-rules ()
      [(_ ((mn mth) ...))
       (lambda (msg . args)
         (apply
           (smart-case msg
             [mn mth] ...
             [else (error #f "unknown method ~s" msg)])
           args))])))
```

Figure 12: An implementation of a `base` class module.

Figure 12. The `$maker` macro simply expands into a `lambda` expression that constructs a vector of the arguments to the macro. Efficient inline allocation of instances is possible because of the protocol established by the `define-class` and `make` macros. For example, assuming the class definitions for `<point>` and `<manhattan-point>` given earlier, the `make` macro expands as shown below.[11]

```
(make <manhattan-point> 4 5)
 ⇒ (let ()                           ; make
      (let ([x 4] [y 5])             ; <manhattan-point>'s $maker
        (let ([x x] [y y])           ; <point>'s $maker
          (vector $dispatch x y)))) ; <base>'s $maker
```

Optimizations described in Chapter 4 reduce this code to `(vector $dispatch 4 5)`.

[11]The identifier `$dispatch` would in fact be replaced by a unique identifier as described in Section 3.6.1.

The `$layout` macro uses `fluid-let-syntax` (see page 29) to bind instance variable identifiers to identifier macros during the expansion of the `body` passed to the `$layout` macro. These identifier macros effectively replace each instance variable reference or assignment with a reference or update to the corresponding element of the instance vector. Instance variable offsets begin at `1` to allow for the dispatch procedure stored at index `0`. Because the `method` macro is expanded within the `body` of the `$layout` macro, formal parameters of a method can shadow the instance variables.

The `$build-dispatch` macro exported by the `<base>` module expands into a `lambda` expression that accepts one or more arguments and applies the method selected by the first argument to the remaining arguments, if any. Dispatch is performed by `smart-case`, a variant of the `case` macro, defined below.

```
(define-syntax smart-case
  (lambda (x)
    (syntax-case x ()
      [(_ exp (key e) ...)
       (with-syntax ([((new-key new-e) ...)
                      (let f ([keys (syntax-object->datum #'(key ...))]
                              [cls #'((key e) ...)])
                        (cond
                          [(null? keys) #'()]
                          [(memv (car keys) (cdr keys))
                           (f (cdr keys) (cdr cls))]
                          [else (cons (car cls) (f (cdr keys) (cdr cls)))]))])
         #'(case exp [new-key new-e] ...))])))
```

When `smart-case` is finally invoked by the `$build-dispatch` of the `<base>` module, the list of clauses `((key e) ...)` contains a method name and a reference to the actual method binding for each method defined in the given class and all classes from which it inherits. This list of clauses is ordered so that methods defined in a base class precede methods defined within the class itself. The `smart-case` macro traverses the list of clauses, removing any clause whose key is also present in a subsequent clause. Inherited method definitions are therefore shadowed by local method definitions within a class. The resulting dispatch procedure contains a single `case` expression that associates the name of each method (inherited or otherwise) with a reference to the corresponding method imported from the appropriate module.

The `isa?` method exported by the `<base>` module simply returns false, because the `<base>` class is not intended for direct instantiation. Because the `<base>` class defines no methods, its `$dispatch` procedure signals an error when any method is invoked.

A method is invoked on an instance by extracting the dispatch procedure from the instance and then invoking this procedure with a symbol naming the method, the instance itself, and any arguments expected by the method. This process is abstracted by the `send` macro, defined below.

```
(define-syntax send
  (syntax-rules ()
    [(_ instance msg arg ...)
     (let ([i instance])
       (unless (instance? i) (error 'msg "~s is not an instance" i))
       ((get-dispatcher i) 'msg i arg ...))]))
(define get-dispatcher
  (lambda (inst) (vector-ref inst 0)))
(define instance?
  (lambda (i)
    (and (vector? i) (> (vector-length i) 0) (procedure? (vector-ref i 0)))))
```

A derived class often extends the functionality of its base class by overriding (redefining) one or more methods of the base class. In such cases it is often useful to define the extended method in terms of the method in the base class. This can be done using the `send-base` macro introduced by the `define-class` macro in Figure 11. This macro simply imports from the module representing the base class and invokes the imported method directly on the `self` instance (bound by `fluid-let-syntax` in the `method` macro) and any arguments supplied for that method. Because it makes little sense to invoke the base method directly on any object other than `self`, the `self` argument is implicit in the syntax of `send-base`[12]

Macro-expansion detail

Because the `define-class` macro is somewhat complex, it may be useful to examine the expansion of the following class definition.

```
(define-class (<bronx-point> x y) (<manhattan-point> x y)
  (ivars (count 0))
  [get-count (method () count)]
  [distance
   (method (pt)
     (set! count (+ count 1))
     (send-base distance pt))])
```

This class definition expands into the module definition shown in Figure 13. At top level, that module definition ultimately expands into the following set of variable definitions.

```
(begin (define #{<bronx-point>|get-count}
         (lambda (#:tmp) (vector-ref #:tmp 3)))
       (define #{<bronx-point>|distance}
         (lambda (#:tmp #:pt)
           (vector-set! #:tmp 3 (+ (vector-ref #:tmp 3) 1))
```

[12]In fact, it is seldom appropriate to invoke an arbitrary method via `send-base`. To make the target method implicit as well, the `send-base` macro could be defined locally within each method. While it is not difficult to modify the `define-class` macro to do this, it would introduce additional compile-time overhead.

```
(module <bronx-point> (($maker base.$maker) ($layout base.$layout)
                       ($build-dispatch base.$build-dispatch get-count distance)
                       $methods $dispatch isa?)
  (import-prefixed <manhattan-point> "base.")
  (define-syntax $layout
    (syntax-rules ()
      [(_ (x ...) body) (base.$layout (count x ...) body)]))
  (define-syntax $maker
    (syntax-rules ()
      [(_ dispatch (x ...))
       (lambda (x y) ((base.$maker dispatch (0 x ...)) x y))]))
  (module $methods (get-count distance)
    (define-syntax send-base
      (lambda (x)
        (syntax-case x ()
          [(_ meth-name arg ...)
           (with-implicit
             (_ <manhattan-point> $methods self)
             #'(let ()
                 (import <manhattan-point>)
                 ((from $methods meth-name) self arg ...)))])))
    (define get-count ($layout () (method () count)))
    (define distance
      ($layout ()
        (method (pt)
          (set! count (+ count 1))
          (send-base distance pt)))))
  (define isa?
    (method (dis)
      (or (eq? $dispatch dis) (base.isa? self dis))))
  (import $methods)
  (define $dispatch ($build-dispatch ()))
  (define-syntax $build-dispatch
    (syntax-rules ()
      [(_ ((mn mth) ...))
       (base.$build-dispatch
         ((get-count get-count) (distance distance) (isa? isa?) (mn mth) ...))])))
```

Figure 13: The module definition generated by the `<bronx-point>` class definition.

```
        (#{<manhattan-point>|distance} #:tmp #:pt)))
    (define #{<bronx-point>|$dispatch}
      (lambda (#:msg . #:args)
        (apply
          (case #:msg
            [get-x #{<point>|get-x}]
            [get-y #{<point>|get-y}]
            [delta #{<point>|delta}]
            [get-count #{<bronx-point>|get-count}]
            [distance #{<bronx-point>|distance}]
            [isa? #{<bronx-point>|isa?}]
            [else (error #f "unknown method ~s" #:msg)])
          #:args)))
    (define #{<bronx-point>|isa?}
      (lambda (#:tmp #:dis)
        (or (eq? #{<bronx-point>|$dispatch} #:dis)
            (#{<manhattan-point>|isa?} #:tmp #:dis)))))
```

For expository purposes, the extended identifier syntax `#{<bronx-point>|get-count}` is used to refer to the unique identifier generated to represent the `get-count` method in the `<bronx-point>` class. The generation of unique identifiers is described in Section 3.6.1.

To better understand the expansion of `$layout`, `method`, `send-base`, and the various identifier macros, we now trace the expansion of the `distance` method, starting with the definition as it appears in Figure 13.

```
(define distance
  ($layout ()
    (method (pt)
      (set! count (+ count 1))
      (send-base distance pt))))
```

First, the `$layout` macro of the `<bronx-point>` class expands, introducing a call to the `$layout` macro of the base class. The identifier `count` is added to the list of instance variables in the resulting macro call.

```
(define distance
  (#{<manhattan-point>|$layout} (count)
    (method (pt)
      (set! count (+ count 1))
      (send-base distance pt))))
```

Next, the `$layout` macro of the `<manhattan-point>` class expands into a call to the `$layout` macro of the `<point>` class. The list of instance variables is unchanged.

```
(define distance
  (#{<point>|$layout} (count)
    (method (pt)
      (set! count (+ count 1))
      (send-base distance pt))))
```

The $layout macro of the <point> class expands into a call to the $layout macro of the <base> class and adds x and y to the list of instance variables.

```
(define distance
  (#{<base>|$layout} (x y count)
    (method (pt)
      (set! count (+ count 1))
      (send-base distance pt))))
```

Finally, the $layout macro of the <base> class produces a fluid-let-syntax form that binds each identifier in the list of instance variables to an identifier macro that inserts explicit vector operators. The method macro expands into a lambda expression with an additional formal parameter, tmp, and introduces an identifier macro that rewrites references to self as references to tmp.

```
(define distance
  (fluid-let-syntax                                    ; <base> $layout
    ([x (identifier-syntax                             ; introduces this
          (x (vector-ref self 1))                      ; fluid-let-syntax
          ((set! x val) (vector-set! self 1 val)))]
     [y (identifier-syntax
          (y (vector-ref self 2))
          ((set! y val) (vector-set! self 2 val)))]
     [count (identifier-syntax
              (count (vector-ref self 3))
              ((set! count val) (vector-set! self 3 val)))])
    (lambda (tmp pt)                                   ; method adds tmp formal
      (fluid-let-syntax ([self (identifier-syntax tmp)]) ; and fluid-let-syntax
        (set! count (+ count 1))
        (send-base distance pt)))))
```

When the lambda expression expands, the formal parameters are renamed to locally unique identifiers (indicated here with the #: prefix). The identifier macros rewrite references to self as references to #:tmp and replace the assignment and reference to count with calls to vector-set! and vector-ref. The send-base macro expands into a let form containing an import from the base class and a direct invocation of the distance method imported from the $methods module of that base class.

```
(define distance
  (lambda (#:tmp #:pt)                                ; count identifier macro
    (vector-set! #:tmp 3 (+ (vector-ref #:tmp 3) 1))  ; inserts vector operations
    (let ()                                           ; send-base expands into
      (import <manhattan-point>)                      ; call to imported method
      ((from $methods distance) #:tmp #:pt))))
```

Expanding the import form and from macro results in a reference to the distance method from the <manhattan-point> class.

```
(define distance
  (lambda (#:tmp #:pt)
    (vector-set! #:tmp 3 (+ (vector-ref #:tmp 3) 1))
    ((lambda ()
```

```
        (((lambda () #{<manhattan-point>|distance})) #:tmp #:pt)))))
```

The source-to-source optimizations described in Chapter 4 convert the fully expanded code above into the final form below.

```
(define distance
  (lambda (#:tmp #:pt)
    (vector-set! #:tmp 3 (+ (vector-ref #:tmp 3) 1))
    (#{<manhattan-point>|distance} #:tmp #:pt)))
```

Because the base class is known when the `send-base` macro is expanded, the usual method dispatch is avoided. The resulting code directly invokes the appropriate method which is exported by the base class.

To continue our investigation of the `define-class` macro, we now trace the expansion of the inherited `$build-dispatch` macros starting with the definition of `$dispatch` given in Figure 13.

```
(define $dispatch ($build-dispatch ()))
```

First, the `$build-dispatch` macro of the `<bronx-point>` module expands in terms of the macro imported from the base-class module. The identifiers `get-count`, `distance`, and `isa?` are inserted at the head of the (empty) list pairing symbolic method names with methods. In fact, this list is purely syntactic and does not yet distinguish the ultimate use of these identifiers as symbolic data or variable references. For expository convenience, we use ordinary rather than extended identifiers in positions of the list that will ultimately be used as symbolic data.

```
(define $dispatch
  (#{<manhattan-point>|$build-dispatch}
    ((get-count #{<bronx-point>|get-count})
     (distance  #{<bronx-point>|distance})
     (isa?      #{<bronx-point>|isa?}))))
```

Next, the `$build-dispatch` macro of the `<manhattan-point>` module expands into a call to the macro imported from the `<point>` module. Here the inherited `distance` method is added to the list that will be used to construct the dispatch table.

```
(define $dispatch
  (#{<point>|$build-dispatch}
    ((distance  #{<manhattan-point>|distance})
     (get-count #{<bronx-point>|get-count})
     (distance  #{<bronx-point>|distance})
     (isa?      #{<bronx-point>|isa?}))))
```

The `$build-dispatch` macro of the `<point>` class expands next, introducing the `get-x`, `get-y`, and `delta` methods.

```
(define $dispatch
  (#{<base>|$build-dispatch}
    ((get-x     #{<point>|get-x})
```

```
        (get-y      #{<point>|get-y})
        (delta      #{<point>|delta})
        (distance   #{<manhattan-point>|distance})
        (get-count  #{<bronx-point>|get-count})
        (distance   #{<bronx-point>|distance})
        (isa?       #{<bronx-point>|isa?}))))
```

Next, the `$build-dispatch` macro of the `<base>` module expands.

```
(define $dispatch
  (lambda (msg . args)
    (apply
      (smart-case msg
        [get-x     #{<point>|get-x}]
        [get-y     #{<point>|get-y}]
        [delta     #{<point>|delta}]
        [distance  #{<manhattan-point>|distance}]
        [get-count #{<bronx-point>|get-count}]
        [distance  #{<bronx-point>|distance}]
        [isa?      #{<bronx-point>|isa?}])
      args)))
```

Finally, the `smart-case` macro expands, culling the inherited `distance` method that is redefined by the `<bronx-point>` class.

```
(define $dispatch
  (lambda (msg . args)
    (apply
      (case msg
        [get-x     #{<point>|get-x}]
        [get-y     #{<point>|get-y}]
        [delta     #{<point>|delta}]
        [get-count #{<bronx-point>|get-count}]
        [distance  #{<bronx-point>|distance}]
        [isa?      #{<bronx-point>|isa?}]
        [else (error #f "unknown method ~s" msg)])
      args)))
```

To conclude our investigation of the macros that make up the object system, we trace the expansion of the `make` macro used to instantiate the `bronx-point` class below.

```
(make <bronx-point> 3 4)
```

When expanded, the `make` macro introduces a local contour where it imports the `<bronx-point>` module and invokes the `$maker` macro from that module.

```
(let ()
  (import <bronx-point>)
  (($maker $dispatch ()) 3 4))
```

The effect of the `import` form is to resolve the identifiers as indicated below.

```
(let ()
  ((#{<bronx-point>|$maker} #{<bronx-point>|$dispatch} ()) 3 4))
```

When the `$maker` macro from the `<bronx-point>` module expands, it constructs a `lambda` expression binding `x` and `y`, the instantiation parameters. The body of the `lambda` expression contains a call to the `$maker` macro imported from the base class module, with `0` (the initial value of `count`) extending the (empty) list of initialization expressions.

```
(let ()
  ((lambda (x y)
     ((#{<manhattan-point>|$maker} #{<bronx-point>|$dispatch} (0)) x y))
   3 4))
```

The `$maker` macro from the `<manhattan-point>` module expands next, introducing another `lambda` expression and adding no new initialization expressions (because the corresponding class adds no new instance variables).

```
(let ()
  ((lambda (x y)
     ((lambda (x y)
        ((#{<point>|$maker} #{<bronx-point>|$dispatch} (0)) x y))
      x y))
   3 4))
```

Another `lambda` expression is introduced when the `$maker` macro from the `<point>` module is expanded. References to the `lambda` formals, `x` and `y`, are added to the list of initialization expressions since the corresponding instance variables are initialized with the values of these instantiation parameters.

```
(let ()
  ((lambda (x y)
     ((lambda (x y)
        ((lambda (x y)
           ((lambda (x y)
              ((#{<base>|$maker} #{<bronx-point>|$dispatch} (x y 0))))
            x y))
         x y))
      x y))
   3 4))
```

Finally, the `$maker` macro of the `<base>` module expands into a `lambda` expression with no formal parameters (since the `<base>` class has no instantiation parameters) that allocates a vector containing the dispatch table and the values of the initialization expressions.

```
(let ()
  ((lambda (x y)
     ((lambda (x y)
        ((lambda (x y)
           ((lambda (x y)
              ((lambda () (vector #{<bronx-point>|$dispatch} x y 0))))
            x y))
         x y))
```

```
      x y))
   3 4))
```

The optimizations described in Chapter 4 reduce the macro output to the following.

```
(vector #{<bronx-point>|$dispatch} 3 4 0)
```

Discussion

Because the ancestors of a class are known at compile time, the object system described in this section is quite efficient. First, object instances are allocated inline. In a recent experiment with automatic inline allocation for C++ objects, a threefold performance increase was obtained [25]. Efficient inline allocation makes lightweight objects, such as `<point>`, practical. Second, method dispatch is efficient. All methods applicable to an instance are collected into a single `case` expression in the dispatch routine. Techniques for generating efficient code for these multiway branching constructs are well known [9, 34, 39, 54, 59, 62]. Dispatch overhead is entirely avoided for methods invoked via `send-base`. Local methods of a class can also be invoked directly by methods of the class or by code importing the appropriate module.

To simplify the presentation, we use the "rest" interface of `lambda` and the `apply` primitive to apply a selected method to a list containing its arguments within the dispatch procedure.

```
(define $dispatch
  (lambda (msg . args)
    (apply (case msg ...) args)))
```

A clever compiler[13] can recognize that `args` is referenced only in the call to the `apply` primitive and avoid generating code to (1) build a list of the incoming argument values and (2) load the outgoing argument registers by extracting values from that list. In *Chez Scheme*, this optimization can also be expressed at the source level, using `case-lambda`, as follows.

```
(define $dispatch
  (case-lambda
    [(msg) ((case msg ...))]
    [(msg arg1) ((case msg ...) arg1)]
    [(msg arg1 arg2) ((case msg ...) arg1 arg2)]
    [(msg arg1 arg2 arg3) ((case msg ...) arg1 arg2 arg3)]
    ...
    [(msg . args) (apply (case msg ...) args)]))
```

When invoked, this `$dispatch` procedure jumps directly to the clause matching the number of arguments supplied. The macro generating this alternative dispatch code can trim the `case` expression in each clause of the `case-lambda` by removing each method whose interface does not match the

[13] Also known as a "sufficiently smart" compiler.

number of arguments for that clause. Sorting the methods by arity can accelerate dispatch. In particular, the compiler may be able to generate a tree of conditional branches (which benefit from the branch-prediction logic in modern processors) rather than an indirect jump (the target of which cannot be predicted).

Because it is implemented at the source level, the object system is easily modified. It would be an interesting exercise to generalize the extensibility of the existing system with a compile-time variant of the meta-object protocol [42]. For example, a programmer can change details of object layout and method dispatch by providing an alternative implementation of the `<base>` module. In fact, barring obvious name clashes, these alternative `<base>` modules can coexist in the same executable image and the `define-class` macro can be used, as is, to derive classes from any of them. Supporting additional object representations could introduce additional overhead in the implementation of `instance?` and `get-dispatcher`, however, which could compromise the efficiency of the existing `send` mechanism.

Much of the simplicity and efficiency of the object system derives from the use of modules in the implementation. While it is possible, for example, to combine the various macros associated with a class into a single monolithic macro that recognizes certain input patterns and expands accordingly, the result is difficult to modify or maintain. Modules prevent name conflicts among the auxiliary macros without requiring this kind of monolithic structure. In addition, the analogy between object inheritance and module import facilitates efficient implementation. For example, the ability to import from the various base modules simplifies the implementation of `send-base` and also makes it quite easy to incorporate all inherited methods into a single source-language `case` statement in the dispatch routine. While these things are made possible by the static nature of the object system, the module system simplifies their implementation.

The object system described in this section has several shortcomings. Chief among them is the lack of opaque structures to represent instances. Because instances are represented as vectors, they are vulnerable to arbitrary manipulation by code outside the class, and encapsulation is lost. Strong encapsulation could be implemented efficiently given access to machinery for constructing lower-level representations, such as *Chez Scheme*'s `define-record`. Another limitation of the object system implemented in this section is that it is not possible to refer to a class within itself. For example, it is not possible to create an instance of a class directly within one of its methods.

3.8 Related Work

The module system described in this chapter has its roots in a simple module system that has long been part of Scheme folklore: definitions are encapsulated within a `let` expression and values are exported by assigning appropriately named variables declared outside the `let` form. For example, the following structure could be used to export three values that share access to a common `hash` procedure not visible outside the `let` binding.

```
(define lookup)
(define insert)
(define make-table)
(let ()
  (define hash ---)
  (set! lookup ---)
  (set! insert ---)
  (set! make-table ---))
```

This structure is generalized by the following syntactic abstraction.

```
(define-syntax folk-module
  (lambda (x)
    (syntax-case x ()
      [(_ (v ...) e0 e1 ...)
       (with-syntax ([(t ...) (generate-temporaries (syntax (v ...)))])
         (syntax
           (begin
             (define v) ...
             (call-with-values
               (lambda () e0 e1 ... (values v ...))
               (lambda (t ...) (set! v t) ... (void))))))])))
```

Using the above macro, the preceding example can be expressed as follows.

```
(folk-module (lookup insert make-table)
  (define hash ---)
  (define lookup ---)
  (define insert ---)
  (define make-table ---))
```

The expansion of this macro is shown below.

```
(begin
  (define lookup)
  (define insert)
  (define make-table)
  (call-with-values
    (lambda ()
      (define hash ---)
      (define lookup ---)
      (define insert ---)
      (define make-table ---)
      (values lookup insert make-table))
```

```
(lambda (#tmp1 #tmp2 #tmp3)
  (set! lookup #tmp1)
  (set! insert #tmp2)
  (set! make-table #tmp3)
  (void))))
```

The `call-with-values` form is used here so that both exported and nonexported module bindings may be established via `define` (or macros that expand into `define`). The definitions in the body of the `folk-module` form become internal definitions of the `lambda` expression that produces the values. The values to be exported are returned to the consumer where they are bound to uniquely generated temporary names and the exports are installed via assignment. This arrangement is necessary because the assignments that install the exported values cannot appear in the same scope as the internal definitions else they would alter the local bindings, not the intended external bindings.

The simplicity of the `folk-module` facility is appealing and, indeed, efficient code can be generated from the output of the macro. Yet this facility has several drawbacks. In particular, it does not permit macros (or modules) to be exported. Because the Revised^5 Report on Scheme now incorporates macros [41], a Scheme module system that does not support macros is incomplete. All bindings exported from a `folk-module` are immediately visible everywhere outside that module. There is no support for named modules or selective import. Another difficulty is that the macro expands into a `begin` form containing both definitions and expressions.[14] Consequently it can be placed within a `lambda` body only at the interface between definitions and expressions. This means that at most one `folk-module` can appear within a `lambda` body (or nested within another module). These limitations make `folk-module` a weak foundation upon which to build other syntactic abstractions as this would force the programmer to be aware of the implementation of those abstractions in order to abide by the restrictions. Our `module` form produces efficient code like that produced by `folk-module` and our system addresses the shortcomings of the other system.

Our system is similar in some respects to systems described by Curtis and Rauen [19] and Rees [53]. Our system is based on simpler core module constructs, however, and derives its expressive power largely via syntactic abstraction. While the other systems build in support for separation of interface from implementation, for example, our system allows multiple interface separation mechanisms to be created via syntactic abstraction. Curtis and Rauen forbid the definition and import of modules in local scopes, a feature we have found quite useful. It is not clear whether Rees intends to permit local module definitions. Although the implementation (in Scheme 48) does not

[14] While supported by *Chez Scheme* this is not valid in R^{5}RS Scheme.

forbid such definitions, it provides no apparent way to use them. In both systems, the use of syntactic abstractions expressed in terms of modules is limited. Neither system implements separate compilation.

Rees points out that prohibiting assignment to imported bindings permits the compiler to assume that any variable not assigned within a module is never assigned. The current Scheme 48 implementation, however, does not enforce this restriction. In particular, a Scheme 48 transliteration of our first example in Section 3.6.2 shows that code outside a module can assign an apparently unassigned variable, even if that variable is not exported. To avoid this, the implementation would have to prevent exported macros from expanding into assignments to either exported or nonexported variables. This would limit the expressive power of macros. For example, the `define-parameter` macro described in Section 3.6.2 could not be used to define exported parameters. Curtis and Rauen claim a similar restriction but give few details and no implementation.

Curtis and Rauen propose a *meta-module* facility that permits the modularization of the code used in transformer expressions. In particular, they allow macro transformers to share code. Meta-modules are similar to ordinary modules except that they are evaluated at expand time and export only variable bindings. This is a useful feature that our system currently lacks, although nothing in our system precludes support for such a feature. We can achieve a similar effect, though with less structure, by abusing the top-level environment.

Queinnec and Padget describe a module language for controlling the visibility of sets of named locations [50, 51]. They describe a high-level macro-expansion protocol intended to support various macro-system implementations such as expansion-passing style [28] or hygienic systems based on syntactic closures [17]. Because their system is not tightly integrated with the macro expander they do not permit local modules, nor do they permit macros to expand into module constructs.

The Dylan programming language [56] supports both modules and lexically scoped macros. Their macro system is more restrictive in its treatment of symbols and identifiers. Macro and module definitions can appear only at the top level of a compilation unit (library), and import is tied to the module syntax, limiting expressiveness. To determine the set of identifiers implicitly exported by macros, the body of each macro transformer is analyzed. Because it is generally impossible to compute the precise set of implicitly exported bindings, an upper bound is computed by assuming exported any binding with the same name as an identifier appearing in the right-hand side of a rewrite rule.

We considered implementing a similar analysis in our system but ultimately dismissed this approach for two reasons. First, by foiling the (necessarily conservative) analysis, seemingly innocent edits can have unanticipated performance consequences. For example, inserting a quoted symbol in a particular context could cause the like-named identifier to be implicitly (and unintentionally) exported. Second, our mechanism for intentional capture, `datum->syntax-object`, is sufficiently powerful that all identifiers are potential implicit exports, as demonstrated in Section 3.6.2. While Dylan provides a limited form of intentional capture via the macro template modifier `?=`, this is insufficient to express macros such as `include`. It may, however, be sufficiently powerful to present difficulties for the analysis they propose.

Haskell's module system is also based on the manipulation of lexical scope [38]. Consequently, modules in Haskell are similar in many ways to modules in our system. In particular, modules may be mutually recursive and, being intended for name-space control, are not first-class values. Unlike our modules, Haskell modules must be closed (see Section 3.6.4), may appear only at top level within programs, and do not permit the export of macros or modules. Like macros, types must be available at compile time. Since Haskell permits the export of types, it is sometimes necessary to provide explicit type signatures for exported values when separately compiling mutually recursive modules. A similar restriction was imposed in Section 3.4.8.

Flatt and Felleisen propose a mechanism for dynamically linking separately compiled program units [30, 31]. While it does not permit the import or export of macros, the `unit` facility is interesting as representative of systems providing both higher-order modules and programmatic control over linking. They argue against module linkage via static import and instead propose a mechanism that provides programmatic control over module linkage. We believe that the interfaces should be specified statically whenever possible as an aid to static program analysis (both by the programmer and by the compiler). However, the static nature of our system does not preclude support for dynamic module linkage. In fact, the `unit` module system can be implemented in our language entirely at the source level, as demonstrated in Section 3.7.

Chapter 4

Fast and Effective Procedure Integration

This chapter describes a fast and effective technique for procedure integration, or inlining. The chapter is organized as follows. Section 4.1 provides an overview of the algorithm. Section 4.2 describes the various optimizations and explains where the new inlining pass fits relative to existing passes of the compiler. Section 4.3 presents an unrestrained inlining algorithm that concentrates on recognizing inlining opportunities without attempting to select appropriately among them. Section 4.4 shows how this simplified algorithm is restrained so that it makes appropriate inlining decisions. Section 4.5 describes special treatment for procedures referenced just once. Section 4.6 discusses extensions that permit better handling of recursive procedures and certain primitives. Section 4.7 provides detailed measurements of the effect of the algorithm on execution time, code size, and compile time in a production compiler. Section 4.8 describes related work and compares our results with those obtained by offline, flow-analysis-based inlining algorithms. Section 4.9 proposes future work.

4.1 Introduction

Inlining is an important optimization for programs that use syntactic and procedural abstraction. By replacing a call with an inline expansion of the procedure body, inlining eliminates procedure linkage overhead and may uncover opportunities for further optimization. Unrestrained inlining, however, can lead to excessive code growth and excessive compile time. The effectiveness of an inlining algorithm is therefore determined not only by its ability to recognize inlining opportunities but also by its discretion in exercising those opportunities.

Recent research on procedure inlining for higher-order languages has focused on the use of offline polyvariant flow analyses to identify appropriate inlining opportunities [35, 5]. Although reported performance gains are impressive, the cost and volatility of such analyses renders the method impractical for use in production compilers, especially load- or run-time compilers whose overhead contributes to overall program run time. This chapter presents a new inlining algorithm for higher-order languages that combines simple analysis techniques with online transformation to achieve comparable or better performance gains in fast linear time.

Several factors contribute to the speed and effectiveness of our algorithm: First, it is polyvariant. Information about the arguments at a call site is used to decide whether inlining that call is cost-effective. Second, it is online. Inlining decisions are based, in part, on the effects of other optimizations performed by the algorithm: constant folding, copy propagation, elimination of useless or unreachable code, and inlining itself. Third, it is context-sensitive. The context in which an expression occurs is used to identify legal transformations. Finally, it is demand-driven. The operands of binding constructs and calls are not processed until they are needed, at which point the context in which they are used has been determined.

Polyvariance permits aggressive inlining without excessive code growth. Code growth is often limited by inlining only procedures whose size is less than a given threshold. Our polyvariant algorithm instead considers the size of the procedure when optimized for the arguments at the call site. For example, given the following definition of `f`, the algorithm may choose to inline `e2` in place of the call `(f 'key2)` even if the size of `f` as a whole exceeds the threshold.

```
(define f
  (lambda (x)
    (case x
      [(key1) e1]
      [(key2) e2]
      ...
      [(keyn) en])))
```

Online transformation provides our algorithm with more accurate information on which to base its inlining decisions. Offline algorithms, which separate analysis, inlining, and subsequent optimization, must estimate the effects of these optimizations. When their estimates are too conservative, inlining opportunities are missed; when their estimates are too liberal, code size increases. Our algorithm inlines aggressively while maintaining tighter control over code growth because it uses the actual optimized code to decide when inlining a call is cost-effective. In some cases, online transformation also provides more accurate control-flow information, which helps to identify additional inlining opportunities.

Contextual information allows the algorithm to determine when an expression can be discarded or processed in a more restricted mode. For example, in the expression `(begin e1 e2)`, `e1` is processed only for its effects. When processed for effect, certain provably terminating effect-free expressions are discarded. In languages such as C and Scheme it is sometimes possible to determine the boolean value of an expression at compile time even when its actual value cannot be determined. This is sufficient when processing the guard of a conditional expression.

Demand-driven processing of call operands allows our algorithm to defer processing until more contextual information is available. For example, suppose `f` is `(lambda (x y) (if (< x 5) x y))`. When attempting to inline the call `(f e1 e2)`, `e1` is not processed until it is needed to evaluate the test `(< x 5)`. If `e1` reduces to `3`, the reference to `y` is unreachable and the useless expression `e2` is processed for effect only. Eliminating useless or unreachable code saves processing time and also improves the accuracy of the information collected by the algorithm. For example, if variables originally referenced only within `e2` no longer appear referenced, their bindings can be processed for effect only.

The inlining algorithm is a source-to-source transformation that operates by a demand-driven traversal of the source program. The algorithm collects information about the size of the residual program and the use of variables in the residual program as it is produced. This information supports inlining decisions and elimination of useless code. Termination is achieved by detecting cycles in the call graph as it is computed and by bounding the amount of effort expended on inlining at each call site in the input program. Code growth is limited by aborting an inlining attempt as soon as the size of the residual code produced in that attempt exceeds a given threshold. Polyvariance is obtained by attempting to inline all calls, while relying on the effort bound and the size threshold to abort an attempt when it becomes too costly. Applicative order and sequencing of effects are preserved by optimizing inlined calls as if they were equivalent `let` expressions. For example, suppose `f` is `(lambda (x y) e1)`. When inlining, the call `(f e2 e3)` is treated as the semantic equivalent of `(let ((x e2) (y e3)) e1)`. Values obtained by processing `e2` and `e3` are used when specializing `e1`, but effects in `e2` and `e3` are residualized at the point of call.

4.2 Background

Consider the following program fragment.

```
(let ()
  (define min
    (lambda (<)
      (lambda (x y) (if (< x y) x y))))
  (lambda (a b c)
    (or a ((min fx<=) b c))))
```

While small, this fragment is intended to be evocative of real programs in its use of procedural and syntactic abstractions. The procedure `min` takes a procedure `<` as an argument and returns a procedure that uses `<` to determine the minimum of its inputs. Here is the program after macro expansion.

```
(let ()
  (define min
    (lambda (<)
      (lambda (x y) (if (< x y) x y))))
  (lambda (a b c)
    (let ([tmp a])
      (if tmp tmp ((min fx<=) b c)))))
```

Now consider the same program after optimization.

```
(lambda (a b c)
  (if a a (if (fx<= b c) b c)))
```

The binding for `min` has been discarded. The binding for `tmp` blindly introduced in the expansion of `or` has been eliminated. The nontail call to `min` is gone and with it the need for a stack overflow check, saving and restoring argument registers containing `b` and `c`, and the allocation of a closure with free variable `<`. Optimization has also removed the anonymous tail call to the procedure returned by `min`. The top-level value of `fx<=` is no longer obtained or passed to `min`. The primitive `fx<=` can now be open-coded, eliminating an anonymous nontail call and the associated register saves and restores. As a result, the compiled code is five times smaller and runs nearly three times faster.

The remainder of this section describes the transformations performed by the inlining optimization pass and concludes by placing this pass in the context of the existing compiler passes. The standard reference for many of the optimizations described here is [1].

4.2.1 Eliminating Useless or Unreachable Code

Eliminating useless or unreachable code reduces the number of instructions executed and may reduce contention for resources such as registers. An expression is *useless* if its evaluation causes no

`(let ([x e1]) e2)`	→	`(begin e1 e2)`	; if `x` is not referenced
`(set! x e1)`	→	`(begin e1 (void))`	; if `x` is not referenced
`(begin e1 e2)`	→	`e2`	; if `e1` is effect-free and halts
`(if e1 e2 e3)`	→	`(begin e1 e3)`	; if `e1` is always false

Figure 14: Examples illustrating elimination of useless or unreachable code.

observable side effect and the resulting value is not used. Bindings and assignments to unreferenced variables are useless. A pure expression evaluated for effect is also useless, provided its evaluation always terminates. An expression is *unreachable* if no possible execution of the program will evaluate that expression. The consequent (alternative) branch of a conditional whose guard always evaluates to false (true) is unreachable. Figure 14 shows elimination of useless or unreachable code and the conditions that permit these transformations.

4.2.2 Constant Propagation

When an unassigned variable is bound to an expression denoting a constant value, the compiler can often replace references to that variable with the text of the expression. For example, the expression `(let ([x 3]) (+ x 4))` becomes `(let ([x 3]) (+ 3 4))`. Here constant propagation renders the binding for `x` useless and presents the opportunity to compute `(+ 3 4)` at compile time.

Constant propagation can also reduce register pressure and improve instruction selection in later passes of the compiler. For example, after constant propagation small constants can be encoded as immediate operands in the instruction stream instead of occupying registers. While other constants may need to be loaded into registers before use, replacing variable references with constants effectively splits and shrinks the live ranges of those variables so that they span only the individual instructions that manipulate those variables. Instruction selection is improved when, for example, an operand of a multiply instruction is found to be a constant power of two, permitting the use of a shift instruction instead.

Typically only immediate constants are copied. Copying structured constants, such as strings, can break object identity. For example, the following transformation is not generally valid.

`(let ([x "no"]) (f x x))` $\not\to$ `(f "no" "no")`

Suppose `f` is bound to the function `eq?`, which tests for pointer equality of its arguments. The expression on the left then evaluates to `#t`. The expression on the right will not, however, unless the compiler guarantees that the two strings in fact refer to the same memory location. Because it preserves object identity for constants through all stages of compilation and linking, however, *Chez*

Scheme can perform this transformation.

4.2.3 Constant Folding

When an effect-free primitive is applied to constant arguments and the result is constant, the computation can be performed at compile time. Constant folding permits the compiler to replace the expression `(+ 3 4)` with the constant `7` and to replace `(car '(1 2))` with `1`, for example, reducing the number of instructions executed.

Although the arguments are constant, the compiler cannot rewrite `(cons 1 2)` as `'(1 . 2)` because the result is not constant; each call to `cons` must return a fresh pair. Moreover, pairs returned by `cons` are mutable, while constants such as `'(1 . 2)` are not. Naturally, the compiler cannot replace the expression `(random 100)` with a constant not only because that would be a terrible approximation of a random function, but also because `random` performs a side effect to update the seed value, and this side effect can be observed by subsequent calls to `random`.

In some cases constant folding is possible even when some of the arguments are not constant. For example, `member` is a function that searches for a value matching its first argument in the list supplied as its second argument. The expression `(member e1 '())` can therefore be rewritten as `(begin e1 #f)` because the list is empty and the search must fail. The sequence can be eliminated if `e1` is effect-free and terminating.

4.2.4 Copy Propagation

When one unassigned variable is bound to another unassigned variable, the compiler can replace references to one with references to the other. For example, after the useless binding for `y` is removed, the expression `(let ((x e1)) (let ((y x)) (+ x y)))` becomes `(let ((x e1)) (+ x x))`. By rendering some bindings useless, copy propagation reduces register pressure.

To see why we perform copy propagation only for unassigned variables, consider the following example. For notational convenience, assume `y` does not occur free in `e2`.

```
(let ([x e1])
  (let ([y x])          (let ([x e1])
    (set! x e2)    ↛      (set! x e2)
    (+ y x)))             (+ x x))
```

In the code on the left, `x` is bound to the value of `e1`, then `y` is bound to the value of `x` before `x` is assigned a new value, and finally sum of the old and new values is returned. The code on the right instead doubles the value of `e2`; the value of `e1` is lost.

It may seem that the problem in the previous example is simply that x is assigned within the scope of the binding for y. As the following example shows, however, the problem is more complicated.

```
(let ([x e1])
  (let ([z (let ([y x]) (lambda () y))])
    (set! x e2)
    (+ (z) x)))
```

Here x is assigned not in the scope of the binding for y, but within the dynamic extent of that binding. If the reference to y in (lambda () y) were replaced with x, the call to z would return the new value of x, not its old value as intended.

Because traditional imperative languages lack expression-level binding constructs, compilers for those languages often rely on static single assignment analysis [20] to determine which of the assignments are, in fact, invariant bindings. This information can be used to perform both constant propagation and copy propagation. In Scheme such constructs do exist and the use of assignment is discouraged, therefore a static single assignment analysis would likely provide little benefit.

4.2.5 Procedure Inlining

When the compiler can determine that only one procedure can be reached from a given call site, the call can be replaced with an inline expansion of the procedure body where the formal parameters are bound locally to the actual parameters. Thus if min is bound to (lambda (x y) (if (< x y) x y)), the call (min e1 e2) can be rewritten as (let ([x e1] [y e2]) (if (< x y) x y)). The local bindings are introduced to preserve the applicative-order, call-by-value semantics of the language [11]. In particular, the actual parameters must be evaluated exactly once, their values must be bound to fresh locations, and evaluation of the actuals must precede evaluation of the body.[1]

Procedure inlining can be viewed as a generalization of constant propagation to lambda expressions. A lambda expression evaluates to a fresh closure, and although the closure is not generally[2] constant, it contains a code pointer that is.[3] Thus, while a lambda expression cannot be copied into arbitrary contexts, it can be propagated into the operator position of a call, since there only the code is of interest. Just as constant propagation enables constant folding, inlining enables β-reduction.

The obvious benefit of inlining is that it eliminates some procedure call overhead. A nontail call may involve allocating a new stack frame, checking for stack overflow, saving and restoring registers around the call, extracting a code pointer from a closure, and an indirect jump. Inlining

[1] Of course continuation operations in e1 or e2 could cause the other expression to be evaluated zero or more times.
[2] A closure with no free values or constant free values may be treated as constant.
[3] Closures are immutable.

```
(define len
  (lambda (x)
    (cond
      [(vector? x) (vector-length x)]
      [(string? x) (string-length x)]
      [(integer? x) (integer-length x)]
      [(list? x) (length x)]
      [else (error 'len "unexpected object ~s" x)])))
```

Figure 15: A simple polymorphic function.

can eliminate some or all of this overhead. When a procedure is inlined at each of its call sites, the cost of allocating the closure can be avoided as well.

A more important benefit of inlining is that it enables other optimizations. Consider the following definitions.

```
(define f (lambda (x) (+ x 4)))
(define g (lambda (n) ... (f 3) ... (f n)))
```

Inlining `f` at the call site `(f 3)` produces `(let ([x 3]) (+ x 4))`, enabling constant propagation and constant folding as we saw in Sections 4.2.2 and 4.2.3. Inlining `f` at the call site `(f n)` yields `(let ([x n]) (+ x 4))`, enabling copy propagation as in Section 4.2.4. When a procedure body is expanded inline at a call site, intraprocedural optimizations effectively acquire interprocedural strength. This is significant because intraprocedural optimizations, while perhaps less effective, are often more efficient than their interprocedural cousins.

Other benefits of inlining include its potential to eliminate currying and higher-order procedures by integrating procedural arguments and return values. Inlining can also reduce polymorphism by reducing the number of call sites that a procedure must satisfy. For instance, if the procedure in Figure 15 is inlined wherever it is called with a vector or string argument, type inference on the resulting program may reveal that `len` is called only with integers or lists. The compiler might exploit this finding by deleting the vector and string cases from the definition of `len`. Reducing polymorphism can improve the generated code and may also improve the accuracy of certain compile-time analyses. Finally, inlining can convert mutual recursion to direct recursion [40] as illustrated in Figure 16.

One potential cost of inlining is increased code size. Replicating a procedure body at each of its call sites can certainly increase the size of the program. Yet, by enabling other optimizations, inlining may also reduce code size. Code size is an important consideration due to the impact of the memory hierarchy on program performance [48]. If the code becomes too large to fit within

```
(letrec ([even? (lambda (x) (or (zero? x) (odd? (- x 1))))]
         [odd? (lambda (x) (not (even? x)))])
  (lambda (n) (even? n)))
 →
(letrec ([even?
          (lambda (x)
            (or (zero? x) (let ([x (- x 1)]) (not (even? x)))))])
  (lambda (n) (even? n)))
```

Figure 16: Inlining `odd?` converts mutual recursion to direct recursion.

the instruction cache, for example, the effect on program run time can be disastrous and can easily overwhelm the benefits derived from inlining.

Inlining may result in larger basic blocks, and this can increase register pressure. At the same time, inlining removes a key obstacle to effective register usage, the procedure call boundary. At a call boundary, the caller does not know which caller-save registers the callee might use and must therefore save any that are live across the call. Similarly, the callee does not know which of the callee-save registers are actually in use by its callers and must therefore save any that it wishes to use. In either case, registers may be saved needlessly. By eliminating the division between caller and callee, inlining may in fact permit more effective use of registers.

Another potential cost of inlining is increased compile time. A new compiler pass must be added to do the work of inlining and naturally this may increase compile time. Moreover, inlining is a front-end optimization; if it increases code size, then each pass of the compiler back-end operates on a larger program and compile time may suffer as a result. Thus there are both run-time and compile-time incentives for keeping tight control over code growth.

A simple way to limit code growth is to inline a procedure only if the size of its body is below a given threshold. This is unnecessarily conservative since other optimizations can reduce the size of a procedure once inlined. In contrast, a *polyvariant* approach bases the inlining decision at each call site on the size of the procedure when optimized for the arguments at that call site. Polyvariance permits more aggressive inlining without an increase in the size threshold. Even large procedures are candidates for inlining, but they are inlined only where the size of the resulting residual code is below the threshold. The result is better performance with tight control over code growth.

4.2.6 Compiler Organization

Figure 17 shows the new optimization pass (cp0) in the context of the existing compiler passes. The reader performs lexical analysis and parsing. The expander performs macro expansion and collects

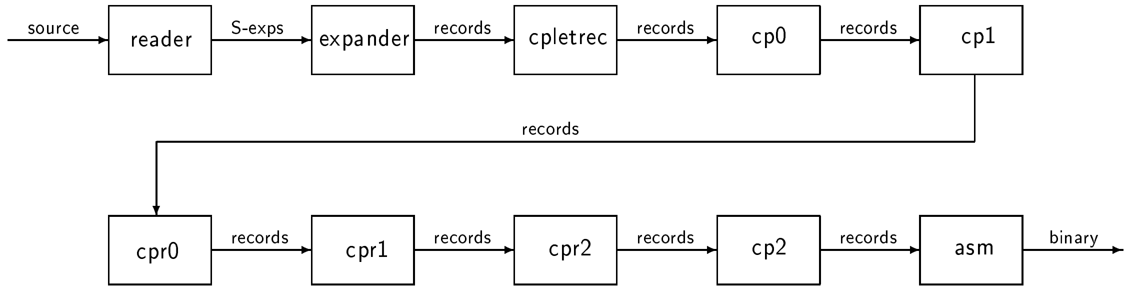

Figure 17: The passes of the *Chez Scheme* compiler.

conservative information about the use of variables, as described in Section 2.5. The cpletrec pass rewrites `letrec` expressions in a canonical form that can be treated more efficiently by subsequent passes. Primitives are open-coded in cp1. This pass also recognizes loops and performs strength reduction. Assignments are converted to explicit structure updates in cpr0. This pass also flags tail calls and loops, assigns locations to variables, performs direct-call and closure optimizations, and begins the process of register allocation. Pass cpr1 inserts register saves and restores; it also chooses an order of evaluation. Pass cpr2 removes redundant register restores. The cp2 pass generates binary code in core with the help of various architecture-specific assemblers.

The intermediate representation of the program becomes increasingly explicit from one pass to the next. One reason cp0 is placed early in the front end is that the intermediate language is more flexible earlier in compilation. This flexibility means that cp0 can perform aggressive transformations without invalidating register allocation decisions, for example, as might happen if it were placed later in the back end. Operating on a higher-level representation also gives cp0 greater leverage. Another reason cp0 is placed in the front end is that *Chez Scheme* includes an interpreter that accepts precisely the language accepted by cp1. Both the compiler and interpreter benefit from optimizations performed by cp0.

The cpletrec transformation is worth describing here since it benefits cp0. Bindings are partitioned into four sets: *unreferenced*, *simple*, *lambda*, and *complex*. A binding is *unreferenced* if the bound variable is unreferenced. When an unassigned variable is bound to an expression that contains no `lambda` expressions, no references to other variables bound by the `letrec`, and no calls except to certain safe primitives, that binding is *simple*. A binding is *lambda* if the variable is unassigned and the value is a `lambda` expression. A binding is *complex* if it does not fit any of the other categories.

```
(letrec ([f (letrec ([g (let ([x 5]) (lambda () ...))])
              (lambda () ... g ...))])
  f)
→ (let ((x 5))
     (letrec ([f (lambda () ... g ...)]
              [g (lambda () ...)])
       f))
```

Figure 18: The cpletrec pass lifts out certain nested bindings.

In the output of cpletrec, simple bindings are lifted and bound by a `let` outside the `letrec`. Variables with complex bindings are lifted to the same `let` where they are bound to an unspecified value. Within the body of the `letrec`, these variables are assigned the values of local temporaries bound to the complex expressions. The `letrec` itself binds `lambda` expressions only. Expressions bound to unreferenced variables are placed in a sequencing form within the body of the `letrec`. Thus if `(xs vs) ...`, `(xl vl) ...`, `(xu vu) ...`, `(xc vc) ...` are the sets of simple, lambda, unreferenced, and complex bindings respectively, then the output looks like the following.

```
(let ((xs vs) ... (xc (void)) ...)
  (letrec ((xl vl) ...)
    (begin vu ...
      (let ((xt vc) ...)
        (set! xc xt) ...)
        body)))
```

The actual transformation is a bit more complicated. Bound expressions are processed bottom-up and the result is considered complex only if it does not fit the other categories and it is not a `let` or `letrec` expression whose bindings can be folded into the outer `let` or `letrec`. Thus cpletrec can perform the transformation depicted in Figure 18.

4.3 Unrestrained Inlining Algorithm

This section presents an unrestrained inlining algorithm with a focus on recognizing inlining opportunities. In Section 4.4 we focus on selecting appropriately among these opportunities.

Figure 19 shows the unrestrained algorithm. The main procedure, `cp0`, takes an input expression `exp`, a context `ctxt`, and an environment `env`. The input expression is in the core language shown in Figure 20. As described in Section 4.2.6, source expressions are translated into this core language by the macro expander and subsequent cpletrec pass. The inlining algorithm returns a residual expression in the same core language. To simplify the presentation, `lambda` abstractions are restricted to a single formal parameter, applications (`call` forms) are restricted to a single operand, and `letrec`

```
(define cp0
  (lambda (exp ctxt env)
    (exp-case exp
      [(const c) exp]
      [(ref x) (cp0-ref x ctxt env)]
      [(begin e1 e2)
       (let ([e1 (cp0 e1 'effect env)] [e2 (cp0 e2 ctxt env)])
         (make-seq e1 e2))]
      [(if e1 e2 e3)
       (let ([e1 (cp0 e1 'test env)])
         (exp-case (result-exp e1)
           [(const c) (make-seq e1 (cp0 (if c e2 e3) ctxt env))]
           [else (let ([ctxt (if (app? ctxt) 'value ctxt)])
                   (let ([e2 (cp0 e2 ctxt env)] [e3 (cp0 e3 ctxt env)])
                     (if (record-equal? e2 e3 ctxt)
                         (make-seq e1 e2)
                         `(if ,e1 ,e2 ,e3))))]))]
      [(set! x e)
       (let ([x (lookup x env)])
         (make-seq
           (if (not (prelex-source-referenced? x))
               (cp0 e 'effect env)
               (let ([e (cp0 e 'value env)])
                 (set-prelex-residual-assigned! x #t)
                 `(set! ,x ,e)))
           `(const ,(void))))]
      [(call e1 e2) (cp0-call e1 (make-operand e2 env) ctxt env)]
      [(primref p)
       (context-case ctxt
         [(app) (fold-prim p ctxt)]
         [(value) exp]
         [(test effect) `(const #t)])]
      [(lambda (x) e)
       (context-case ctxt
         [(app) (inline exp ctxt env)]
         [(value) (with-extended-env ((env x) (env x #f))
                    `(lambda (,x) ,(cp0 e 'value env)))]
         [(test effect) `(const #t)])]
      [(letrec ((x e1)) e2)
       (with-extended-env ((env x) (env x #f))
         (let ([opnd (make-operand e1 env)])
           (set-prelex-operand! x opnd)
           (let ([e2 (cp0 e2 ctxt env)])
             (if (or (not (prelex-residual-referenced? x))
                     (exp-case e2 [(const c) #t] [(primref p) #t] [else #f]))
                 e2
                 `(letrec ((,x ,(operand-value opnd))) ,e2)))))])))
```

Figure 19: The unrestrained inlining algorithm.

$$\begin{array}{rcl} e & ::= & \texttt{(const } c\texttt{)} \mid \texttt{(ref } x\texttt{)} \mid \texttt{(primref } p\texttt{)} \mid \texttt{(call } e_0\ e_1\texttt{)} \\ & \mid & \texttt{(if } e_1\ e_2\ e_3\texttt{)} \mid \texttt{(begin } e_1\ e_2\texttt{)} \mid \texttt{(set! } x\ e\texttt{)} \\ & \mid & \texttt{(lambda (}x\texttt{) } e\texttt{)} \mid \texttt{(letrec ((}x\texttt{ (lambda (}x_1\texttt{) } e_1\texttt{))) } e_2\texttt{)} \end{array}$$

$$c \in \mathit{Const},\ p \in \mathit{Prim},\ x \in \mathit{Var},\ e \in \mathit{Exp}$$

Figure 20: The core language.

forms are restricted to a single binding. Generalization to multiple-argument and variable-arity procedures is straightforward.

Contextual information (`ctxt`) is used to identify situations in which certain transformations are legal. Four contexts are distinguished: *test*, used when processing an expression for its boolean value, *effect*, used when processing an expression for side effects only, *value*, used when the actual value is needed, and *call*, used when processing an expression in the operator position of an application. Observable side effects are conserved in all contexts. Test, effect, and value contexts are represented by the constant symbols `test`, `effect`, and `value`. Call contexts are represented by structures defined as follows.[4]

```
(define-structure (app opnd ctxt) ((inlined #f)))
```

The `opnd` field contains an operand structure (described below) for the actual parameter at the call site. The outer context of the call is stored in the `ctxt` field. The `inlined` flag is set if this call is inlined and is consulted when residualizing a call.

The environment, `env`, maps source-program variables to residual-program variables. The initial environment `empty-env` is the identity function on variables. Source- and residual-program variables are represented as `prelex` structures minimally defined as follows.

```
(define-structure (prelex name operand)
  ([source-referenced? #f]
   [source-assigned? #f]
   [residual-referenced? #f]
   [residual-assigned? #f]))
```

The `name` field contains the name of the variable for display purposes. When a variable is bound as the formal parameter of an inlined procedure or bound by a `letrec` expression, the `operand` field contains an operand structure (described below). The remaining fields are flags that describe uses of the variable in the source and residual program. The `referenced` flag supports elimination of

[4]This macro (from [27]) defines a predicate, `app?`, that tests whether an object is a structure of this type, as well as accessors and mutators for the various fields, e.g., `app-opnd`, and `set-app-inlined!`. The implicit `inlined` field is automatically initialized to false (`#f`) when the structure is allocated.

useless assignments and useless bindings. The `assigned` flag is used to prevent copy propagation and inlining through assigned variables. The macro expander introduces `prelex` structures for variables and initializes the usage flags; `cp0` uses and refines this conservative information. In fact, because it maintains consistent information for program variables, and because its source and residual languages are the same, `cp0` could be iterated if desired.

Operand structures support deferred processing for formal parameters and `letrec` bindings. They are defined as follows.

```
(define-structure (operand exp env) ((value #f) (residualize-for-effect #f)))
```

Here `exp` contains the source expression corresponding to the actual parameter or bound value, `env` is an environment binding free variables of `exp`, and `value` is used to cache the residual code for `exp` once computed. The `residualize-for-effect` flag is used when residualizing a call.

The remainder of this section explains how the algorithm handles the core forms of the language. The presentation follows the order of the cases in Figure 19.

4.3.1 Constants

Handling constants is straightforward. When processed directly by `cp0`, constant source expressions are returned unaltered. Residual constants are treated specially when processing conditional expressions or constructing residual sequences as described below.

4.3.2 Sequences

The expression `(begin e1 e2)` evaluates `e1` for its effects, discards its result, and evaluates and returns the value of `e2`. Therefore, `cp0` processes `e1` in an `effect` context, processes `e2` in the current context (`ctxt`), and uses `make-seq` (see Figure 21) to construct a residual sequence if necessary. When processed for effect, constants, primitive references, `lambda` expressions, variable references, and effect-free primitive calls all reduce to constants and are discarded by `make-seq`.

A residual sequence is constructed when `cp0` is unable to prove `e1` effect-free. Otherwise, the first argument to `make-seq` will satisfy the `simple?` predicate, and no sequence is constructed. Residual sequences are nested only in the first position. This invariant is maintained with constant overhead and permits constant-time access (via `result-exp`) to the result expression of a residual sequencing form, making aggressive propagation of values more efficient. The following examples show `make-seq` in use.

```
(define make-seq
  (lambda (e1 e2)
    (if (simple? e1)
        e2
        (let ([e1 (exp-case e1
                    [(begin e1a e1b) (if (simple? e1b) e1a e1)]
                    [else e1])])
          (exp-case e2
            [(begin e3 e4) `(begin (begin ,e1 ,e3) ,e4)]
            [else `(begin ,e1 ,e2)])))))
(define result-exp
  (lambda (e)
    (exp-case e
      [(begin e1 e2) e2]
      [else e])))
(define simple?
  (lambda (exp)
    (exp-case exp
      [(const c) #t]
      [(ref x) #t]
      [(lambda (x) body) #t]
      [(primref p) #t]
      [else #f])))
```

Figure 21: Procedures for constructing and accessing residual sequences.

```
(make-seq `(ref x) `(const 4))
  ⇒  (const 4)
(make-seq `(set! x (const 3)) `(const 4))
  ⇒  (begin (set! x (const 3)) (const 4))
(make-seq `(begin (set! x (const 3)) (const 4)) `(ref y))
  ⇒  (begin (set! x (const 3)) (ref y))
(make-seq `(set! x (const 3)) `(begin (set! y (ref x)) (ref y)))
  ⇒  (begin (begin (set! x (const 3)) (set! y (ref x))) (ref y))
```

4.3.3 Conditional Expressions

The guard of a conditional expression is processed for its boolean value using a `test` context. When processed for boolean value, primitive references, `lambda` expressions, and certain effect-free primitive calls reduce to the constant `#t`. If the `result-exp` of the residual guard expression is constant, the flow of control can be determined at compile time and `make-seq` is used to construct a sequence of the guard (if not effect-free) and the selected branch.

When `cp0` cannot determine the flow of control through a conditional expression it is processing in a call context, it conservatively substitutes `value` for the `app` context before processing the branches. This prevents interleaved evaluation of the operator and operand expressions in a call. Interleaved evaluation is prohibited by the language semantics but could easily occur if a call context were propagated into both branches of a conditional as in the example below.

```
((if e1 e2 e3) e4) ↛ (if e1 (e2 e4) (e3 e4))
```

In the code on the left, the `if` expression is evaluated as a unit either before or after the evaluation of `e4`. In the code on the right, it is possible to evaluate `e4` between the evaluation of `e1` and `e2` or `e3`. While conservatively processing the branches in a `value` context prevents interleaved execution, it also prevents useful transformations such as the following.[5]

```
((if e1 cdr car) e2)  →  (if e1 (cdr e2) (car e2))
```

When the flow of control through a conditional expression cannot be determined and the residual expressions for the two branches are equivalent in the current context, the algorithm residualizes a sequence of the test and one of the branches.[6] This decision is made by `record-equal?` defined below.

```
(define record-equal?
  (lambda (e1 e2 ctxt)
    (exp-case e1
      [(const c1)
       (exp-case e2
         [(const c2)
          (context-case ctxt
            [(effect) #t]
            [(test) (if c1 c2 (not c2))]
            [else (eq? c1 c2)])]
         [else #f])]
      [else #f])))
```

Two residual expressions are deemed equivalent only if both are constant and either the context is `effect`, or the context is `test` and both constants have the same boolean value, or the two constants are identical. We do not compare the `result-exp` of the two expressions because branches that yield equivalent constants may still differ in side effects.

It may appear that this transformation is seldom useful. Indeed, a programmer is not likely to write `(if e1 #f #f)`. The flattening of values in `test` and `effect` contexts can produce such cases, however. Consider the following example.

```
(let ((x 3))
  (if (and (read) (= x 0))
      e1
      e2))
```

[5] Alternatively `cp0` could extend the environment with a fresh variable `x` containing the call-site operand then process `` `(call ,e2 (ref ,x)) `` and `` `(call ,e3 (ref ,x)) `` in place of the branches `e2` and `e3`. A residual `let` binding can be constructed if `x` is referenced. The operands of enclosing `app` contexts must also be bound to temporaries to avoid excessive code duplication.

[6] A similar transformation can be made by online partial evaluators [37].

Copy propagation and constant folding reduce the test `(= x 0)` to `#f`. With `(and (read) (= x 0))` treated as `(if (read) (= x 0) #f)`, the preceding transformations reduce the entire expression to `(begin (read) e2)`.

4.3.4 Variable Assignments

If a local variable `x` is unreferenced, then an assignment `(set! x e)` is useless and `e` is processed in an `effect` context. Because the algorithm may still be in the process of computing referenced flags for residual-program variables when it encounters the assignment, it bases this decision on the conservative source-program referenced flags instead.[7] If `x` is referenced in the source program, the expression `e` is processed in a `value` context, the `residual-assigned` flag is set in the `prelex` for `x`, and an assignment to the renamed variable is residualized. In either case, `make-seq` is used to construct a residual sequence, if necessary, with a constant result expression. This convention avoids the need to check explicitly whether the residual expression is a `set!` expression when processing the guard of a conditional and when attempting constant propagation.

4.3.5 Procedure Calls

The treatment of `call` illustrates the demand-driven nature of the algorithm. To permit deferred processing, the argument at a call site is packaged with the current environment in an operand structure. The expression in operator position is then processed by `cp0-call`, defined in Figure 22. This procedure constructs an `app` context containing the operand and the context at the call site, then processes the operator expression in this new context.

Processing of arguments is deferred until the algorithm encounters references to the corresponding formal parameters (see Section 4.3.9) or decides to residualize the call. If a call is inlined, the residual code returned is a version of the integrated procedure's body specialized to its argument via recursive application of the inliner. In this case the operand is residualized for effect, if necessary, in a sequence with the specialized code. If the call has not been inlined, the residual code returned is a possibly simplified version of the operator. In this case the operand is processed for value using `value-visit-operand!` and the call is residualized.

The procedure `value-visit-operand!` returns the cached residual code for the argument if present. Otherwise it processes the argument in `value` context, caching the result. Caching residual

[7] In general, it is only safe to consult the `residual` flags of a `prelex` structure after all expressions in the scope of the corresponding variable have been processed.

```
(define cp0-call
  (lambda (e opnd ctxt env)
    (let* ([ctxt (make-app opnd ctxt)]
           [e (cp0 e ctxt env)])
      (if (app-inlined ctxt)
          (residualize-operand opnd e)
          `(call ,e ,(value-visit-operand! opnd))))))
(define residualize-operand
  (lambda (opnd e2)
    (if (not (operand-residualize-for-effect opnd))
        e2
        (let ([e1 (or (operand-value opnd)
                      (cp0 (operand-exp opnd) 'effect (operand-env opnd)))])
          (make-seq e1 e2)))))
(define value-visit-operand!
  (lambda (opnd)
    (or (operand-value opnd)
        (let ((e (cp0 (operand-exp opnd) 'value (operand-env opnd))))
          (set-operand-value! opnd e)
          e))))
```

Figure 22: Procedures for processing calls and operands.

code in the operand structure provides a form of path compression that prevents the algorithm from traversing a chain of bindings (see Section 4.3.9) more than once. Operands are visited in `value` context unless they are associated with unreferenced variables, in which case they are processed for effect by `residualize-operand` and there is no need to cache the result.

Processing on demand allows arguments bound to unreferenced formal parameters of an inlined procedure to be processed for effect rather than value. This often reduces the size and execution time of the residual program and may improve the accuracy of the information collected for variables in the residual program. For example, if `f` is `(lambda (x y z) (if (> z 0) (+ z y) x))`, the inlined call `(f e1 e2 3)` reduces to `(begin e1 ((lambda (y) (+ 3 y)) e2))`. Because `e1` is processed for effect, the residual code for the call may actually be smaller when inlined. If variable references in the source of `e1` do not appear in the residual code, expressions bound to those variables may be processed for effect as well.

```
(define fold-prim
  (lambda (p ctxt)
    (let ([opnd (app-opnd ctxt)])
      (let ([result
             (or (and (effect-free? p)
                      (context-case (app-ctxt ctxt)
                        [(effect) `(const ,(void))]
                        [(test) (and (result-true? p) `(const #t))]
                        [else #f]))
                 (and (foldable? p)
                      (let ([val (value-visit-operand! opnd)])
                        (exp-case (result-exp val)
                          [(const c) (on-error #f `(const ,(eval (list p c))))]
                          [else #f]))))])
        (if result
            (begin
              (set-operand-residualize-for-effect! opnd #t)
              (set-app-inlined! ctxt #t)
              result)
            `(primref ,p))))))
```

Figure 23: Procedure for constant folding.

4.3.6 Primitive References

The algorithm handles primitive references differently in different contexts.[8] In a call context, the algorithm attempts constant folding via the `fold-prim` procedure given in Figure 23. In a `value` context it simply residualizes the primitive reference. In `test` or `effect` contexts it residualizes the constant `#t` because a primitive reference always has a non-false value (primitives are bound to procedures) and has no side effect (the reference cannot signal an error as all primitives are bound).

Contextual information from the call site is used when attempting constant folding. When a primitive is known to be effect-free and the primitive call is processed in `effect` context, as indicated by the `app-ctxt` field, `fold-prim` residualizes the constant `(void)`. Examples of effect-free primitives include `cons`, `list`, `pair?`, and `not`. When an effect-free primitive call is processed in `test` context and the primitive is known to return a non-false value, `fold-prim` residualizes the constant `#t`. Effect-free primitives that always return non-false values include `cons` and `list`.

If contextual information and knowledge of the primitive do not justify constant folding but the primitive can be applied at compile time, then the operand is processed for value. If the operand's result expression is constant, the primitive is applied to the result expression and the constant result is residualized. The `on-error` macro is used to detect any errors that arise while applying the

[8] The `context-case` macro provides concise notation for selecting context-dependent action. Where more than one context specifier appears in the pattern on the left-hand side, as in the clause `[(test effect) '(const #t)]`, either context will match and select the action on the right-hand side.

primitive so that the primitive call can be residualized. This can occur if, for example, a divisor is zero. In *Chez Scheme* `on-error` is defined as follows.

```
(define-syntax on-error
  (syntax-rules ()
    ((_ err-value e)
     (call/1cc
       (lambda (k)
         (parameterize ((error-handler (lambda args (k err-value))))
           e))))))
```

If constant folding succeeds, the `inlined` flag of the context is set so that `cp0-call` does not residualize a call, and the `residualize-for-effect` flag of the operand is set so that `residualize-operand` will construct a sequence, if necessary, to preserve any side effects in the operand. If constant folding fails, the primitive reference is residualized for use by `cp0-call` as the operator in a residual call.

4.3.7 Lambda Expressions

When the algorithm encounters a `lambda` expression in an `app` context, it inlines the call using the `inline` procedure given in Figure 24. The body of the `lambda` expression is processed using the outer context from the call site, `(app-ctxt ctxt)`, in an environment mapping the formal parameter `x` to a fresh residual-program variable containing the operand from the call site.[9] The `source-referenced` and `source-assigned` flags of the fresh residual-program variable are initialized with the values of the `residual-referenced` and `residual-assigned` flags of the corresponding source-program variable.

For variables referenced or assigned in the residual code of the body, `make-let-binding` constructs a residual `let` binding (direct `lambda` call). A `let` binding is not constructed if the body has simplified to a constant, a primitive reference, or a reference to a free variable, or if no references or assignments to the bound variable were residualized while processing the body. Instead, a flag is set in the operand structure so that the operand will be processed for effect upon return to `cp0-call`, and the optimized body is returned. The algorithm also avoids constructing a `let` binding when the residual body is simply a reference to the bound variable. In this case the operand bound to the variable is processed for value and the result is residualized. When a `let` binding is constructed, the operand is processed for value only if references to the bound variable were residualized in the body. In particular, if a variable is assigned but not referenced in the residual code, the operand is processed for effect and the variable is bound to `(void)`. These transformations effectively inline a version of the procedure

[9] The macro `with-extended-env` is used here as a binding form in the object language and the meta language. In the object language, it extends the environment on the right-hand side with a binding from the old `x` to a fresh residual-program variable `x` containing the given operand `opnd`. In the meta language, it binds the new environment and residual-program variable to `env` and `x` within its body.

```
(define inline
  (lambda (proc ctxt env)
    (exp-case proc
      [(lambda (x) body)
       (let ([opnd (app-opnd ctxt)])
         (with-extended-env ((env x) (env x opnd))
           (let ([body (cp0 body (app-ctxt ctxt) env)])
             (let ([result (make-let-binding x opnd body)])
               (set-app-inlined! ctxt #t)
               result))))])))

(define make-let-binding
  (lambda (x opnd body)
    (let ([effect-opnd
           (lambda (result)
             (set-operand-residualize-for-effect! opnd #t)
             result)])
      (exp-case body
        [(const c) (effect-opnd body)]
        [(primref p) (effect-opnd body)]
        [(ref y) (if (eq? x y) (value-visit-operand! opnd) (effect-opnd body))]
        [else
         (cond
           [(prelex-residual-referenced? x)
            `(call (lambda (,x) ,body) ,(value-visit-operand! opnd))]
           [(prelex-residual-assigned? x)
            (effect-opnd `(call (lambda (,x) ,body) (const ,(void))))]
           [else (effect-opnd body)])]))))
```

Figure 24: Procedures for inlining and construction of residual `let` bindings.

specialized to the call site for which the call context was originally constructed.

In a `value` context `cp0` processes the body of a `lambda` expression in an environment extended with a fresh residual-program variable for the formal parameter and reconstructs a `lambda` expression binding the fresh variable in the optimized body. In `test` and `effect` contexts, a `lambda` expression reduces to the constant `#t` since a closure is non-false and creating a closure has no observable side effect.

Although the macro expander has already performed α-conversion, `cp0` must rename bound variables to prevent inadvertent capture when code is replicated during inlining. For example, consider the effect of inlining `f` within the body of `f` below.

```
(let ([f (lambda (f x y) (f (lambda (f a b) (+ x y a b)) (g 3) x))])
  (f f (g 1) (g 2)))
 →  (let ([x₁ (g 1)] [y (g 2)])
      (let ([x₂ (g 3)])
        (let ([a (g 3)])
          (+ x₁ y a x₂))))
```

The procedure `f` has been inlined once at the explicit call in the body of the original `let` and once where it is used as a procedural argument in the body of `f`. Where `f` is inlined, the variable `x` has been renamed as $\mathtt{x}_1$ and $\mathtt{x}_2$ to show that variable capture would otherwise occur in the expression (+ $\mathtt{x}_1$ y a $\mathtt{x}_2$).

4.3.8 Letrec Expressions

Handling a `letrec` expression is much like handling a `lambda` expression in a call context: the body is processed in an extended (circular) environment and useless bindings are pruned. The handling of `letrec` expressions is greatly simplified by the actions of the `cpletrec` pass which precedes `cp0` (see Section 4.2.6). In particular, because the core `letrec` form binds only `lambda` expressions, bindings for unreferenced variables of a `letrec` are effect-free and can simply be discarded. The algorithm also avoids constructing a residual `letrec` expression when the body is simply a constant or primitive reference.

4.3.9 Variable References

Variable references are the focal points for constant propagation, copy propagation, and inlining. To make the code more readable, `cp0` handles variable references by calling `cp0-ref`, shown in Figure 25. In `effect` context, `cp0-ref` replaces variable references with the constant `(void)` since the value is to be discarded and referencing a bound variable has no effect. In `value`, `test` and `app` contexts, `cp0-ref` obtains the corresponding residual-program variable from the environment and processes for value the operand, if any, bound to that variable as a prelude to propagation and inlining. The initial processing of the operand conveniently biases the algorithm in favor of inlining at the leaves of the call graph.

If the residual-program variable has no operand or the source variable was assigned, `cp0-ref` returns a reference to the residual variable. Otherwise the residual variable, its operand, and the context are passed to `copy` where the result expression of the operand's residual code is examined. When the result expression is constant, `copy` returns that constant to `cp0` where it will replace the variable reference. When the result expression is a reference to a variable `y` assigned in the source program, `copy` residualizes a reference to `x`. If `y` is unassigned but has no operand, `copy` residualizes a reference to `y`. If `y` is unassigned and has an operand, `copy2` is called to process `y` and its operand. These transformations accomplish copy propagation.

We say that the operand bound to a variable `x` is *copyable* if it is safe to replace references to `x`

```
(define cp0-ref
  (lambda (x ctxt env)
    (context-case ctxt
      [(effect) `(const ,(void))]
      [else (let ([x (lookup x env)])
              (let ([opnd (prelex-operand x)])
                (if opnd
                    (begin
                      (value-visit-operand! opnd)
                      (if (prelex-source-assigned? x)
                          (residualize-ref x)
                          (copy x opnd ctxt)))
                    (residualize-ref x))))])))

(define copy
  (lambda (x opnd ctxt)
    (let ((rhs (result-exp (operand-value opnd))))
      (exp-case rhs
        [(const c) rhs]
        [(ref y)
         (if (prelex-source-assigned? y)
             (residualize-ref x)
             (let ([opnd (prelex-operand y)])
               (if (and opnd (operand-value opnd))
                   (copy2 y opnd ctxt)
                   (residualize-ref y))))]
        [else (copy2 x opnd ctxt)]))))

(define copy2
  (lambda (x opnd ctxt)
    (let ((rhs (result-exp (operand-value opnd))))
      (exp-case rhs
        [(lambda (z) body)
         (context-case ctxt
           [(value) (residualize-ref x)]
           [(test) `(const #t)]
           [(app) (inline rhs ctxt empty-env)])]
        [(primref p)
         (context-case ctxt
           [(value) rhs]
           [(test) `(const #t)]
           [(app) (fold-prim p ctxt)])]
        [else (residualize-ref x)]))))

(define residualize-ref
  (lambda (id)
    (set-prelex-residual-referenced! id #t)
    `(ref ,id)))
```

Figure 25: Procedures for processing variable references.

with the `result-exp` of the residual code for that operand. In general, an operand is copyable if the variable `x` is not assigned and the `result-exp` of the operand is a constant, a `primref`, or a reference to an unassigned variable `y`. When the operand of `y` is also copyable, it is safe to replace references to both `x` and `y` with the result-exp of `y`'s operand.

Suppose a program contains a chain of unassigned variable bindings in which x_i is bound to x_{i-1} for $1 \leq i \leq n$, as shown below.

```
(let ((x0 e))
  (let ((x1 x0))
    (let ((x2 x1))
      ...
       (let ((xn xn-1))
         xn))))
```

A simple inductive argument on the length of the chain shows that the algorithm propagates the outermost copyable entity, if any, in the chain of references to every variable later in the chain.

In the base case, the length of the chain is zero and we have the following.

```
(let ((x0 e))
  x0)
```

When processing the reference to x_0, `cp0-ref` calls `value-visit-operand!` since x_0 has an operand. Because x_0 is unassigned, `cp0-ref` calls `copy` after visiting the operand for value. If the operand is not copyable, the claim is vacuously true. Suppose instead the operand is copyable. If the `result-exp` of the operand is a constant, then `copy` returns the constant to `cp0-ref` where it replaces the reference to x_0. If the `result-exp` of the operand is a `primref`, then `copy` invokes `copy2` and, since the context is `value`, `copy2` returns the `primref` to `cp0-ref` where it replaces the reference to x_0. If the `result-exp` of the operand is a reference to an unassigned variable `y`, then `y`'s operand, if any, must not be copyable since we are given that the reference chain has length zero. If `y` has no operand, then `copy` residualizes a reference to `y`. If `y`'s operand is not copyable, then `copy2` residualizes a reference to `y`. In any case, the copyable operand of x_0, if any, replaces the reference to x_0.

Now assume the claim is true for chains of length k and consider a chain of length $k+1$.

```
(let ((x0 e))
  (let ((x1 x0))
    (let ((x2 x1))
      ...
       (let ((xk xk-1))
         (let ((xk+1 xk))
           xk+1)))))
```

When processing the reference to x_{k+1}, `cp0-ref` calls `value-visit-operand!` since x_{k+1} is unassigned and has an operand. The operand of x_{k+1} is a reference to x_k, so `value-visit-operand!` calls `cp0` recursively to process that reference. Since x_k ends an unassigned variable chain of length k, we know by the inductive hypothesis that `cp0` replaces the reference to $\mathtt{x}_k$ with the outermost copyable entity, if any, in the chain. If the reference to $\mathtt{x}_k$ was replaced, then the operand of $\mathtt{x}_{k+1}$ is copyable and, by a case analysis like that for $\mathtt{x}_0$, we know that references to $\mathtt{x}_{k+1}$ will be replaced by the `result-exp` of the operand. If the reference to $\mathtt{x}_k$ was not replaced, then either `copy` or `copy2` will residualize a reference to $\mathtt{x}_k$ in place of $\mathtt{x}_{k+1}$ since the operand of $\mathtt{x}_{k+1}$ is a ref to an unassigned variable,

The `copy2` procedure uses contextual information to select an appropriate transformation. When the result expression of the operand is a `lambda` expression and the context is `value`, `copy2` residualizes a reference to the variable since `lambda` expressions are not copyable in general (see Section 4.2.5). In `test` context, `copy2` residualizes the constant `#t` since a `lambda` expression evaluates to a closure and closures are non-false. In an `app` context, the procedure bound to the variable is inlined.

When the result expression of the operand is a `primref` expression and the context is `value`, `copy2` residualizes the primitive reference, effectively treating primitives as constants to be propagated. In `test` context, `copy2` residualizes the constant `#t` since primitives are bound to closures and closures are non-false. When processing a `primref` in an `app` context, `copy2` calls `fold-prim` to attempt constant folding.

If the variable is not bound to a `lambda` or `primref` expression, `copy2` residualizes a reference to the variable. Variable references are constructed by the `residualize-ref` procedure, shown in Figure 25, which sets the `residual-referenced` flag of the given `prelex` structure. This flag is examined by `inline` and the `letrec` case of `cp0` to determine whether or not a binding is useless.

4.4 Restraining the Algorithm

To be effective, an inlining algorithm must recognize opportunities for inlining and select appropriately among them. The unrestrained algorithm described in Section 4.3 concentrates on the first of these tasks in its single-minded pursuit of inlining opportunities. In fact, the algorithm is so intent on inlining that it may not terminate. This section addresses the second task, seeking to prevent the costs of inlining from outweighing the benefits. As indicated in Section 4.2.5, the costs of concern here are increased or indefinite processing time and increased code size.

A linear bound is enforced on the run time of the algorithm by maintaining an effort counter whenever an expression is processed more than once, *i.e.*, for each attempt to inline an operand. The effort counter advances each time `cp0` is called.[10] If the effort bound is exceeded, the inlining attempt

[10] For efficiency, counters start at the threshold value and "advance" toward zero. This avoids the need for separate

```
(define cp0
  (lambda (exp ctxt env ec sc)
    (decrement ec 1)
    (exp-case exp
      [(const c) exp]
      [(ref x) (cp0-ref x ctxt env ec sc)]
      [(begin e1 e2)
       (let ([e1 (cp0 e1 'effect env ec sc)] [e2 (cp0 e2 ctxt env ec sc)])
         (make-seq e1 e2))]
      [(if e1 e2 e3)
       (let ([e1 (cp0 e1 'test env ec sc)])
         (exp-case (result-exp e1)
           [(const c) (make-seq e1 (cp0 (if c e2 e3) ctxt env ec sc))]
           [else (let ([ctxt (if (app? ctxt) 'value ctxt)])
                   (let ([e2 (cp0 e2 ctxt env ec sc)] [e3 (cp0 e3 ctxt env ec sc)])
                     (if (record-equal? e2 e3 ctxt)
                         (make-seq e1 e2)
                         (begin (decrement sc 1) `(if ,e1 ,e2 ,e3)))))]))]
      [(set! x e)
       (let ([x (lookup x env)])
         (make-seq
           (if (not (prelex-source-referenced? x))
               (cp0 e 'effect env ec sc)
               (let ([e (cp0 e 'value env ec sc)])
                 (decrement sc 1)
                 (set-prelex-residual-assigned! x #t)
                 `(set! ,x ,e)))
           `(const ,(void))))]
      [(call e1 e2) (cp0-call e1 (make-operand e2 env ec) ctxt env ec sc)]
      [(primref p)
       (context-case ctxt
         [(app) (fold-prim p ctxt sc)]
         [(value) (decrement sc 1) exp]
         [(test effect) `(const #t)])]
      [(lambda (x) e)
       (context-case ctxt
         [(app) (inline exp ctxt env ec sc)]
         [(value)
          (decrement sc 1)
          (with-extended-env ((env x) (env x #f))
            `(lambda (,x) ,(cp0 e 'value env ec sc)))]
         [(test effect) `(const #t)])]
      [(letrec ((x e1)) e2)
       (with-extended-env ((env x) (env x #f))
         (let ([opnd (make-operand e1 env ec)])
           (set-prelex-operand! x opnd)
           (let ([e2 (cp0 e2 ctxt env ec sc)])
             (if (or (not (prelex-residual-referenced? x))
                     (exp-case e2 [(const c) #t] [(primref p) #t] [else #f]))
                 e2
                 (begin (decrement sc (+ 1 (operand-size opnd)))
                        `(letrec ((,x ,(operand-value opnd))) ,e2))))))])))
```

Figure 26: The restrained inlining algorithm.

is aborted (via nonlocal exit) and the call is residualized instead. Because the inlining attempt may be aborted, `inline` and `fold-prim` must not set the inlined flag until processing is complete. The effort counter is not reset for nested integrations, as this could lead to nonlinear processing time. Because the number of call sites in the original source program is fixed and because we bound the amount of effort expended processing each of these sites, the run time of the algorithm is linear in the size of the input program.

Code growth is limited via a size counter that tracks the size of the residual code as it is produced. This counter advances whenever a form is residualized in the output. If the size threshold is exceeded while attempting to integrate a procedure, the inlining attempt is abandoned and the call is residualized instead. Polyvariance is achieved by attempting to integrate a procedure at each of its call sites while relying on the size counter to abort any attempts for which the residual code produced would exceed the size limit. Section 4.5 describes an extension that imposes no limit on the size of the residual code while inlining an operand bound to a variable referenced only once in the source program. Inlining procedures called only once does not directly increase the size of the residual program and, through specialization, may decrease code size.

The effort and size counters are intuitive, no-nonsense mechanisms that directly regulate processes contributing to the costs that concern us. These counters are passed as additional arguments to the restrained `cp0`, as shown in Figure 26. The effort counter, `ec`, is decremented on entry to `cp0`. The size counter, `sc`, is decremented when a form is residualized in the output. At present, the algorithm does not decrement the size counter for residual `const` or `begin` forms. Intuitively, the `begin` form exists only within the intermediate representation of the program and does not increase code size; its subexpressions will be concatenated in the instruction stream. The decision not to count constants is arbitrary.

Counters are represented by structures defined in Figure 27. A counter is either active or passive. Active counters are used to limit code growth and processing time during inlining attempts. The `ctxt` field of an active counter contains the `app` context of the call whose inlining attempt it governs. The `k` field of an active counter holds a continuation that awaits the inlined procedure body. An inlining attempt is aborted by returning `#f` to this continuation. Passive counters measure resource consumption but exert no control. The `ctxt` field of a passive counter is `#f`; the `k` field contains an error continuation. The compiler calls `cp0` with passive effort and size counters initially; `cp0` creates active counters for inlining attempts.

When processing a call, `cp0` stores the current effort counter in the operand structure so that

fields to record the current value and the threshold.

```
(define-structure (counter value ctxt k))

(define passive-counter
  (lambda ()
    (make-counter (most-positive-fixnum) #f
      (lambda x (error #f "attempt to abort from passive counter")))))

(define passive-counter-value
  (lambda (counter)
    (- (most-positive-fixnum) (counter-value counter))))

(define active-counter?
  (lambda (x)
    (and (counter? x) (counter-ctxt x))))

(define decrement
  (lambda (counter amount)
    (let ([n (- (counter-value counter) amount)])
      (set-counter-value! counter n)
      (when (< n 0)
        (reset-integrated! (counter-ctxt counter))
        ((counter-k counter) #f)))))

(define reset-integrated!
  (lambda (ctxt)
    (set-app-inlined! ctxt #f)
    (let ((ctxt (app-ctxt ctxt)))
      (when (app? ctxt) (reset-integrated! ctxt)))))
```

Figure 27: Procedures for creating and manipulating counters.

the cost of processing the operand is charged to the original call site. In particular, this allows source-program operands to be processed with a passive effort counter. The current size counter is not stored in the operand structure. When the operand is first processed for value, a passive size counter is used instead. The resulting size count is cached along with the residual code in the operand structure. The cached size is added to the current size counter if a binding to this operand is residualized. To support these changes, operand structures and the procedures that manipulate them are extended as shown in Figure 28.

Cycles are detected by consulting flags in operand structures to recognize when the value of an operand is requested during the initial processing or inlining of that operand. The `inner-pending` flag, tested and set by `cp0-ref`, is used to detect recursive references within an operand as in `(letrec ((f (lambda () (f)))) (f))`. The `outer-pending` flag, tested and set by `copy2`, is used to detect recursive references encountered while inlining a procedural operand, as in the expression `((lambda (x) (x x)) (lambda (x) (x x)))`. These flags can be extended to govern loop unrolling within a recursive procedure and unfolding at the call site. If a cycle is discovered while processing the operand for a variable, a variable reference is residualized. The use of the `inner-pending` and

```
(define-structure (operand exp env ec)
  ([value #f]
   [residualize-for-effect #f]
   [size 0]
   [inner-pending #f]
   [outer-pending #f]))

(define value-visit-operand!
  (lambda (opnd)
    (or (operand-value opnd)
        (let* ([sc (passive-counter)]
               [e (with-fields opnd (operand exp env ec)
                    (cp0 exp 'value env ec sc))])
          (set-operand-value! opnd e)
          (set-operand-size! opnd (passive-counter-value sc))
          e))))

(define score-value-visit-operand!
  (lambda (opnd sc)
    (let ((val (value-visit-operand! opnd)))
      (decrement sc (operand-size opnd))
      val)))

(define residualize-operand
  (lambda (opnd e2 sc)
    (if (not (operand-residualize-for-effect opnd))
        e2
        (let ([e1 (or (operand-value opnd)
                      (with-fields opnd (operand exp env ec)
                        (cp0 exp 'effect env ec sc)))])
          (if (simple? e1)
              e2
              (begin (decrement sc (operand-size opnd))
                     (make-seq e1 e2)))))))
```

Figure 28: Procedures for creating and manipulating operands.

`outer-pending` flags of operands represents a natural graph-marking approach to cycle detection.

Most of the changes in the restrained algorithm are confined to `cp0-ref`, shown in Figure 29, and `copy` and `copy2`, shown in Figure 30. The changes to `copy` are simply to accept and pass along the additional effort counter and size counter arguments. When the context is not `effect` and the residual variable has an operand, `cp0-ref` now tests the `inner-pending` flag of the operand before proceeding. If the flag is clear, `cp0-ref` uses `dynamic-wind` [27] to set the flag before processing the operand and to clear the flag when processing is complete or when processing is aborted via non-local exit. If the flag is set, `cp0-ref` must have been called upon to process a reference to the variable during the initial processing of its operand. For example, consider the operation of `cp0` on the following program.

```
(define cp0-ref
  (lambda (x ctxt env ec sc)
    (context-case ctxt
      [(effect) `(const ,(void))]
      [else (let ([x (lookup x env)])
              (let ([opnd (prelex-operand x)])
                (if (and opnd (not (operand-inner-pending opnd)))
                    (begin
                      (dynamic-wind
                        (lambda () (set-operand-inner-pending! opnd #t))
                        (lambda () (value-visit-operand! opnd))
                        (lambda () (set-operand-inner-pending! opnd #f)))
                      (if (prelex-source-assigned? x)
                          (residualize-ref x sc)
                          (copy x opnd ctxt ec sc)))
                    (residualize-ref x sc))))])))
(define residualize-ref
  (lambda (id sc)
    (decrement sc 1)
    (set-prelex-residual-referenced! id #t)
    `(ref ,id)))
```

Figure 29: Procedures for processing variable references with restraint.

```
(lambda (n)
  (letrec ([f (lambda (x) (if (zero? x) 1 (* x (f (- x 1)))))])
    (f n)))
```

The operand `(lambda (x) ...)` is not processed until `cp0` encounters the reference to `f` in the call `(f n)`. At that point the `inner-pending` flag is clear, so `cp0-ref` sets the flag via `dynamic-wind` before calling `value-visit-operand!`. When `cp0` processes the operand, `cp0-ref` is called to process the reference to `f` in the call `(f (- x 1))`. At this point the `inner-pending` flag on `f`'s operand is set, indicating a cycle, so a variable reference is residualized. This prevents indefinite recursion that would otherwise result if `value-visit-operand!` were called again on this operand.

The major change in `copy2` is in the handling of `lambda` expressions in `app` contexts. First `copy2` checks whether the `outer-pending` flag of the operand is set. If so, a cycle has been detected and no inlining is performed. Instead, a variable reference is residualized as the operator of the call. If the `outer-pending` flag is clear, `copy2` sets the flag and attempts to inline the procedure. To attempt inlining, `copy2` obtains a continuation that includes the winders installed by `dynamic-wind` and stores this continuation in active size and effort counters. If the current size counter is active, its value is used as the limit for the newly constructed size counter so that inlining attempts are scored more fairly.[11] Otherwise the limit is set to the value of the `cp0-size-limit` parameter. To

[11] This unfortunately makes the outcome sensitive to the order in which `cp0` processes subexpressions.

```
(define copy
  (lambda (x opnd ctxt ec sc)
    (let ((rhs (result-exp (operand-value opnd))))
      (exp-case rhs
        [(const c) rhs]
        [(ref y)
         (if (prelex-source-assigned? y)
             (residualize-ref x sc)
             (let ([opnd (prelex-operand y)])
               (if (and opnd (operand-value opnd))
                   (copy2 y opnd ctxt ec sc)
                   (residualize-ref y sc))))]
        [else (copy2 x opnd ctxt ec sc)]))))

(define copy2
  (lambda (x opnd ctxt ec sc)
    (let ((rhs (result-exp (operand-value opnd))))
      (exp-case rhs
        [(lambda (z) body)
         (context-case ctxt
           [(value) (residualize-ref x sc)]
           [(test) `(const #t)]
           [(app)
            (or (and (not (operand-outer-pending opnd))
                     (dynamic-wind
                       (lambda () (set-operand-outer-pending! opnd #t))
                       (lambda ()
                         (call/1cc
                           (lambda (abort)
                             (let ([limit
                                     (if (active-counter? sc)
                                         (counter-value sc)
                                         (cp0-size-limit))])
                               (inline rhs ctxt empty-env
                                 (if (active-counter? ec)
                                     ec
                                     (make-counter (cp0-effort-limit) ctxt abort))
                                 (make-counter limit ctxt abort))))))
                       (lambda () (set-operand-outer-pending! opnd #f))))
                (residualize-ref x sc))])]
        [(primref p)
         (context-case ctxt
           [(value) rhs]
           [(test) `(const #t)]
           [(app) (fold-prim p ctxt sc)])]
        [else (residualize-ref x sc)]))))
```

Figure 30: Procedures for copy propagation with restraint.

```
(define fold-prim
  (lambda (p ctxt sc)
    (let ([opnd (app-opnd ctxt)])
      (let ([result
              (or (and (effect-free? p)
                       (context-case (app-ctxt ctxt)
                         [(effect) `(const ,(void))]
                         [(test) (and (result-true? p) `(const #t))]
                         [else #f]))
                  (and (foldable? p)
                       (let ([val (value-visit-operand! opnd)])
                         (exp-case (result-exp val)
                           [(const c) (on-error #f `(const ,(eval (list p c))))]
                           [else #f]))))])
        (if result
            (begin
              (set-operand-residualize-for-effect! opnd #t)
              (set-app-inlined! ctxt #t)
              result)
            (begin
              (decrement sc 1)
              `(primref ,p)))))))
```

Figure 31: Procedure for constant folding with restraint.

ensure linearity, a new effort counter is constructed only if the current effort counter is passive. The active counters are used in the call to `inline`. If the counters do not expire, the procedure has been inlined. If one of the counters reaches zero, the `decrement` procedure invokes the continuation saved in that counter to return `#f` as the value of the `and` expression nested within the `or` in `copy2`. This causes a variable reference to be residualized as the operator of the call. If the size limit is exceeded, the current inlining attempt is the one aborted. If the effort limit is exceeded, inlining is aborted back to the point where the effort counter first became active. This may include nested inlining attempts. When an inlining attempt is aborted, the `inlined` flag of the `app` context stored in the expired counter is reset along with the `inlined` flags of all enclosing `app` contexts. This is necessary because a successful attempt may yet fail when `residualize-operand` processes the operand.

The minor change in `copy2` is that the current size counter is now passed to `fold-prim` when attempting constant folding. The `fold-prim` procedure, shown in Figure 31, now decrements the size counter if the primitive must be residualized. Similar changes are made to `cp0-call`, `inline`, and `make-let-binding`, as shown in Figure 32. When `cp0-call` and `make-let-binding` construct residual calls, `score-value-visit-operand!` decrements the current size counter by the amount cached in the operand. When `residualize-operand` (see Figure 28) must residualize an operand for effect, the size counter is decremented by the cached operand size only if the residual operand is not simple.

This prevents it from charging the size of a potentially large expression against the counter only to have `make-seq` drop the expression. Over-counting may still occur, however, if the residual operand is a sequence with a simple result-expression, since the size of that expression is included in the cached operand size.

One interesting aspect of the demand-driven, online algorithm is the tight coupling of analysis and transformation in which transformation is the rate-limiting process. In an offline algorithm, analysis precedes transformation. An offline analysis typically attempts to determine as much as possible so that the subsequent transformation phase is not hampered by lack of information. Of course, the analysis must be abstract in order to terminate and, being unaware of the particular needs of the transformation phase, may choose to abstract arbitrarily at a point where transformation might have been possible had it selected another point. In contrast, transformation drives analysis in the online approach. The analysis determines only what is necessary to permit transformation. This improves both the efficiency and the outcome of the entire process: the analysis wastes less time discovering facts that ultimately have no application, and the transformation process is less often hampered by arbitrary abstraction decisions because decisions are now made by the mechanisms regulating transformation.[12]

The effort counter alone is sufficient to ensure termination and to enforce a linear bound on processing time and code growth. One may ask why three regulatory mechanisms are used. The size counter prevents excessive code growth in any one part of the program. Generally the size threshold is set well below the effort threshold. The size counter then prevents any one part of the program from using an unfair share of the effort budget, so that all parts of the program may benefit from inlining. The cycle-detection mechanism generally interrupts recursive inlining long before the effort counter is exceeded. This is important because it saves processing time and because it prevents the algorithm from wasting effort that could otherwise be applied to inlining. To illustrate, suppose that while inlining a call to `f`, the algorithm encounters a call to `g`, where `g` is bound to a recursive procedure. If cycle-detection were disabled, then recursively visiting the operand of `g` would eventually exceed the effort limit and the entire inlining attempt for `f` would be aborted. By detecting the cycle involving `g` early, the algorithm residualizes the call to `g` and resumes its effort to inline `f`.

Because it ensures linearity, the effort limit can cause `cp0` to miss inlining opportunities. For example, `cp0` may expend a great deal of effort processing one branch of a conditional expression only to detect a cycle immediately upon processing the other branch. A subsequent inlining attempt

[12] While these decisions can be arbitrary, they are also arbitrary in the offline case. The online method has one less source of arbitrariness.

```
(define cp0-call
  (lambda (e opnd ctxt env ec sc)
    (let* ([ctxt (make-app opnd ctxt)]
           [e (cp0 e ctxt env ec sc)])
      (if (app-inlined ctxt)
          (residualize-operand opnd e sc)
          `(call ,e ,(score-value-visit-operand! opnd sc))))))

(define inline
  (lambda (proc ctxt env ec sc)
    (exp-case proc
      [(lambda (x) body)
       (let ([opnd (app-opnd ctxt)])
         (with-extended-env ((env x) (env x opnd))
           (let ([body (cp0 body (app-ctxt ctxt) env ec sc)])
             (let ([result (make-let-binding x opnd body sc)])
               (set-app-inlined! ctxt #t)
               result))))])))

(define make-let-binding
  (lambda (x opnd body sc)
    (let ([effect-opnd
           (lambda (result)
             (set-operand-residualize-for-effect! opnd #t)
             result)])
      (exp-case body
        [(const c) (effect-opnd body)]
        [(primref p) (effect-opnd body)]
        [(ref y)
         (if (eq? x y) (score-value-visit-operand! opnd sc) (effect-opnd body))]
        [else
         (cond
           [(prelex-residual-referenced? x)
            `(call (lambda (,x) ,body) ,(score-value-visit-operand! opnd sc))]
           [(prelex-residual-assigned? x)
            (effect-opnd `(call (lambda (,x) ,body) (const ,(void))))]
           [else (effect-opnd body)])]))))
```

Figure 32: Procedures for constant folding and inlining with restraint.

that should otherwise succeed may then fail when the effort counter is decremented. Using only the size counter and the cycle detection mechanisms might therefore yield more optimal code, though at the cost of a nonlinear algorithm.

4.5 Inlining Singly Referenced Procedures

When a procedure reaches only one call site, inlining that call does not directly increase code size and may, in fact, reduce code size via specialization. A procedure can certainly reach only one call site if it is bound to an unassigned variable referenced only in the operator position at that call site. Because code growth is not a concern when inlining these procedures, inlining can be far more aggressive. Although intuitively appealing, inlining these singly referenced procedures is not trivial.

One key difficulty is that a variable singly referenced in the source program may in fact be multiply referenced in the residual program. This can occur in two ways. First, the singly referenced variable may be the value bound to a variable referenced more than once, as in the source program on the left below.

```
(let ((x1 e))
  (let ((x2 x1))
    ...
     (let ((xn xn-1))   →   (let ((x1 e))
       (let ((y xn))          (x1 x1))
         (y y)))))
```

Although each $\mathtt{x}_i$, $1 \leq i \leq n$, is referenced just once, `y` appears twice. From Section 4.3.9, we know this program residualizes to the code on the right, in which $\mathtt{x}_1$ is referenced twice. Second, a variable can become multiply referenced in the residual program when it is a free variable of a procedure inlined in several places. For example, inlining `f` wherever it appears in the following source program produces the residual code on the right with `x` now multiply referenced.

```
(let ((x e))
  (let ((f (lambda () x)))   →   (let ((x e))
    ((f) (f))))                    (x x))
```

Another difficulty is that a variable referenced more than once in the source program may appear only once in the residual program. This can occur when variable references appear within useless or unreachable code that is eliminated. Because `cp0` is still in the process of computing the residual code, accurate single-reference information is not available when it could be exploited, and `cp0` must use conservative source-program information instead.[13]

[13] Recall that information about residual program variables is used only when pruning useless `let` and `letrec` bindings (in `make-let-binding` and the `letrec` case of `cp0`). This is possible because bindings are pruned after the residual code for the body of the binding form has been constructed.

```
(define-structure (prelex name operand)
  ([source-referenced? #f]
   [source-assigned? #f]
   [source-singly-referenced? #t]
   [residual-referenced? #f]
   [residual-assigned? #f]
   [residual-singly-referenced? #t]))

(define residualize-ref
  (lambda (id sc)
    (decrement sc 1)
    (when (prelex-residual-referenced? id)
      (set-prelex-residual-singly-referenced! id #f))
    (set-prelex-residual-referenced! id #t)
    `(ref ,id)))
```

Figure 33: Modifications to record and maintain single-reference information for variables.

```
(define cp0-ref
  (lambda (x ctxt env ec sc)
    (context-case ctxt
      [(effect) `(const ,(void))]
      [else (let ([new-x (lookup x env)])
              (when (eq? new-x x) (set-prelex-source-singly-referenced! x #f))
              (let ([opnd (prelex-operand new-x)])
                (if (and opnd (not (operand-inner-pending opnd)))
                    (or (cp0-single-ref new-x opnd ctxt env ec sc)
                        (begin
                          (dynamic-wind
                            (lambda () (set-operand-inner-pending! opnd #t))
                            (lambda () (value-visit-operand! opnd))
                            (lambda () (set-operand-inner-pending! opnd #f)))
                          (if (prelex-source-assigned? new-x)
                              (residualize-ref new-x sc)
                              (copy new-x opnd ctxt ec sc))))
                    (residualize-ref new-x sc))))])))

(define copy
  (lambda (x opnd ctxt ec sc)
    (let ((rhs (result-exp (operand-value opnd))))
      (exp-case rhs
        [(const c) rhs]
        [(ref y)
         (if (prelex-source-assigned? y)
             (residualize-ref x sc)
             (let ([opnd (prelex-operand y)])
               (unless (prelex-source-singly-referenced? x)
                 (set-prelex-source-singly-referenced! y #f))
               (if (and opnd (operand-value opnd))
                   (copy2 y opnd ctxt ec sc)
                   (residualize-ref y sc))))]
        [else (copy2 x opnd ctxt ec sc)]))))
```

Figure 34: Single reference information is handled conservatively.

```
(define copy2
  (lambda (x opnd ctxt ec sc)
    (let ((rhs (result-exp (operand-value opnd))))
      (exp-case rhs
        [(lambda (z) body)
         (context-case ctxt
           [(value) (residualize-ref x sc)]
           [(test) `(const #t)]
           [(app)
            (or (and (not (operand-outer-pending opnd))
                     (dynamic-wind
                       (lambda () (set-operand-outer-pending! opnd #t))
                       (lambda ()
                         (call/1cc
                           (lambda (abort)
                             (let ([limit
                                    (if (active-counter? sc)
                                        (counter-value sc)
                                        (cp0-size-limit))])
                               (inline rhs ctxt empty-env
                                 (if (active-counter? ec)
                                     ec
                                     (make-counter (cp0-effort-limit) ctxt abort))
                                 (if (prelex-source-singly-referenced? x)
                                     (passive-counter)
                                     (make-counter limit ctxt abort)))))))
                       (lambda () (set-operand-outer-pending! opnd #f))))
                (residualize-ref x sc))])]
        [(primref p)
         (context-case ctxt
           [(value) rhs]
           [(test) `(const #t)]
           [(app) (fold-prim p ctxt sc)])]
        [else (residualize-ref x sc)]))))
```

Figure 35: Copy propagation exploits single reference information for inlining.

Of the two problems, the first is more serious. Naively treating as singly referenced a variable that is in fact surreptitiously referenced more than once can lead to excessive code growth and excessive processing time. On the other hand, conservatively treating as multiply referenced some variables that are singly referenced in the residual code simply means that some opportunity for optimization is missed. The extension described in this section is careful to avoid the former problem but not the latter.

Single-reference information is recorded by extending `prelex` structures with source and residual `singly-referenced` flags initialized by the macro expander and maintained by `cp0`. When residualizing a reference to a variable that is already marked as referenced, the single-reference flag is

```
(define cp0-single-ref
  (lambda (x opnd ctxt env ec sc)
    (and (app? ctxt)
         (let loop ((x x) (opnd opnd))
           (and (prelex-source-singly-referenced? x)
                (not (prelex-source-assigned? x))
                (exp-case (operand-exp opnd)
                  [(lambda (z) body)
                   (let ((sc (passive-counter)))
                     (with-fields opnd (operand exp env ec)
                       (let ((result (inline exp ctxt env ec sc)))
                         (set-operand-single-reference-size! opnd
                           (passive-counter-value sc))
                         result)))]
                  [(ref z)
                   (let ((new-z (lookup z (operand-env opnd))))
                     (and (not (eq? z new-z))
                          (loop new-z (prelex-operand new-z))))]
                  [else #f]))))))

(define-structure (operand exp env ec)
  ([value #f]
   [residualize-for-effect #f]
   [size 0]
   [single-reference-size 0]
   [inner-pending #f]
   [outer-pending #f]))
```

Figure 36: Procedures that exploit single reference information for inlining.

reset. These changes are shown in Figure 33. At this point it is natural to ask why not instead maintain actual reference counts for variables. The main reason is that it is both complicated and costly to maintain accurate reference counts due to the extensive use of non-local exits. We outline mechanisms for doing so at the conclusion of this section.

The `copy` procedure now clears the `singly-referenced` flag of any variable reached via copy propagation through a variable that is multiply referenced. The `cp0-ref` procedure conservatively clears the `singly-referenced` flag of any free variable it encounters while integrating. Free variables are easily recognized because they are not renamed by `lookup`. These changes, shown in Figure 34, prevent the algorithm from naively treating as singly referenced a variable that is referenced more than once due to copy propagation or inlining.

Single reference information is exploited by `copy2`, shown in Figure 35, and the new procedure `cp0-single-ref`, shown in Figure 36. When attempting to inline a procedure bound to a singly referenced variable, `copy2` now uses a passive size counter. While this permits the algorithm to inline procedures larger than the size limit, the size of procedures inlined by `copy2` is still constrained by

```
(define residualize-operand
  (lambda (opnd e2 sc)
    (if (not (operand-residualize-for-effect opnd))
        e2
        (or (and (operand-single-reference-size opnd)
                 (or (not (operand-value opnd))
                     (simple? (operand-value opnd)))
                 (begin
                   (decrement sc (operand-single-reference-size opnd))
                   e2))
            (let ([e1 (or (operand-value opnd)
                          (with-fields opnd (operand exp env ec)
                            (cp0 exp 'effect env ec sc)))])
              (if (simple? e1)
                  e2
                  (begin
                    (decrement sc (operand-size opnd))
                    (make-seq e1 e2))))))))
```

Figure 37: Single reference size must be used when residualizing operands.

the effort limit. To overcome this limitation, `cp0-single-ref` first examines the operand expression for a singly referenced variable and attempts inlining with a passive size counter and the effort counter stored in the operand if it finds a `lambda` expression. Because source-program operands contain the passive effort counter initially passed to `cp0`, `cp0-single-ref` can fully inline `lambda` expressions bound to singly referenced source-program variables even if doing so would exceed the effort limit. When searching for a `lambda` expression, `cp0-single-ref` follows chains of unassigned singly referenced local bindings. Copy propagation is performed explicitly here because the operand has not yet been visited for value. In fact, the operand is visited for value and `copy2` is called only if `cp0-single-ref` does not attempt inlining.

A passive size counter is used while inlining singly referenced procedures. If inlining succeeds, the resulting size must be charged back to the call site that originally constructed the singly referenced operand. Thus `residualize-operand` is modified in Figure 37 to use the size computed during single-reference inlining if there is no operand value (indicating that the operand was inlined by `cp0-single-ref`) or the operand value is simple (indicating that `copy2` inlined the operand). Once again overscoring is possible if the inlined operand is visited for value and the resulting value is not simple.

Because `cp0-single-ref` does not visit operands for value, this more aggressive treatment of singly referenced procedures benefits only explicit `lambda` expressions. Visiting an operand for value may expose a `lambda` expression disguised as a sequence or a `let` or `letrec` binding in the source program. It may therefore appear that more general results might be obtained by using the effort counter of the operand when inlining a singly referenced procedure in `copy2`. Unfortunately, this would exaggerate the cost of processing the operand since the effort counter in the operand is also used when visiting for value.

In fact, by not visiting the operand for value, `cp0-single-ref` avoids two sources of inaccuracy. First, a variable can be marked as referenced within the residual program even though it is referenced only within an operand and that operand is subsequently inlined. Second, a variable can be marked as multiply referenced in similar circumstances. To illustrate, consider the following program.

```
(let ([a (lambda (n) (display n) (1+ n))])
  (let ([b a])
    (let ([c b])
      (let ([d c])
        (d 1)))))
```

Because the variables `a`, `b`, `c`, and `d` are singly referenced and `d` is referenced as the operator in a call, `cp0-single-ref` will traverse the binding chain, find the `lambda` expression, and inline the call, ultimately producing `(begin (display 1) 2)`. Without the single reference extension, `cp0-ref` would first visit `d`'s operand `c` for value. This would cause `c`'s operand to be visited for value. Eventually, `b`'s operand would be visited for value and `copy2`, unable to propagate a `lambda` expression in `value` context, would residualize a reference to `a`, causing `a` to appear referenced in the residual program. Having recursively computed a value for `b`'s operand, the algorithm would return to compute a value for `c`'s operand. Since `b`'s operand is now a reference to `a`, `copy2` would be called once again to propagate `a` only to residualize another reference, causing `a` to now appear multiply referenced. Because `a` appears referenced in the residual program, `cp0` would construct a useless `let` binding, as shown below, if the single reference extension were disabled.

```
(let ([a (lambda (n) (display n) (1+ n))])
  (begin (display 1) 2))
```

Despite its limitations, `cp0-single-ref`'s aggressive inlining of singly referenced procedures can be quite useful. For example, if the restrained inlining algorithm were implemented in continuation-passing style (CPS) it might resemble the following.

```
(define cp0
  (lambda (exp ctxt env ec sc k)
    (decrement ec 1
      (lambda ()
        (exp-case exp
          [(const c) (k exp)]
          [(begin e1 e2)
           (cp0 e1 'effect env ec sc
             (lambda (e1)
```

```
                (cp0 e2 ctxt env ec sc
                  (lambda (e2) (k (make-seq e1 e2))))))]
           ...)))))
(define decrement
  (lambda (counter amount k)
    (let ([n (- (counter-value counter) amount)])
      (set-counter-value! counter n)
      (if (< n 0)
          (reset-integrated! (counter-ctxt counter)
            (lambda () ((counter-k counter) #f)))
          (k)))))
```

If the `decrement` procedure is inlined at the top of `cp0`, the call `(k)` to its continuation can be fully inlined because `k` is singly referenced and the expression `(lambda () (exp-case exp ...))` is a source-program operand. Without the single reference handling described in this section, the continuation would be bound by a `let` in the residual program, resulting in less efficient execution.

4.6 Extensions

This section describes extensions that permit the restrained algorithm to better handle recursive procedures, to extend the benefits of demand-driven processing to certain primitive calls, and to optimize calls that employ different parameter passing and value return conventions.

4.6.1 Inlining Recursive Procedures

Calls to non-recursive procedures are inlined when the size of the resulting residual code does not exceed the size limit and the effort limit is not exceeded. Calls to recursive procedures are inlined only if the above conditions hold and (1) no recursive calls remain in the residual code, or (2) the procedure body can be specialized to the actual parameters at the call site. Many calls satisfying condition (1) are already inlined as a natural consequence of the algorithm already presented. In this section we describe extensions that allow the algorithm to discover additional cases in which calls satisfy condition (1) and that allow the algorithm to inline calls satisfying condition (2).

To support inlining of recursive procedures, we extend the algorithm to collect additional information about procedures that may be inlined. First, recursive procedures are marked so that an alternative inlining strategy can be applied. Using the cycle detection mechanisms described in Section 4.4, the algorithm will discover that a variable `f` is bound to a recursive procedure when the `prelex-operand` of `f` is initially processed for value.[14] Second, the algorithm collects the operands

[14] Recall that variable bindings and call operands are processed on demand. Operand structures represent both.

at the call sites within the body of the recursive procedure. When a procedural operand is first processed for value, the `inner-pending` flag prevents indefinite expansion of recursive calls within the body and instead forces them to residualize as ordinary (recursive) calls. The algorithm records the operands at these call sites for later scrutiny.

Recursive procedures are inlined with an alternative strategy that collects information about whether recursive references remain when the procedure is inlined at a particular call site and also whether references to the formal parameters of the procedure were residualized during the inlining attempt. For example, when attempting to inline a call to a recursive procedure bound to `f`, the environment is extended to map `f` to a fresh variable, `f′`, with no operand, and the procedure body is processed in the outer context of the call context. Because `f′` has no operand, recursive calls within the body are forced to residualize. If, after processing the body, `f′` is unreferenced, then no recursive calls were residualized, *i.e.*, the call folded completely and the inlining attempt succeeds. This happens, for example, when processing the call `(f 0)` given the definition of `f` below.

```
(define f
  (lambda (x)
    (if (zero? x)
        1
        (* x (f (- x 1))))))
```

If, after processing the body, `f′` is referenced but the fresh formal parameters are not referenced, then copy propagation or inlining successfully replaced references to the formals within the body. This often means the operands of recursive calls to `f` are known. For example, if `f` is the factorial function defined above, processing the call `(f 5)` in this manner yields `(* 5 (f′ 4))`. If some of the operands at the original call site have constant residual expressions, a size counter is set along with modest effort and unfolding counters, and we attempt to completely unfold the call. If these limits are not exceeded and no recursive calls remain in the resulting residual code, the inlining attempt succeeds. Unrolling short loops in this way can expose additional constants and create new opportunities for optimization. For example, the call `(f 5)` reduces to `120`.

When the preceding conditions do not hold, a call to a recursive procedure may still be inlined if the procedure body can be specialized to the arguments at the call site. For example, consider the definition of `fold` below.

```
(define fold
  (lambda (f x b end? first rest)
    (if (end? x)
        b
        (f (first x) (fold f (rest x) b end? first rest)))))
```

When the operand bound to `fold` is initially processed, the algorithm records the residual code for the operands at each recursive call as described above. Specialization is attempted if some formal parameters are trivially invariant and the residual expressions for their operands at the original call site are not variable references. The latter restriction avoids some unprofitable specialization in which free variables at the call site are substituted into the body of a local loop inlined at the call site.

In the definition of `fold` above, all parameters except `x` are trivially invariant. To determine this, the algorithm examines the operands at each recursive call within the body of `fold`. As described earlier, these operands are collected when the `prelex-operand` of `fold` is initially visited for value. Using this information, the algorithm computes the set of unassigned formal parameters that are referenced at the corresponding argument positions at each recursive call site within the body. Said another way, a formal parameter is in this set only if it is unassigned and at each recursive call site within the body, the corresponding operand is a reference to that formal parameter. Unassigned formal parameters are easily identified by consulting the `prelex-residual-assigned?` flags. Because all code in the scope of the formal parameters (ie. the procedure body) has already been processed for value, the `prelex-residual` flags are used instead of the more conservative `prelex-source` flags.

Because the formal parameters `f`, `b`, `end?`, `first`, and `rest` are found to be trivially invariant, the algorithm rewrites the earlier definition of `fold` as follows and caches the resulting code in the `prelex-operand` of `fold`.

```
(define fold
  (lambda (f x b end? first rest)
    (let loop ((x x))
      (if (end? x)
          b
          (f (first x) (loop (rest x)))))))
```

The body of the function is replaced with a local loop whose only parameters are the formal parameters of `fold` that were not invariant. Recursive calls to `fold` are replaced with recursive calls to the local `loop`.[15] In effect, this provides a light-weight form of lambda-dropping [21] on demand. Inlining the modified version of `fold` can result in useful specialization. For example, inlining the call

```
(fold * n 1 zero? (lambda (x) x) (lambda (x) (- x 1)))
```

yields the following specialized code.

```
(let loop ((x n))
  (if (zero? x)
      1
```

[15] Because the closure bound to `fold` may escape to unknown call sites, non-call references to `fold` (even within the procedure body) cannot be replaced with references to `loop`.

```
(* x (loop (- x 1)))))
```

4.6.2 Primitive Handlers

When the algorithm encounters a primitive application, it attempts constant folding by default. Certain primitives, however, benefit from more specialized treatment. For example, the primitives `memq`, `memv`, `member`, `assq`, `assv`, and `assoc` search in various ways for the value of their first argument within the list supplied as their second argument. If the second argument is the empty list, the search must fail. Primitive handlers provide a way to incorporate optimizations that exploit properties such as this. For example, the primitive handler associated with this set of primitives can make the following transformation.

```
(member e1 '()) → (begin e1 #f)
```

This transformation has two consequences. First, the constant result (`#f`) of the primitive application is exposed where it may be exploited by other optimizations such as elimination of useless or unreachable code. Second, the algorithm can process `e1` for effect only instead of processing it for value. When processed only for effect, many provably terminating effect-free expressions can be discarded. In particular, this allows the algorithm to collect more accurate information about which variables are actually referenced. This information can then be used to eliminate useless bindings.

Primitive handlers of the kind just described simply provide primitive calls with the same benefits of demand-driven online processing that ordinary user code enjoys by default. For example, the call `(member e1 '())` is automatically translated to `(begin e1 #f)` given the following user-level definition of `member`.

```
(define member
  (lambda (x ls)
    (cond
      ((null? ls) #f)
      ((eq? x (car ls)) ls)
      (else (member x (cdr ls))))))
```

The primitive handler for `member` has a clear analog at the source level because `member` is actually a library function that could have been defined as shown above.[16]

Other primitive handlers implement optimizations that exploit particular properties of system primitives. For example, the primitives `eq?`, `eqv?`, and `equal?` must return true when applied to arguments whose result expressions are both references to the same unassigned variable. The handler

[16] Unlike the simple definition given here, the actual library definition performs error checking to detect cases where the input is an improper or cyclic list.

for these primitives first visits the operands for value then compares the result expressions of the two residual arguments. If both are references to the same unassigned variable, a sequence is residualized to preserve any side effects in the operands before returning `#t`. Visiting the operands for value first ensures that copy propagation has been done before the result expressions are compared. As a result, the primitive handler can perform transformations like the following.

```
(lambda (x)
  (let ((y x))
    (let ((z y))
      (eq? x z))))  →  (lambda (x) #t)
```

Similar transformations are possible for the primitives `+`, `-`, `*`, and `/`. Handlers for these primitives first visit the operands for value in order to perform any constant folding and constant propagation and then examine the result expressions of the residual arguments looking for the identity element for the corresponding operation. These handlers can make the following transformations.[17]

```
(+ e1 0) → e1
(- e1 0) → e1
(- 0 e1) → (- e1)
(* e1 1) → e1
(* 1 e1) → e1
(/ e1 1) → e1
(* e1 0) → (begin e1 0)
(* 0 e1) → (begin e1 0)
(/ 0 e1) → (begin e1 0)
```

In the last three transformations it is useful to look for a zero argument before processing the operands for value. While this may not recognize cases in which the zero is produced via constant propagation or constant folding, it is easy to check again after visiting the operands for value.[18] When successful, this inexpensive syntactic check has the advantage that the operands have not been processed yet for value. In particular, the `prelex` flags for variables referenced only within the useless operands are not set prematurely. More accurate `prelex` flags can help the algorithm identify and eliminate more useless bindings.

```
(lambda (x)
  (let ((y (* x x)))  →  (lambda (x) 0)
    (/ 0 y)))
```

Primitive handlers can also be used to introduce new core constructs. The following subsections describe the use of primitive handlers to introduce new calling conventions to better support `apply` and `call-with-values`.

[17] In general, these transformations cannot be justified unless the type of `e1` is known to be numeric.

[18] Constant zero arguments not apparent in the original source program may be revealed when a procedure is initially processed for value prior to inlining. During subsequent inlining attempts, the simple syntactic check in the primitive handlers for `*` and `/` may therefore succeed in some cases that might not be expected.

Handling `apply`

Normally, inlining is attempted only for calls of the form (e$_0$ e$_1$ ... e$_n$). The Scheme programming language also provides a primitive, `apply`, that takes $n \geq 2$ arguments and invokes its first argument, a$_1$, passing as arguments any a$_i$, $1 < i < n$, and any elements of the list a$_n$. In effect, `apply` introduces a different calling convention. The primitive handler for `apply` allows the inliner to recognize and inline calls that use this calling convention.

To support `apply`, `app` structures are extended with a new `convention` field that indicates the calling convention (either `call` or `apply`) at the call site represented by the `app` structure. The `app-convention` field is consulted when constant folding is attempted and when inlining is attempted. When residualizing a call that cannot be reduced by either of these methods, the algorithm consults the `app-convention` field to select the appropriate convention for the residual call.

The primitive handler for `apply` simply processes its first operand (the procedure to be invoked) in a new `app` context containing the remaining operands. The `convention` field of this `app` context is marked `apply`. This calling convention is supported by the routines that perform constant folding and procedure inlining, as described below.

In the case of constant folding, the `app-convention` field determines the composition of the list passed to `eval`.[19] For example, the call `(eval (list p c))` in the definition of `fold-prim` (on page 129) would be replaced by the following.

```
(eval
  (case (app-convention ctxt)
    ((call) (list p c))
    ((apply) (list 'apply p (list c)))))
```

When extended to calls of more than one argument, constant folding is attempted only when each operand at the call site residualizes as a constant. In the corresponding excerpt given below, the variable `cs` is bound to a list of operands with constant residual values.

```
(eval
  (case (app-convention ctxt)
    ((call) (cons p cs))
    ((apply) (list 'apply p cs))))
```

When processing a `lambda` expression in an `app` context using an `apply` convention, the inlining

[19]To simplify the presentation, we assume constant folding is achieved by passing to `eval` a list with the primitive as its head and a list of constant arguments as its tail. In the actual implementation of the algorithm, `apply` is used to apply the procedure value of the primitive to a list of constant arguments. The structure of this argument list depends on the `app-convention` field of the call context. The procedure value of the primitive is accessed via a special slot in the symbol structure. Using `apply` instead of `eval` avoids the overhead of compiling or interpreting a constructed call each time constant folding is attempted.

algorithm attempts to rewrite the `lambda` expression so that it performs the necessary list destructuring explicitly. This has the effect of unrolling the loop contained in the primitive `apply` code with the benefit that the `lambda` expression can then be inlined. For example, a call of the form

$$\texttt{(apply (lambda (x}_1\ \texttt{...}\ \texttt{x}_n\texttt{) e) a}_1\ \texttt{...}\ \texttt{a}_m\ \texttt{a}_r\texttt{)}$$

is transformed into the following nested `let` expression.

```
(let ((x1 a1) ... (xm am) (t ar))
  (let ((xm+1 (car t)) (t (cdr t)))
    ...
      (let ((xn-1 (car t)) (t (cdr t)))
        (let ((xn (car t)))
          (if (null? (cdr t))
              e
              (error 'apply "incorrect number of arguments"))))))
```

One unfortunate consequence of this transformation is that less informative error messages may be generated in the case where the list bound to $\texttt{a}_r$ is not of length $n - m$. For this reason, the transformation is applied only at higher optimization levels.

When a `lambda` expression is successfully transformed as just described, it is then processed in a new `app` context containing the relevant operands from the original `app` context. The new `app` context uses the ordinary `call` calling convention. Processing the transformed `lambda` expression in an ordinary `app` context is equivalent to optimizing a `let` expression. The following examples show compile-time transformations enabled by the primitive handler for `apply`.

```
(apply + '(1 2 3))
  → 6
(apply (lambda (x y z) (+ x (* y z))) 1 '(2 3))
  → 7
(apply (lambda (x y z) (+ x (* y z))) a b)
  → (let ((y (car b)) (t (cdr b)))
       (let ((z (car t)))
         (if (null? (cdr t))
             (+ a (* y z))
             (error 'apply "incorrect number of arguments"))))
```

The advantage in the first and second examples is clear. In the third example, the transformed code avoids the cost of allocating a closure for the `lambda` expression and avoids the cost of an anonymous procedure call. Copy propagation has eliminated the outer `(let ((x a) (t b)) ...)` described in the general transformation.

Handling `call-with-values`

The primitives `values` and `call-with-values` support procedure calls that return multiple values. Multiple values are returned via the `values` primitive, which accepts zero or more arguments. A multiple-value call is initiated by `call-with-values`. The primitive procedure `call-with-values` expects two arguments: the *producer* is a procedure of no arguments, and the *consumer* is a procedure that should expect as many values as may be returned when the producer is invoked. For example, the following expression evaluates to `7`.

```
(call-with-values
  (lambda () (values 1 2 3))
  (lambda (x y z) (+ x (* y z))))
```

The primary objective of the primitive handler for `call-with-values` is to apply the consumer directly to the values that are returned when the producer is invoked. When the consumer is a known procedure, the `call-with-values` handler may be able to inline the consumer where the values are produced, as shown below.

```
(let ([g (lambda () (values 1 2))]
      [f (lambda (x y) (list y x))])
  (call-with-values g f))
 →  (list 2 1)
```

When the consumer is not known, the `call-with-values` handler attempts to at least residualize a direct call to the consumer where the values are produced, as in the following example.

```
(let ([g (lambda () (values 1 2))])
  (lambda (f)
    (call-with-values g f)))
 →  (lambda (f) (f 1 2))
```

These transformations succeed only when the producer is a known procedure and the algorithm is able to determine precisely which values are returned.

The primitive handler for `call-with-values` is invoked with two operands corresponding to the producer and consumer arguments. To preserve the applicative-order semantics of this primitive, the handler introduces local temporary bindings (marked singly referenced) for the operands and relies on the existing inlining machinery to residualize effects where arguments are evaluated while propagating copyable values. A reference to the producer temporary (`p-temp`) is processed in a call context with an outer context of `value`. If the call is inlined, the handler examines the result expression of the residual code in an attempt to identify the values returned. If the result expression is a constant, a variable reference, an assignment, a `lambda` expression, or a primitive reference, the result expression is used as the single value returned. If the result expression is a call to the `values` primitive, the arguments of that call are used as the values returned. Otherwise, the primitive handler must residualize the call to the producer.

If the producer call was inlined and the return values were found, the primitive handler constructs a residual sequence that (1) evaluates the effect-only portion of the `p-temp` binding and (2) returns the result of attempting to inline a call to the consumer temporary (`c-temp`) with the returned values as operands.[20] If the producer call could not be inlined or was inlined but the return values could not be found, the primitive handler constructs a residual call to `call-with-values`. When constructing the residual call, the primitive handler attempts to expose `lambda` expressions in the `result-exp` of the residual producer and consumer arguments to assist subsequent passes of the compiler in generating more efficient code [6]. For example, the primitive handler would convert

```
(lambda (f g)
  (call-with-values
    (begin
      (printf "producer evaluated~n")
      (lambda () (f)))
    (begin
      (printf "consumer evaluated~n")
      (lambda (x y z) (g x y z)))))
```

into the following.

```
(lambda (f g)
  (printf "producer evaluated~n")
  (printf "consumer evaluated~n")
  (call-with-values
    (lambda () (f))
    (lambda (x y z) (g x y z))))
```

4.7 Performance

We have implemented the inlining algorithm as an additional pass within *Chez Scheme*, a commercial optimizing compiler for Scheme. The inlining algorithm was evaluated using a suite of common Scheme benchmark programs [5, 35]. The benchmark programs provided by Jagannathan and Wright [35] had already been processed by their local simplification pass, which performs many of the same optimizations as our inliner. To facilitate comparison with their results, performance data was collected on a 150 MHz SGI MIPS R4400 workstation.

Table 1 shows the performance increase obtained when the benchmarks are optimized using

[20] Because the return values are extracted from residual code, the operands constructed for them are marked as having been processed.

Table 1: Performance increase due to inlining with different size and effort limits.

	Effort limit 100						Effort limit 1000					
Size limit:	10	20	30	40	50	60	10	20	30	40	50	60
boyer	1.03	1.04	1.04	1.04	1.04	1.04	1.03	1.04	1.06	1.06	1.07	1.06
dynamic	1.05	1.07	1.07	1.08	1.06	1.05	1.08	1.10	1.05	1.05	1.10	1.07
graphs	1.79	1.78	1.78	1.78	1.78	1.77	1.76	2.25	2.24	2.25	2.25	1.59
lattice	4.30	4.37	4.39	4.40	4.39	4.39	4.30	4.57	4.48	4.57	4.56	4.57
matrix	1.29	1.27	1.27	1.29	1.30	1.29	1.31	1.28	1.30	1.31	1.32	1.30
maze	1.45	1.38	1.40	1.35	1.40	1.42	1.46	1.39	1.40	1.42	1.41	1.39
nbody	.95	.97	.98	1.04	1.04	1.05	.92	1.00	.98	.93	1.04	1.03
splay	1.26	1.29	1.30	1.29	1.28	1.28	1.32	1.20	1.33	1.31	1.38	1.35
conform	2.41	2.50	2.62	2.70	2.69	2.66	2.44	2.52	2.47	2.68	2.67	2.69
earley	1.10	1.10	1.08	1.09	1.09	1.10	1.09	1.11	1.12	1.12	1.12	1.12
em-fun	1.15	1.18	1.05	1.02	1.09	1.02	1.20	1.15	1.07	1.15	1.15	1.14
em-imp	1.16	1.10	1.06	1.11	1.15	1.06	1.16	1.09	1.06	1.13	1.05	1.06
interpret	1.07	1.03	1.09	1.09	1.10	1.10	1.08	.99	1.10	1.07	1.07	1.07
peval	1.22	1.07	1.05	1.07	1.07	1.06	1.07	1.21	1.23	1.22	1.23	1.24
simplex	1.32	1.32	1.30	1.32	1.32	1.32	1.31	1.32	1.31	1.32	1.30	1.30

different size and effort limits. With the exception of nbody, all programs show consistent and significant speedups. Profiling reveals that nbody spends most of its time indexing three levels deep into a structure of nested arrays. Thus, the variability we observe is likely due to cache effects. When run on the Intel Pentium Pro, nbody shows consistent speedups ranging from 1.06 to 1.11 (6–11%). As is to be expected, increasing the effort limit generally improves performance somewhat. At higher effort limits, graphs improves substantially.

Inlining more than doubles the performance of lattice, graphs, and conform. A large percentage of run-time calls made by lattice (73%) and graphs (43%) are to a user-defined `memv` procedure that searches a list for a given key. With a size limit of 4, our algorithm completely unfolds calls to `memv` in both programs because the target list happens to be a short constant at each call site for `memv`. The speedup of 4.57 for lattice at size limit 20 represents an additional speedup of 1.15 beyond the speedup of 3.97 due to unfolding calls to `memv` at size limit 4. The speedup of 2.25 for graphs at size limit 20 represents an additional speedup of 1.43 beyond the speedup of 1.57 due to unfolding calls to `memv` at size limit 4. The algorithm does not unfold any of the recursive calls within conform. The large speedup is due simply to the fact that conform contains many small procedures that can be inlined.

Table 2 shows the effect of different size and effort limits on the size of the resulting object code. The table reports the ratio of code size after optimization to the original code size. For many

Table 2: Ratio of code size after inlining to original code size before inlining with different size and effort limits. Numbers less than 1 indicate a decrease in code size.

	Effort limit 100						Effort limit 1000					
Size limit:	10	20	30	40	50	60	10	20	30	40	50	60
boyer	.97	.97	.99	1.00	1.00	1.00	.97	.99	.99	1.01	1.01	1.01
dynamic	.87	1.18	1.20	1.20	1.31	1.31	.86	1.26	1.30	1.37	1.55	1.57
graphs	.72	.71	.71	.71	.71	.71	.80	.85	.85	.85	.85	.90
lattice	1.13	.91	.91	.91	.91	.91	1.13	.89	.89	.89	.89	.89
matrix	.96	1.01	1.03	1.03	1.03	1.03	.94	.99	1.01	1.00	1.00	1.00
maze	.74	.74	.78	.78	.78	.78	.74	.74	.78	.78	.78	.78
nbody	1.20	1.22	1.25	1.25	1.25	1.25	1.24	1.27	1.33	1.34	1.40	1.46
splay	1.00	1.00	1.02	1.06	1.06	1.06	.99	1.01	1.12	1.12	1.10	1.12
conform	.73	.80	.82	.86	.86	.86	.72	.75	.76	.80	.80	.80
earley	.91	.92	.96	.96	.96	.96	.82	.85	.89	.89	.91	.94
em-fun	.88	.92	1.07	1.09	1.09	1.09	.88	.91	1.15	1.17	1.13	1.12
em-imp	.87	.89	.89	.92	.92	.92	.85	.89	.88	.90	.90	.90
interpret	1.19	1.26	1.31	1.31	1.31	1.31	1.17	1.23	1.28	1.29	1.29	1.35
peval	1.01	1.04	1.04	1.04	1.04	1.04	1.00	1.03	1.03	1.03	1.03	1.03
simplex	.86	.86	.86	.86	.86	.86	.86	.86	.86	.86	.90	.90

programs, inlining actually decreases code size. For most programs, code size increases gradually as the size limit increases. This confirms the intuition that the size counter is an effective mechanism for regulating code growth.

Table 3 shows the run time of the algorithm for different size and effort limits. The algorithm completes both analysis and transformation of each program (except dynamic) in less than one second. Increasing the effort limit by an order of magnitude typically only doubles the run time of the algorithm. When the size limit is varied, the run time of the algorithm increases roughly in proportion to the increase in code size.

4.8 Related Work

Jagannathan and Wright [35] use a polyvariant flow analysis to identify inlining opportunities and to estimate the size of a procedure when specialized for a particular call site. Their approach is offline: all inlining decisions are made prior to any transformation. Our algorithm combines a less accurate analysis (effectively sub-0CFA [4]) with online transformation and polyvariant specialization. The time required by their flow analysis varies widely and can be excessive—110 seconds to analyze dynamic—rendering their method impractical for use in a production compiler.

Table 3: Run time of the algorithm (in seconds) with different size and effort limits.

	Effort limit 100						Effort limit 1000					
Size limit:	10	20	30	40	50	60	10	20	30	40	50	60
boyer	.03	.04	.03	.04	.04	.04	.02	.08	.07	.08	.09	.09
dynamic	.33	.48	.52	.50	.54	.53	.34	.56	.57	.68	.81	1.06
graphs	.08	.08	.09	.09	.08	.07	.11	.13	.14	.13	.15	.14
lattice	.08	.08	.09	.08	.07	.08	.09	.11	.12	.10	.11	.12
matrix	.11	.11	.11	.11	.13	.11	.14	.18	.20	.21	.21	.21
maze	.11	.11	.13	.12	.13	.12	.11	.12	.13	.13	.14	.13
nbody	.20	.20	.20	.22	.20	.22	.29	.47	.48	.51	.57	.57
splay	.10	.10	.10	.09	.10	.11	.13	.15	.13	.13	.14	.16
conform	.08	.12	.13	.14	.14	.15	.24	.27	.28	.32	.34	.31
earley	.13	.15	.15	.15	.15	.17	.26	.31	.40	.41	.44	.45
em-fun	.17	.19	.20	.21	.21	.22	.40	.45	.66	.70	.68	.64
em-imp	.16	.18	.18	.18	.19	.18	.34	.45	.46	.50	.54	.51
interpret	.28	.29	.33	.34	.34	.34	.29	.31	.47	.51	.56	.55
peval	.08	.08	.09	.12	.13	.13	.17	.19	.19	.19	.22	.19
simplex	.05	.06	.06	.06	.05	.05	.09	.10	.10	.14	.14	.13

Table 4 provides a rough comparison of the best speedups obtained by the flow-directed inlining algorithm of Jagannathan and Wright with the speedups obtained by our algorithm (labeled cp0) given a size limit of 50 and an effort limit of 1000. The table also compares processing time and the ranges of object code size for the two methods. The comparison is complicated by the fact that their optimizer effectively block compiles the benchmarks, yet their baseline for comparison is not block-compiled. Enabling block compilation reduces baseline object-code size and improves baseline performance considerably. To compensate for the difference in benchmarking methodology we present both our absolute results and the results we obtain relative to a baseline that is not block compiled. It appears that we achieve equivalent or better speedups for all benchmarks except boyer and better code size ratios for all benchmarks except splay, yet our algorithm completes both analysis and transformation in far less time than their analysis alone.

Ashley [5] evaluated the effectiveness of four different flow analyses for inlining. The analyses range from a fast analysis less accurate than 0CFA to a polyvariant analysis similar to 1CFA. His inlining algorithm is based on that of Jagannathan and Wright but exposes more opportunities for inlining.

Table 5 compares the performance increase, processing time, and code size ratios for Ashley's inlining algorithm justified by sub-0CFA and 1CFA analyses with the results obtained by our algorithm

Table 4: Performance increase, processing time and range of code size ratios for our algorithm (cp0) and the polyvariant flow-directed inlining of Jagannathan and Wright (flow). The columns labeled "non-block" indicate results relative to a baseline that is not block compiled. All programs were compiled and run using *Chez Scheme*. All data was collected on the 150 MHz SGI MIPS R4400.

	Performance Increase			Processing Time (seconds)		Code Size Ratios		
Baseline	block	non-block				block	non-block	
Algorithm	cp0	cp0	flow	cp0	flow	cp0	cp0	flow
boyer	1.07	1.34	1.72	.09	1.4	.97 – 1.01	.82 – .86	.95 – 1.10
dynamic	1.10	1.20	1.20	.56	110.3	.86 – 1.57	.78 – 1.43	1.52 – 2.48
graphs	2.25	2.45	1.30	.13	.3	.71 – .90	.65 – .82	.79 – .87
lattice	4.57	6.40	1.27	.11	.2	.89 – 1.13	.72 – .92	1.02 – 3.09
matrix	1.32	1.35	1.11	.21	.4	.94 – 1.03	.86 – .94	.93 – 1.21
maze	1.46	1.88	1.67	.11	.5	.74 – .78	.48 – .51	.89 – .98
nbody	1.04	1.38	1.11	.22	2.6	1.20 – 1.46	1.08 – 1.31	1.25 – 2.62
splay	1.38	1.35	1.20	.14	4.0	.99 – 1.12	.94 – 1.06	.94 – .95

with effort limits of 100 and 1000. All data was generated using the same compiler and benchmarking methodology. In particular, the baseline and optimized programs were block-compiled in all cases. The data was collected on practically identical hardware (our machine has half the physical memory and second-level cache). All results in the table reflect a size threshold of 30. Our algorithm achieves better or virtually equivalent speedups for all of the benchmarks and typically completes both analysis and transformation in less time than is required for flow analysis alone. The volatility of the 1CFA analysis is highlighted by the fact that it exhausted virtual memory (128Mb core, 100Mb swap) while attempting to analyze `interpret`. Our code size ratios are consistently better than those obtained using either flow analysis. This supports our claim that online transformation provides more accurate size information on which to base inlining decisions.

The intuitive appeal of an offline flow-directed approach is that the results of flow analysis might be used to justify other optimizations in addition to inlining. In fact, Ashley found that inlining violates the soundness of flow information about procedures which could otherwise be used by procedure call optimizations in later passes of the compiler. This means that flow analysis must be repeated after inlining.

In the SML/NJ compiler, Appel uses an offline algorithm in which a data gathering pass is run alternately with a transformation pass that performs several optimizations including constant folding, β-contraction, uncurrying, and inlining of functions called only once [2]. This transformation

Table 5: Comparison of performance increase, processing time, and ratio of code size to original code size for two flow-directed algorithms and our algorithm run with effort limits of 100 and 1000. All data was gathered on the 200 MHz Intel Pentium Pro. All programs were compiled and run using *Chez Scheme*. Processing time for $_0$cfaand 1cfa cases includes analysis time only, *i.e.*, it does not reflect the cost of inlining or subsequent simplification. A size threshold of 30 was used in all cases.

	Performance Increase				Processing Time (seconds)				Code Size Ratio			
	$_0$cfa	1cfa	100	1000	$_0$cfa	1cfa	100	1000	$_0$cfa	1cfa	100	1000
boyer	1.01	1.03	1.01	1.02	.01	.07	.02	.03	1.47	2.33	.98	1.01
dynamic	1.11	1.05	1.10	1.10	.32	2.09	.20	.23	3.56	4.72	1.46	1.61
graphs	1.20	1.27	1.70	2.18	.05	.20	.03	.05	.73	.82	.68	.72
lattice	1.03	1.33	3.92	4.06	.01	.19	.03	.04	1.20	1.51	.94	.92
matrix	1.05	1.09	1.22	1.21	.05	3.02	.06	.09	1.11	1.23	1.04	1.02
maze	1.27	1.27	1.28	1.29	.05	.19	.03	.04	.88	.89	.75	.75
nbody	1.06	1.10	1.09	1.06	.13	1.98	.08	.19	1.29	1.37	1.19	1.25
splay	1.09	1.11	1.23	1.26	.06	.95	.02	.05	1.42	1.46	1.07	1.18
conform	1.72	1.75	1.94	1.95	.04	.31	.05	.12	1.06	1.19	.86	.81
earley	1.10	1.11	1.07	1.10	.07	.45	.06	.15	1.06	1.13	1.00	.94
interpret	1.12	—	1.14	1.15	.39	—	.13	.19	2.06	—	1.47	1.45
peval	1.15	1.15	1.14	1.15	.07	.41	.04	.07	1.20	1.31	1.04	1.03
simplex	1.39	1.39	1.43	1.43	.03	.19	.03	.04	.89	.92	.81	.81

phase is iterated until the number of contractions found is below some threshold. Next, inline expansion is performed using several heuristics to estimate the cost and benefit of inlining. To improve these estimates, this "round" of optimization is iterated beginning with the contraction phase. Although our algorithm can be iterated, we have found that few programs actually benefit from additional passes.

Appel and Jim recently described a linear-time algorithm that performs constant folding, dead-variable elimination, and inlining of functions called only once [3]. They have implemented an $O(n^2)$ variant of the algorithm to replace the contraction phase of the SML/NJ compiler described above. The new algorithm is online in that usage counts for variables are updated as optimizations are performed, and it is demand-driven in that bodies of binding forms are processed before processing the bindings. The new algorithm typically performs as many contractions in one pass as their earlier offline algorithm performs in three. They do not consider the more difficult problem of inlining functions called more than once.

Dean and Chambers [23] describe an online approach in which the cost and benefit of inlining at a call site are estimated by examining the code produced when the routine is tentatively inlined and optimized for the call site. To amortize the high cost of these inlining trials, they cache the results in a persistent database indexed by a description of the static information at the call site that justified optimizations that made inlining acceptable. By using inlining trials instead of offline scoring estimates, they were able to reduce overall compile time and code size with minimal loss of run-time performance. Our algorithm can be viewed as incorporating a lightweight form of inlining trials.

Wegman and Zadeck [66] present a fast global constant propagation algorithm and show how it can be extended to perform many of the specializations needed when procedures are inlined. Their algorithm propagates constants through the static single-assignment (SSA) graph [20] using a demand-driven traversal of the control-flow graph that facilitates elimination of unreachable code which, in turn, improves the accuracy of the information collected. The inlining algorithm they propose is offline and does not address key issues such as identifying inlining candidates, limiting code growth, and achieving termination in the presence of recursion.

Ball describes an analysis that determines which parameters contribute to the value of the expressions in a procedure body [8]. When constant parameters are available at a call site he uses the parameter dependency information to guide inlining decisions with an estimate of code savings and performance gain that is based on predictions about the impact of subsequent optimizations. We suspect the speed of our algorithm could be improved by using the type-group analysis of Dean and Chambers, or the more direct parameter dependency analysis of Ball, to exploit polyvariance only where it is likely to be useful.

Serrano presents simple heuristics for deciding when and how to inline calls within programs written in a first-order subset of Scheme [55]. His algorithm compares the number of abstract syntax tree nodes representing the body of a called function with the number of arguments passed at a call site. A call is inlined when the ratio of these numbers is below a threshold that decays as a function of the number of nested calls to the inlining algorithm. When inlining a recursive function, his algorithm creates a local recursive definition at the call site in the hope that useful specialization will result. Because no attempt is made to account for the effect of subsequent optimizations on code size, his algorithm can reject profitable inlining opportunities that appear costly but which, in fact, do not significantly increase code size. For the six benchmarks we have in common, conform, boyer, peval, earley, matrix, and maze, the average of his best speedups is 1.18 and code size changes by a factor of .96 on average. While direct comparison is difficult due to architectural differences[21]

[21] Serrano measured performance using the Bigloo Scheme implementation on a DEC Alpha 3000/400 with 64 megabytes of RAM running DEC OSF/1 v4.0.

and differences in the baseline Scheme implementations, our average speedup of 1.48 (on the 150 Mhz MIPS R4400) and 1.31 (on the 200 Mhz Intel Pentium Pro) and average code ratio of .92 (on both platforms) suggest that our algorithm is more effective. No data is available on the run time of his inlining algorithm.

4.9 Future Work

Speed and efficacy are hallmarks of the optimization pass developed in this chapter. While the optimizer is already quite capable, there are many interesting directions to explore in future work that may lead to a faster or more effective algorithm. In particular, while this chapter has demonstrated that online transformation yields effective optimization, future work might explore hybrid approaches that combine offline analysis to improve run time or quality of results.

4.9.1 A Faster Algorithm

As demonstrated in Section 4.7, the existing implementation of the algorithm is already fast enough for use in a production compiler. Careful profiling and hand-optimization have not yet been applied, however, and it is likely that the run time of the algorithm could be improved in this manner. More significant gains might be achieved by modifying the core algorithm.

Avoiding pointless respecialization attempts

Many inlining attempts are destined to fail. Recognizing and avoiding these doomed inlining attempts could save processing time. To maximize the benefit of online transformation, however, it is important to pursue any inlining attempt with potential to succeed. As suggested in Section 4.8, we could adapt Ball's parameter dependency analysis [8] to determine which expressions in the body of a procedure must be residualized regardless of the actual parameter values. This information could be collected by an offline analysis and used to compute a minimum size for each procedure. Before attempting to inline a call, the inlining algorithm could compare the value of the current size counter against the precomputed minimum size of the candidate procedure and avoid any attempt that will clearly exceed the available size limit.

Even when it is not clear that an attempt will exceed the available size limit, it could still be useful to decrement the size counter, before attempting to inline, by the total minimum size of the procedure so that the size counter may abort the inlining attempt earlier. For example, suppose the procedure body contains two large expressions `e1` and `e2`, where the size of `e1` is independent of actual parameter values and the size of `e2` is dependent on the parameter values. When attempting to inline a call to this procedure, the algorithm could first decrement the current size counter by the precomputed minimum size of `e1` and then process `e2` with the resulting size counter. If the size limit is exceeded while processing `e2`, the algorithm saves time because it avoids processing `e1`. Since `e1` does not depend on parameter values, it makes sense to process `e2` first.

In fact, when the offline analysis identifies a parameter-independent expression, it can often precompute the actual residual code for that expression. Precomputed residual code could be represented by extending the intermediate language with a `residual` constructor that records both the precomputed code and its size. This would provide a natural way to incorporate the modified decision process outlined above into the existing inlining algorithm. When it encountered a `residual` form, the inlining algorithm could immediately decrement the size counter by the recorded size and return the precomputed code. Not only does this provide a fast way to account for the size of parameter-independent code, it also saves the time that the existing inlining algorithm would invest reconstructing code that could have been precomputed.

Because contextual information at the call site can have a significant effect on the size of the resulting code, it may be worthwhile to precompute different versions of the parameter-independent code. In particular, it may be useful to precompute the code that would result if the procedure were inlined with an outer context of `value` and another version of the code that would result if the outer context were `effect`.

Reducing interpretive overhead

Because the inlining algorithm is structured as an interpreter, existing techniques for implementing fast interpreters may be applicable. For example, partial evaluation techniques might be used to specialize the inlining algorithm to the task of inlining a frequently called procedure. This might be particularly useful in the case of cross-module inlining during separate compilation. When compiling a library that will be used many times, it may be cost effective to generate specialized inlining procedures. In some respects, the primitive handlers described in Section 4.6.2 can be viewed as the result of specializing the inlining algorithm with respect to the code for the various primitives by hand. Partial evaluation techniques might be adapted to automate this process for all library code.

Just as there are primitives that do not merit primitive handlers, there are surely procedures in user-defined libraries that do not merit specialized procedures for inlining. The same analysis posited earlier as a way to determine the minimum size of residual procedures might also be used to flag procedures whose size is highly dependent on the input parameters and might therefore benefit from fast polyvariant specialization.

4.9.2 A More Effective Algorithm

Profile-directed inlining

Data from execution profiles can be used to focus inlining effort on frequently executed call sites. Small improvements in frequently executed code can have a larger impact on overall run time than large improvements in code that is seldom, if ever, executed. Using profile data to focus inlining efforts could enable the algorithm to make more productive use of the allotted compile time and code size. The existing algorithm could be adapted by scaling the size and effort limits enforced during an inlining attempt according to the expected execution frequency of the call site.

Since actual execution profiles are not always available, it would be useful to develop a static profile estimator. For example, if the `error` primitive is invoked on all control paths through a procedure, it is reasonable to assume that the procedure will rarely, if ever, be called and therefore conclude that the procedure should, in fact, not be inlined. Programmers often place error-handling code out of line deliberately in order to prevent it from being fetched into the cache. More general static branch prediction has been the subject of recent research [15, 49].

Profile data is normally collected for a complete run of a program. When fast turnaround is demanded of the compiler, this approach may be too time consuming. In addition, this approach relies on the availability of a training dataset that closely approximates actual workloads encountered at run time. Profile-directed recompilation presents another alternative [13, 14]. In this model, profile data collected during the execution of a program can be used to selectively and dynamically recompile frequently executed portions of the program as it runs.

Prior work on profile-directed recompilation in Scheme has focused on low-level program representations that support fast code generation. Using such a low-level program representation, it would be difficult to support dynamic profile-directed procedure inlining. Using higher-level representations increases the time required for run-time code generation. Since code generation time contributes to total run time, it is difficult to predict whether high-level optimizations will improve run time performance enough to amortize the increased cost of compilation. This is further incentive to explore techniques that could speed up the inlining algorithm.

Refined regulatory mechanisms

Although approximate, the size limit mechanism used to regulate code growth due to inlining is surprisingly effective. The size of an expression is estimated by counting the number of nodes in the abstract syntax tree representing the expression. More accurate prediction of actual object code size might result in even tighter control over code growth. Explicitly modeling the availability of other resources, such as registers, might also help contain the costs of inlining. Doing this efficiently is a worthwhile subject for further study.

The quality of results obtained by the algorithm currently depends on the order in which expressions are processed. This is particularly evident in the handling of conditionals, sequencing forms, and the operands of calls that cannot be inlined. Consider the expression `(begin e1 e2)`. The algorithm on page 136 processes `e1` and `e2` using the same size counter. If `e1` is processed first, it may consume the entire size limit, preventing useful inlining within `e2`. Better overall results might be obtained by allocating some fraction of the size limit to both subexpressions. It may be useful to collect any surplus size allotment that remains after processing the subexpressions so that it can be redistributed among the subexpressions that exceeded their initial allotment. Developing efficient mechanisms that distribute size limit resources fairly to subexpressions is an interesting area for future work.

More accurate reference counts

The output of `cp0` could be improved if more accurate reference count information were collected. We saw earlier that accuracy may be lost when visiting an operand for value. This is particularly evident when processing `letrec` bindings such as the following.

```
(lambda (x)
  (letrec ([f (lambda (n) (if (zero? n) 1 (1+ (f (1- n)))))])
    (if (zero? (f 0)) x (begin (set! x 45) cons))))
```

Due to inaccurate counts for variable references and conservative information for variable assignments, this expression reduces to `(lambda (x) cons)` only when `cp0` is iterated. To remedy this, the macro expander could be modified to collect actual reference counts for each variable. When residualizing a variable reference, `cp0` could increment the count for that variable. When eliminating useless or unreachable code, `cp0` could decrement the counts of each variable referenced in that code.

While intuitively appealing, this solution is probably too costly as it requires traversing all eliminated code in order to decrement reference counts. In addition, accurate reference count information may not be available until after it is needed. For example, to avoid residualizing the useless assignment to `x` in the example above, `cp0` must decrement the reference count for `x` in the eliminated consequent branch before processing the alternative branch. If the conditional expression followed the assignment in the program text, an accurate count for `x` would not be available when processing the assignment. Moreover, the extensive use of non-local exits complicates the task of maintaining accurate reference counts. For example, unreachable code might be eliminated in the process of inlining one procedure, causing `cp0` to decrement reference counts for a free variable. If that inlining attempt is subsequently abandoned, the reference count on the free variable is now inaccurate. Because some inaccuracy is inevitable, a slightly less accurate, but very efficient, `cp0` may be preferable to a more accurate, but much slower, algorithm. By iterating `cp0` we may obtain a more accurate algorithm without sacrificing compile time and without increasing the conceptual complexity of the algorithm.

Recognizing pure functional programs

In many cases, the inlining algorithm declines opportunities for optimization because it conservatively assumes that all mutable data structures are mutated and that all freshly allocated structures are tested for pointer equality. This conservative outlook prevents the algorithm from applying transformations, such as the following, that are not safe in general.

```
           (list 1 2 3)  →  '(1 2 3)
(apply (lambda x x) e1)  →  e1
```

When data structures are not mutated or tested for pointer equality, as is often the case, such transformations are safe and can significantly improve the quality of results obtained by the inlining algorithm.

Static flow analysis could be used to identify expressions whose values may be mutated or tested for pointer equality. The inlining algorithm could use this information to recognize circumstances in which more aggressive optimizations may be applied safely. It would be interesting to compare the ability of general-purpose static analyses that vary in speed and accuracy [7] and fast escape analyses [10, 24] to justify these optimizations.

Modifying the input language

Section 4.2.6 describes a source-to-source translation pass that simplifies the structure of `letrec` expressions prior to inlining. Other transformations worth considering as a prelude to inlining include `lambda`-dropping [21] and closure conversion [65].

Section 4.6.1 describes a simple form of `lambda`-dropping that results in better specialization of recursive procedures. Applying a more general `lambda`-dropping algorithm as a pre-pass could improve the quality of code generated and perhaps save processing time. To make more effective use of registers, the compiler could perform `lambda`-lifting [16, 36] after the inlining pass.

Closure conversion rewrites a program so that all closures are explicitly constructed at the source level and no `lambda` expression has free variables. Free variable accesses are translated to explicit operations on the closure argument now passed to each `lambda` expression. While the values of the free variables may not be known until run time, the code field of a closure is known at compile time. Supporting these *partially static* structures would enable the algorithm to inline procedures with free variables and could offer other benefits as well [37]. In particular, support for partially static structures would benefit object-oriented programs. For example, in typical C++ implementations object instances are partially static, with a static method table and dynamic instance variable slots. Support for partially static structures could allow the inliner to inline method calls that would otherwise require an indirect anonymous call.

Chapter 5

Conclusion

5.1 Summary

Robust, lexically scoped syntactic abstractions can be implemented via hygienic macros. This dissertation extends the scope of syntactic abstraction in three ways. First, we enhance the expressive power of a macro system that supports syntactic abstraction. Second, we introduce a module system that enables the use of syntactic abstraction within large software development projects. Finally, we incorporate optimizations that simplify the expression of syntactic abstractions while improving their efficiency.

Chapter 2 describes a syntactic abstraction facility based on the powerful lexically scoped macro system originally developed by Dybvig, Hieb, and Bruggeman [29]. Key contributions of the original work are its careful treatment of identifiers and its unification of "high-level" and "low-level" macros in a system that enforces hygiene with constant overhead. Because a hygienic system cannot treat identifiers as simple symbolic data, the system provides tools for introducing new lexically scoped identifiers, constructing macros that implicitly bind or reference identifiers, and comparing identifiers based on their intended use as free identifiers, bound identifiers, or symbolic data. Together with powerful mechanisms for syntactic reflection, the elegant pattern-based syntax construction and syntax destructuring facilities provided by the system eliminate the distinction between high- and low-level macros.

The original work produced an efficient hygienic expansion algorithm by recognizing and avoiding the traditional conflation of variables and symbols. The input and output of macro transformers is represented by syntax objects that retain the contextual information (marks, substitutions, source-correlation data, etc.) associated with a particular list-structured source expression. This representation supports the lazy α-substitution mechanism that prevents copying of structured constants and also makes the algorithm efficient. Substitutions are introduced by the core binding constructs where they are layered on the syntax objects representing the macro input and are gradually pushed down onto subforms of the input as it is destructured by the parser. An identifier reference is resolved by applying the substitutions that have been pushed down on the identifier and then looking up the result in an expand-time store to determine the syntactic type (keyword, variable, etc.) of the identifier. Substitutions are keyed with marks so that an identifier is renamed only if the marks on the identifier match the marks on the substitution. A fresh mark is applied to the input and the output of a macro transformer. Because identical marks cancel when they meet, marks adhere only to identifiers introduced in the expansion of a macro. When a macro introduces a binding for an introduced identifier, the identifiers representing the bound variable and references to that variable will all retain the same mark. When the core binding construct expands, it introduces a substitution for the bound identifier that can match only the like-named references with the same mark.

Chapter 2 makes several important contributions. First, we extend the treatment of identifiers in the original work with the addition of lexically scoped identifier macros and demonstrate that this enhances the expressive power of the macro system. Many transformations that would otherwise require an *ad hoc* code walk can now be expressed directly using identifier macros. Second, we present `fluid-let-syntax`, a new mechanism for expressing intentional capture that complements the existing `datum->syntax-object` mechanism. We provide a detailed description of the two capture mechanisms including several examples of their application. Using `fluid-let-syntax` we are able to implement, entirely at the user level, a facility for defining integrable procedures that are indistinguishable from ordinary procedures. Third, we present the extended macro expansion algorithm formally and show how the new constructs are supported. In addition, we correct minor errors in the original formalization of the algorithm, and we comment on implementation issues in cases where a straightforward implementation of the algebra would be inefficient. Finally, we show how the macro expander can collect and preserve information that can be used by the compiler and the run-time system.

Chapter 3 presents the design and implementation of a realistic module system that is seamlessly integrated with the lexically scoped macro facility described in Chapter 2. Our design augments the small core language of Scheme with simple module and import forms and employs the powerful syntactic abstraction facility from Chapter 2 to provide more elaborate module constructs. The module system is based entirely on the manipulation of static scope. Consequently, a program containing modules can be understood in terms of a straightforward translation into the core language without module forms. Moreover, module constructs expand into static core-language forms that can be compiled efficiently and produce efficient executable code.

A key observation in Chapter 3 is that the two-level α-conversion machinery used by the macro expander provides a basis for the implementation of the module system. Definitions within a module are processed as internal definitions using a two-pass algorithm. The first pass determines the set of defined identifiers, introduces substitutions to rename these identifiers, and extends the store to record the syntactic types of these identifiers. The second pass expands the right-hand sides of variable definitions in an environment extended with the substitutions collected in the first pass. A `module` effectively intercepts the substitutions introduced by the definitions it contains and records substitutions corresponding to exported identifiers in an interface associated with the module identifier in the expand-time store. The `import` form retrieves the module interface and installs the substitutions recorded for exported variables into the scope inhabited by the `import` form. Identifiers exported from a module are therefore visible only within the module and where import has made the corresponding substitutions available.

We describe our solutions to several important practical problems including incremental and separate compilation, isolated scopes, and efficient import of large modules. In addition, we show that it is generally impossible to determine the precise set of identifiers implicitly exported from a module, even if macros are expressed using only the limited capabilities of `syntax-rules` transformers. We propose a simple alternative that restores the ability to perform static analysis even for modules that export macros.

Module definitions and imports may appear in any context where variable or keyword definitions may appear, allowing modules to be nested or to be used locally. As a result, the module system is useful not only for large-scale software development but also as an expression-level language feature for *micro-modular* programming. We demonstrate the synergy that results from this uniform treatment of module and import forms as definitions. For example, we show that syntactic abstraction can be used to derive from the simple module constructs support for anonymous modules, qualified module access, selective import and renaming, mutual recursion, multiple views on a single module, arbitrary combination of modules, separate interfaces, compound interfaces, incrementally satisfiable interfaces, and more dynamic module facilities recently proposed in the literature. In addition, we present extended examples showing how the module system can be used to implement a fully static system for object-oriented programming. The features described in Chapters 2 and 3 have been fully implemented in the front end of the *Chez Scheme* compiler, and a portable implementation is freely available via `ftp`.

Chapter 4 presents the design and implementation of a procedure inlining algorithm that is both fast and effective. Our algorithm is fast enough to process programs of several thousand lines in under one second on available hardware. Moreover, the run time of our algorithm is linear in the size of the input program; this is crucial for scalable optimizations. The algorithm improves performance dramatically for several benchmark programs without sacrificing performance on any.

The inlining algorithm performs copy propagation, constant propagation, constant folding, useless or unreachable code elimination, and procedure inlining in a single pass. The algorithm is online in that analysis and transformation are intertwined. Because the various optimizations are performed concurrently, inlining decisions are often based on more accurate information than would be available in an offline approach. Polyvariance is obtained by attempting to inline all calls while aborting any attempts that result in excessive code growth or processing time. The inlining algorithm is aggressive because even large procedures are candidates for inlining, yet prudent because no procedure is inlined unless the size of the resulting specialized code is below an adjustable threshold. To improve the speed and accuracy of the non-iterative analysis component, call operands and variable bindings are processed on demand. Often this means that an operand or binding can be processed in a more restricted context that permits a greater degree of optimization. The run time of the algorithm and the size of the residual code are controlled by a no-nonsense mechanism that directly regulates the processes that contribute to increased compile time and increased code size. A cycle detection mechanism enables the algorithm to identify recursive procedures and treat them specially.

To show that our algorithm is both fast and effective, we compare our results with those obtained using state-of-the-art flow-directed inlining algorithms. Comparing the run time of programs before and after inlining shows that our algorithm yields competitive (and often better) performance gains. Our algorithm consistently produces better code size ratios than these offline algorithms. This supports our claim that online transformation provides our algorithm with more accurate information about the size of the residual code, permitting tighter control over code growth. Finally, the total run time of our algorithm (including analysis and transformation) is consistently lower than the run time of the flow analyses (which does not include subsequent transformation) employed by the offline algorithms.

Because it is both fast and effective, the algorithm is well-suited for use in a production compiler. The algorithm also provides a competent basis for comparison that can be used to isolate the additional benefit, if any, obtained through more involved analysis techniques.

5.2 Future Work

This dissertation represents the first step in an ambitious research program that seeks to make modular language design a reality. Compilers and run-time systems for new domain-specific languages may one day be constructed from well-structured modular language components. New language designs may reuse components of existing designs in a sound and principled manner. Our experience suggests an interesting conjecture: when a range of semantics is possible for a new language construct, such as modules, it is preferable to choose the most static and efficient semantics as the default and rely on the combination of higher-order functions and syntactic abstraction to provide more dynamic variants.

There are several practical issues to be resolved in future work. First, complex macros often need shared access to auxiliary functions and data structures. While the module system can easily be used to accomplish the necessary scoping relationships, it does not provide a mechanism for specifying that certain bindings are to be evaluated at compile time so that their values are available during macro expansion.[1] Second, separate compilation of top-level modules exporting macros introduces the potential for space leaks during compilation. It may be necessary to provide a mechanism for unloading a top-level module definition. If so, it is not clear what this means for anonymous modules. Finally, there is opportunity for optimization across module boundaries. Some flexibility in separate compilation is required to take full advantage of cross-module optimizations. For example, a limited form of the implicit export problem arises if we allow cross-module procedure inlining because nonexported variables may appear free in the body of the inlined procedure. The problem is simplified by the fact that all possible references or assignments to nonexported (neither explicitly nor implicitly exported) variables are apparent in the text of the code when it is compiled.

Another interesting direction for future research is finding a way to describe both static and dynamic program properties so that existing implementation components can be customized and

[1]The availability of expand-time value bindings is an issue only because our macro system is powerful enough to make it so. In particular, this functionality would not be missed if the macro system were limited to `syntax-rules` transformers.

combined to obtain realistic compilers and run-time systems for novel language designs. While it is certainly useful to encode type information within the module interface, we would like to generalize this to include other program properties and invariants. For example, in Section 3.4.8 we saw the need to convey a single assignment property to the compiler when implementing the `declare` and `install` macros. Similarly, it can be useful to indicate that a particular operation is commutative or associative, that a pair of operations is distributive, or that an operation will never reinstate its continuation. Given this information we would like simple ways to describe and combine modular program optimizations and ways to mechanically verify that these transformations preserve the various properties.

Bibliography

[1] Alfred D. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[3] Andrew W. Appel and Trevor Jim. Shrinking lambda expressions in linear time. To appear in *Journal of Functional Programming.*

[4] J. Michael Ashley. A practical and flexible flow analysis for higher-order languages. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 184–194, 1996.

[5] J. Michael Ashley. The effectiveness of flow analysis for inlining. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 99–111, 1997.

[6] J. Michael Ashley and R. Kent Dybvig. An efficient implementation of multiple return values in Scheme. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 140–149, July 1994.

[7] J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems*, 20(4):845–868, July 1998.

[8] J. Eugene Ball. Predicting the effects of optimization on a procedure body. *SIGPLAN Notices*, 14(8):214–220, August 1979. *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction.*

[9] Robert L. Bernstein. Producing good code for the case statement. *Software – Practice and Experience*, 15(10):1021–1024, 1985.

[10] Bruno Blanchet. Escape analysis: Correctness proof, implementation, and experimental results. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 25–37, 1998.

[11] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16(1-3):151–195, 1991.

[12] Timothy A. Budd. *Multiparadigm Programming in Leda*. Addison-Wesley, 1995.

[13] Robert G. Burger. *Efficient Compilation and Profile-Driven Recompilation in Scheme*. PhD thesis, Indiana University, 1997.

[14] Robert G. Burger and R. Kent Dybvig. An infrastructure for profile-driven dynamic recompilation. In *IEEE Computer Society 1998 International Conference on Computer Languages*, pages 240–251, May 1998.

[15] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1):188–222, Jan 1997.

[16] William Clinger and Lars Thomas Hansen. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 128–139, 1994.

[17] William Clinger and Jonathan Rees. Macros that work. In *Conference Record of the Seventeenth Annaual ACM Symposium on Principles of Programming Languages*, pages 155–162, January 1991.

[18] Patrick Cousot and Rhadia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[19] Pavel Curtis and James Rauen. A module system for Scheme. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 13–19, June 1990.

[20] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[21] Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: Transforming recursive equations into programs with block structure. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 90–106, June 1997.

[22] Harley Davis, Pierre Parquier, and Nitsan Séniak. Talking about modules and delivery. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 113–120, 1994.

[23] Jeffrey Dean and Craig Chambers. Towards better inlining decisions using inlining trials. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 273–282, 1994.

[24] Alain Deutsch. On the complexity of escape analysis. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 358–371, 1997.

[25] Julian Dolby. Automatic inline allocation of objects. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 7–17. ACM Press, 1997.

[26] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, 1987.

[27] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, second edition, 1996.

[28] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-passing style: A general macro mechanism. *Lisp and Symbolic Computation*, 1(1):53–75, 1988.

[29] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1993.

[30] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 94–104, September 1998.

[31] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 236–248, June 1998.

[32] M. Hanus. A unified computation model for functional and logic programming. In *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 80–93, 1997.

[33] Robert Harper, Robin Milner, and Mads Tofte. The definition of Standard ML. Technical Report ECS-LFCS-89-81, Department of Computer Science, University of Edinburgh, 1989.

[34] John L. Hennessy and Noah Mendelsohn. Compilation of the Pascal case statement. *Software – Practice and Experience*, 12(9):879–882, 1982.

[35] Suresh Jagannathan and Andrew Wright. Flow-directed inlining. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 193–205, 1996.

[36] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In J.-P. Jouannaud, editor, *Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203. Springer-Verlag, 1985.

[37] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[38] Simon L. Peyton Jones and John Hughes (editors). Haskell 98: A non-strict, purely functional language. Technical report, February 1999.

[39] Sampath Kannan and Todd A. Proebsting. Correction to 'producing good code for the case statement'. *Software – Practice and Experience*, 24(2):233, 1994.

[40] Owen Kaser, C. R. Ramakrishnan, and Shaunak Pawagi. On the conversion of indirect to direct recursion. *ACM Letters on Programming Languages and Systems*, 2(4):151–164, mar 1993.

[41] Richard Kelsey, William Clinger, and Jonathan A. Rees (Editors). Revised^5 report on the algorithmic language Scheme. *SIGPLAN Notices*, 33(9):26–76, 1998.

[42] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[43] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 151–161, 1986.

[44] Eugene E. Kohlbecker. *Syntactic Extensions in the Programming Language Lisp*. PhD thesis, Indiana University, Bloomington, Indiana, 1986.

[45] David MacQueen. Modules for standard ML. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 198–207, 1984.

[46] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[47] Martin Müller, Tobias Müller, and Peter Van Roy. Multi-paradigm programming in Oz. In Donald Smith, Olivier Ridoux, and Peter Van Roy, editors, *Visions for the Future of Logic Programming: Laying the Foundations for a Modern successor of Prolog*, Portland, Oregon, 7 December 1995. A Workshop in Association with ILPS'95.

[48] David A. Patterson and John L. Hennessy. *Computer Organizations and Design: the Hardware/Software Interface*. Morgan Kaufmann, 1998.

[49] Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 67–78. ACM, 1995.

[50] Christian Queinnec and Julian Padget. A deterministic model for modules and macros. Bath Computing Group Technical Report 90-36, University of Bath, Bath (UK), 1990.

[51] Christian Queinnec and Julian Padget. Modules, macros and Lisp. In *Eleventh International Conference of the Chilean Computer Science Society*, pages 111–123, Santiago (Chile), October 1991. Plenum Publishing Corporation, New York NY (USA).

[52] Jonathan Rees. Modular macros. Master's thesis, Massachusetts Institute of Technology, May 1989.

[53] Jonathan Rees. *Another Module System for Scheme.* Massachusetts Institute of Technology, 1994. Scheme 48 documentation.

[54] Arthur Sale. The implementation of case statements in Pascal. *Software – Practice and Experience*, 11(9):929–942, 1981.

[55] Manuel Serrano. Inline expansion: when and how? In *Programming Languages: Implementations, Logics, and Programs, 9th International Symposium*, volume 1292 of *Lecture Notes in Computer Science*, pages 143–157, September 1997.

[56] Andrew Shalit. *The Dylan Reference Manual.* Addison Wesley Longman, 1996.

[57] Olin Shivers. A Scheme shell. *Lisp and Symbolic Computation.* to appear.

[58] Olin Shivers. A universal scripting framework, or Lambda: the ultimate 'little language'. In Joxan Jaffar and Roland H. C. Yap, editors, Concurrency and Parallelism, Programming, Networking, and Security, volume 1179 of *Lecture Notes in Computer Science*, pages 254–265. Springer-Verlag, 1996.

[59] Richard M. Stallman. Using and porting GNU CC. Technical report, Free Software Foundation, 1992.

[60] Guy L. Steele Jr. *Common Lisp, the Language.* Digital Press, second edition, 1990.

[61] Sho-Huan Simon Tung and R. Kent Dybvig. Reliable interactive programming with modules. *Lisp and Symbolic Computation*, 9(4):343–358, 1996.

[62] Úlfar Erlingsson, Mukkai Krishnamoorthy, and T.V. Raman. Efficient multiway radix search trees. *Information Processing Letters*, 60:115–120, 1996.

[63] Oscar Waddell and J. Michael Ashley. Visualizing the performance of higher-order programs. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 75–82. SIGPLAN Notices, June 1998.

[64] Oscar Waddell and R. Kent Dybvig. Visualizing partial evaluation. *ACM Computing Surveys, Symposium on Partial Evaluation*, 30(3es):24–es, September 1998.

[65] Mitchell Wand and Paul Steckler. Selective and lightweight closure conversion. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 435–445, 1994.

[66] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 3(2):181–210, 1991.

Curriculum Vitae

Oscar Waddell received a Bachelor of Arts degree, with Honors in Computer Science, from Oberlin College in 1990. From 1990 through 1993, he was an Associate Instructor in the Computer Science Department at Indiana University. From 1993 until 1998, he was a Research Assistant at Indiana University. During that time, he also performed contract and subcontract work on a commercial implementation of the Scheme programming language. In 1998, he was a Postdoctoral Research Scientist at the University of Kansas. At the start of the 1999 academic year, he returned to Indiana University as a Research Assistant Professor in the Computer Science Department.