Higher-Order and Symbolic Computation, 18, 299–326, 2005

Fixing Letrec: A Faithful Yet Efficient Implementation of Scheme's Recursive Binding Construct*

OSCAR WADDELL owaddell@cs.indiana.edu
DIPANWITA SARKAR dsarkar@cs.indiana.edu
R. KENT DYBVIG dyb@cs.indiana.edu
Indiana University, Bloomington, Indiana, USA

Abstract. A Scheme `letrec` expression is easily converted into more primitive constructs via a straightforward transformation given in the Revised^5 Report. This transformation, unfortunately, introduces assignments that can impede the generation of efficient code. This article presents a more judicious transformation that preserves the semantics of the revised report transformation and also detects invalid references and assignments to left-hand-side variables, yet enables the compiler to generate efficient code. A variant of `letrec` that enforces left-to-right evaluation of bindings is also presented and shown to add virtually no overhead.

Keywords: letrec restriction, recursive binding construct, internal definitions, modules, mutual recursion, optimization, Scheme

1. Introduction

Scheme's `letrec` is a recursive binding form that permits the definition of mutually recursive procedures and, more generally, mutually recursive values that contain procedures. It may also be used to bind variables to arbitrary nonrecursive values, and a single `letrec` expression is often used for both purposes. This is especially common when `letrec` is used as an intermediate-language representation for internal variable definitions and local modules [14].

A `letrec` expression has the form

(`letrec` ([x_1 e_1] ... [x_n e_n]) *body*)

where each x is a variable and each e is an arbitrary expression, often but not always a `lambda` expression.

The Revised^5 Report on Scheme [7] defines `letrec` via the following transformation into more primitive constructs, where t_1 ... t_n are fresh temporaries.

*A preliminary version of this article was presented at the 2002 Workshop on Scheme and Functional Programming [15].

```
(letrec ([x1 e1] ... [xn en]) body)
→ (let ([x1 undefined] ... [xn undefined])
    (let ([t1 e1] ... [tn en])
      (set! x1 t1)
      ...
      (set! xn tn))
    body)
```

This transformation effectively defines the meaning of `letrec` operationally; a `letrec` expression (1) binds the variables $x_1 \ldots x_n$ to new locations, each holding an "undefined" value, (2) evaluates the expressions $e_1 \ \ldots \ e_n$ in some unspecified order, (3) assigns the variables to the resulting values, and (4) evaluates the body. The expressions $e_1 \ \ldots \ e_n$ and *body* are all evaluated in an environment that contains the bindings of the variables, allowing the values to be mutually recursive.

The revised report imposes an important restriction on the use of `letrec`: it must be possible to evaluate each of the expressions $e_1 \ \ldots \ e_n$ without evaluating a reference or assignment to any of the variables $x_1 \ \ldots \ x_n$. References and assignments to these variables may appear in the expressions, but they must not be evaluated until after control has entered the body of the `letrec`. We refer to this as the "`letrec` restriction." The revised report states that "it is an error" to violate this restriction. This means that the behavior is unspecified if the restriction is violated. While implementations are not required to signal such errors, doing so is desirable. The transformation given above does not directly detect violations of the `letrec` restriction. It does, however, imply a mechanism whereby violations can be detected, i.e., a check for the *undefined* value can be inserted before each reference or assignment to any of the left-hand-side variables that occurs within a right-hand side expression.

The revised report transformation of `letrec` faithfully implements the semantics of `letrec` as described in the report, and it permits an implementation to detect violations of the `letrec` restriction. Yet, many of the assignments introduced by the transformation are unnecessary, and the obvious error detection mechanism inhibits copy propagation and inlining for `letrec`-bound variables.

A theoretical solution to these problems is to restrict `letrec` so that its left-hand sides are unassigned and right-hand sides are `lambda` expressions. We refer to this form of `letrec` as `fix`, since it amounts to a generalized form of fixpoint operator. The compiler can handle `fix` expressions efficiently, and there can be no violations of the `letrec` restriction with `fix`. Unfortunately, restricting `letrec` in this manner is not an option for the implementor and would in any case reduce the generality and convenience of the construct.

This article presents an alternative to the revised report transformation of full `letrec` that attempts to produce `fix` expressions covering as many `letrec` bindings as possible while falling back to the use of assignments where necessary. In essence, the alternative transformation "`fix`es" `letrec` without breaking it. This enables the compiler to generate efficient code while preserving the semantics of the revised report transformation. The transformation is shown to eliminate most of the introduced assignments and to improve

run time dramatically. The transformation also incorporates a mechanism for detecting all violations of the `letrec` restriction that, in practice, has virtually no overhead.

This article investigates as well the implementation of a variant of `letrec`, which we call `letrec*`, that evaluates the right-hand sides from left to right and assigns each left-hand side immediately to the value of the right-hand side. It is often assumed that this would result in less efficient code; however, we show that this is not the case when our transformation is used. While fixed evaluation order often results in overspecification and is thus generally undesirable, `letrec*` would be a useful addition to the language and a reasonable intermediate representation for internal variable definitions, where left-to-right evaluation is often expected anyway.

The remainder of this article is organized as follows. Section 2 describes our transformation in three stages, starting with a basic version, adding an assimilation mechanism for nested bindings, and adding validity checks to detect violations of the `letrec` restriction. Section 3 introduces the `letrec*` form and describes its implementation. Section 4 presents an analysis of the effectiveness of the various transformations. Section 5 describes related work. Section 6 summarizes the article and presents our conclusions. A formal description of the basic `letrec` transformation with validity checks is presented in an Appendix.

2. The transformation

The transformation of `letrec` is developed in three stages. Section 2.1 describes the basic transformation. Section 2.2 describes a more elaborate transformation that assimilates `let` and `letrec` bindings nested on the right-hand side of a `letrec` expression. Section 2.3 describes how violations of the `letrec` restriction are detected.

The transformation expects that bound variables in the input program are uniquely named. It also assumes that an earlier pass of the compiler has recorded information about references and assignments of the bound variables. In our implementation, these conditions are met by running input programs through the `syntax-case` macro expander [4]. If this were not the case, a simple flow-insensitive pass to perform alpha conversion and record reference and assignment information could be run prior to the transformation algorithm. Variables are considered *referenced* if the variable might be referenced and *assigned* if the variable might be assigned. A straightforward conservative approximation is to consider a variable referenced (assigned) if a reference (assignment) appears anywhere within its scope.

The transformation is implemented in two passes. The first introduces the code that detects violations of the `letrec` restriction, and the second performs the basic transformation and assimilation. The order of the two passes is important, since the second pass performs code motion that may disguise or eliminate errors that the first pass is designed to catch. We describe the basic transformation first.

2.1. Basic transformation

Each `letrec` expression (`letrec ([`*x e*`] ...)` *body*`)` in the input program is converted as follows.

1. The expressions e ... and *body* are converted to produce e' ... and *body*$'$.
2. The bindings [x e'] ... are partitioned into several sets:

   [x_u e_u] ... *unreferenced*
   [x_s e_s] ... *simple*
   [x_l e_l] ... *lambda*
   [x_c e_c] ... *complex*

3. A set of nested `let` and `fix` expressions is formed from the partitioned bindings:

```
(let ([x_s e_s] ... [x_c (void)] ...)
  (fix ([x_l e_l] ...)
    e_u ...
    (let ([x_t e_c] ...)
      (set! x_c x_t)
      ...)
    body'))
```

   where x_t ... is a set of fresh temporaries, one per x_c. The innermost `let` is produced only if [x_c e_c] ... is nonempty. The expressions e_u ... are retained for their effects.

4. Because the bindings for unreferenced `letrec`-bound variables are dropped, assignments to unreferenced variables are also dropped wherever they appear.

   `(set!` x_u e`)` $\rightarrow e$

During the partitioning phase, a binding [x e'] is considered

unreferenced if x is unreferenced, else
simple if x is unassigned and e' is a simple expression, else
lambda if x is unassigned and e' is a lambda expression, and
complex if it does not fall into any of the other categories.

A simple expression contains no occurrences of the variables bound by the `letrec` expression and must not be able to obtain its continuation via `call/cc`, either directly or indirectly. The former restriction is necessary because simple expressions are placed outside the scope of the bound variables. Without the latter restriction, it would be possible to detect the fact that the bindings are created after the evaluation of a simple right-hand-side expression rather than before. To enforce the latter restriction, our implementation simply rules out all procedure calls except those to certain primitives (not including `call/cc`) when primitive calls can be recognized as such by the compiler. In fact, our implementation actually considers simple only literals, `quote` expressions, references to bound variables other than the left-hand-side variables, `if` expressions with simple subexpressions, `begin` expressions with simple subexpressions, and calls to effect-free primitives with simple argument expressions. We rule out effects for reasons to be discussed later, and although we could work harder to uncover more simple expressions, these cases appear to catch nearly all simple expressions in practice without much compile-time overhead.

The transformation converts `letrec` expressions into an equivalent mix of `let`, `set!`, and `fix` expressions. A `fix` expression is a variant of `letrec` that binds only unassigned

variables to `lambda` expressions. It represents a subset of `letrec` expressions that can be handled easily by later passes of a compiler. In particular, no assignments through external variables are necessary to implement mutually recursive procedures bound by `fix`. Instead, the closures produced by a `fix` expression can be block allocated and "wired" directly together. This leaves the `fix`-bound variables unassigned, thus simplifying optimizations such as inlining and loop recognition. `fix` is identical to the `labels` construct handled by Steele's Rabbit compiler [13] and the `Y` operator of Kranz's Orbit compiler [8, 9] and Rozas' Liar compiler [11, 12].

The output expression also includes calls to `void`, a primitive that evaluates to some "unspecified" value. It may be defined as follows.

```
(define void (lambda () (if #f #f)))
```

We do not use a special "undefined" value; instead, we use a different mechanism for detecting violations of the `letrec` restriction, as described in Section 2.3.

An unreferenced right-hand side e' may be dropped if e' is a `lambda` expression or is simple and effect-free. This does not affect the code produced by our compiler since a later pass already eliminates such expressions when they are used only for effect.

2.2. *Assimilating nested binding forms*

When a `letrec` right-hand side is a `let` or `letrec` expression, the partitioning described above treats it as *complex*. For example,

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

is translated into

```
(let ([f (void)])
  (let ([f_t  (let ([g (void)])
                (let ([g_t  (let ([x 5])
                              (lambda () ...))])
                  (set! g g_t))
                (lambda () ... g ...))])
    (set! f f_t))
  f)
```

This is unfortunate, since it penalizes programmers who use nested `let` and `letrec` expressions in this manner to express scoping relationships more tightly.

We would prefer a translation into the following equivalent expression.

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

(Since we expect variables to be uniquely named, moving the binding of `x` out causes no scoping problems.)

Therefore, the actual partitioning used is a bit more complicated. When a binding `[`x e'`]` fits immediately into one of the first three categories, the rules above suffice. The exception to these rules occurs when x is unassigned and e' is a `let` or `fix` binding, in which case the transformer attempts to fold the nested bindings into the partitioned sets. This leads to fewer introduced assignments and more direct-call optimizations in later passes of the compiler.

When e' is a `fix` expression `(fix (`$[\hat{x}_l\ \hat{e}_l]\ \ldots$`)` $\widehat{body}$`)`, the bindings $[\hat{x}_l\ \hat{e}_l]\ \ldots$ are simply added to the *lambda* partition, and the binding `[`x $\widehat{body}$`]` is added to the set of bindings to be partitioned. Essentially, this transformation treats the nested bindings as if they had originally appeared in the enclosing `letrec`. For example,

`(letrec ([f` e_f`] [g (letrec ([a` e_a`]) ` e_g`)] [h` e_h`]) ` *body*`)`

is effectively treated as the following.

`(letrec ([f` e_f`] [g` e_g`] [a` e_a`] [h` e_h`]) ` *body*`)`

When e' is a `let` expression `(let (`$[\hat{x}\ \hat{e}]\ \ldots$`)` $\widehat{body}$`)` and the set of bindings $[\hat{x}\ \hat{e}]\ \ldots$ can be fully partitioned into a set of *simple* bindings $[\hat{x}_s\ \hat{e}_s]\ \ldots$ (which must reference neither $\hat{x}\ \ldots$ nor the left-hand-side variables of the enclosing `letrec`) and a set of *lambda* bindings $[\hat{x}_l\ \hat{e}_l]\ldots$, we add $[\hat{x}_s\ \hat{e}_s]\ \ldots$ to the *simple* partition, $[\hat{x}_l\ \hat{e}_l]\ \ldots$ to the *lambda* partition, and `[`x $\widehat{body}$`]` to the set of bindings to be partitioned.

For example, when e_a is a `lambda` or simple expression,

`(letrec ([f` e_f`] [g (let ([a` e_a`]) ` e_g`)] [h` e_h`]) ` *body*`)`

is treated as the following.

`(letrec ([f` e_f`] [g` e_g`] [a` e_a`] [h` e_h`]) ` *body*`)`

If, during this process, we encounter a binding `[`$\hat{x}$ $\hat{e}$`]` where $\hat{x}$ is unassigned and $\hat{e}$ is a `let` or `fix` expression, or if we find that the body is a `let` or `fix` expression, we simply fold the bindings in and continue with the assimilation attempt.

While Scheme allows the right-hand sides of a binding construct to be evaluated in any order, the order used must not involve (detectable) interleaving of evaluation. For possibly assimilated bindings, the definition of *simple* must therefore be modified to preclude effects. Otherwise, the effects caused by the bindings and body of an assimilated `let` could be separated, producing a detectable interleaving of the assimilated `let` with the other expressions bound by the outer `letrec`.

One situation not handled by the transformation just described is the following, in which a local binding is used to hold a counter or other similar piece of state.

```
(letrec ([f (let ([n 0])
              (lambda ()
                (set! n (+ n 1))
                n))])
  body)
```

We must not assimilate in such cases if doing so would detectably separate the creation of the (mutable) binding from the evaluation of the nested `let` body. In the example above, however, the separation cannot be detected, since the body of the nested `let` is a `lambda` expression, and assimilated bindings of `lambda` expressions are evaluated only once.

To avoid penalizing such uses of local state, we refine the partitioning algorithm. When e' is a `let` expression `(let ([`$\hat{x}$ $\hat{e}$`] ...)` $\widehat{body}$`)` and the set of bindings `[`$\hat{x}$ $\hat{e}$`] ...` can be fully partitioned into a set of *simple* bindings `[`$\hat{x}_s$ $\hat{e}_s$`] ...` and a set of *lambda* bindings `[`$\hat{x}_l$ $\hat{e}_l$`] ...`, except that one or more of the variables $\hat{x}_s$ ... is assigned, and $\widehat{body}$ is a `lambda` expression, we add `[`$\hat{x}_s$ $\hat{e}_s$`] ...` to the *simple* partition, `[`$\hat{x}_l$ $\hat{e}_l$`] ...` to the *lambda* partition, and `[`x $\widehat{body}$`]` to the set of bindings to be partitioned.

The `let` and `fix` expressions produced by recursive transformation of a `letrec` expression can always be assimilated if they have no complex bindings. In particular, the assimilation of `fix` expressions in the intermediate language effectively implements the assimilation of pure `letrec` expressions in the source language.

2.3. Validity checks

According to the Revised[5] Report, it must be possible to evaluate each of the expressions e_1 ... e_n in

`(letrec ([`x_1 e_1`] ... [`x_n e_n`]) `*body*`)`

without evaluating a reference or assignment to any of the variables x_1 ... x_n. This is the `letrec` restriction first mentioned in Section 1.

The revised report states that "it is an error" to violate this restriction. Implementations are not required to signal such errors; the behavior is left unspecified. An implementation may instead assign a meaning to the erroneous program. Older versions of our system "corrected" erroneous programs such as the following.

```
(letrec ([x 1] [y (+ x 1)]) (list x y))⇒(1 2)
(letrec ([y (+ x 1)] [x 1]) (list x y))⇒(1 2)
```

While this may seem appealing at first, we believe an implementation should detect and report language violations rather than giving meaning to technically meaningless programs, since any meaning we assign may not be the one intended. Reporting language violations also helps users create more portable programs. Fortunately, violations of the `letrec` restriction can be detected with practically no overhead, as we describe in this section.

It is possible to detect violations of the `letrec` restriction by binding each left-hand-side variable initially to a special "undefined" value and checking for this value at each reference and assignment to the variable within the right-hand-side expressions. This approach results in many more checks than are actually necessary. More importantly, it makes all bindings complex, nullifying the advantages of the transformations described in Sections 2.1 and 2.2. This in turn may inhibit later passes from performing various optimizations such as inlining and copy propagation.

It is possible to analyze the right-hand sides to determine the set of variables referenced or to perform an interprocedural flow analysis to determine the set of variables that

might be undefined when referenced or assigned, by monitoring the flow of the undefined values. With this information, we could perform the transformations described in Sections 2.1 and 2.2 for all but those variables that might be undefined when referenced or assigned.

We use a different approach that never inhibits our transformations and thus does not inhibit optimization of `letrec`-bound variables merely because they may be undefined when referenced or assigned. Our approach is based on two observations: (1) a separate boolean variable may be used to indicate the validity of a `letrec` variable, and (2) we need just one such variable per `letrec`. The latter observation holds since if evaluating a reference or assignment to one of the left-hand-side variables is invalid at a given point, evaluating a reference or assignment to any of the left-hand-side variables is invalid at that point. With a separate valid flag, the transformation algorithm can do as it pleases with the original bindings.

This flag is introduced as a binding of a fresh variable, `valid?`, wrapped around the `letrec` expression. The flag is set initially to false, meaning that references to left-hand-side variables are not allowed, and changed to true just before control enters the body of the `letrec`, since each reference and assignment from that point on is valid, whether executed directly by the body, via a call to one of the `letrec`-bound variables, or via a continuation throw that returns control to one of the right-hand-side expressions.

```
(let ([valid? #f])
  (letrec ([x e] ...)
    (set! valid? #t)
    body))
```

A validity check simply tests `valid?` and signals an error if `valid?` is false.

```
(unless valid? (error 'x "undefined"))
```

Validity checks are inserted wherever the implementation deems them to be necessary. If no validity checks are deemed to be necessary, the valid-flag binding and assignment are suppressed. This is important even if a later pass of the compiler eliminates useless bindings and assignments, since the presence of the assigned valid flag could inhibit assimilation by the second pass of the transformation.

In a naive implementation, validity checks would be inserted at each reference and assignment to one of the left-hand-side variables within the right-hand-side expressions. No checks need to be inserted in the body of the `letrec`, since the bindings are necessarily valid once control enters the body.

We can improve upon this by suppressing checks within a right-hand-side expression if that expression is a `lambda` expression. Control cannot enter the body of the `lambda` expression before the valid flag is set to true except by way of a (checked) reference to the corresponding left-hand-side variable. One implication of this is that no valid flag or checks are necessary if all right-hand-side expressions are `lambda` expressions.

More generally, validity checks need not be inserted into the body of a `lambda` expression appearing in one of the right-hand-side expressions if we can prove that the resulting procedure cannot be invoked before control enters the body of the `letrec`. To handle the general case, we introduce the notion of *protected* and *unprotected* references. A reference

(or assignment) to a variable is protected if it is contained within a `lambda` expression that cannot be evaluated and invoked during the evaluation of an expression. Otherwise, it is unprotected.

Valid flags and checks are introduced during the first pass of the transformation algorithm, which operates on the original source program, i.e., before the transformation into `fix`, `let`, and `set!` forms as described earlier.

The pass uses a top-down recursive-descent algorithm. While processing the right-hand sides of a `letrec`, the left-hand-side variables of the `letrec` are considered to be in one of three states: *protected*, *protectable*, or *unprotected*. A variable is protectable if references and assignments found within a `lambda` expression are *safe*, i.e., if the `lambda` expression cannot be evaluated and invoked before control enters the body of the `letrec`. Each variable starts out in the protectable state when processing of the right-hand-side expression begins.

Upon entry into a `lambda` expression, all protectable variables are moved into the protected state, since they cannot possibly require validity checks. Upon entry into an *unsafe* context, i.e., one that might result in the evaluation and invocation of a `lambda` expression, the protectable variables are moved into the unprotected state. This occurs, for example, while processing the arguments to an unknown procedure, since that procedure might invoke the procedure resulting from a `lambda` expression appearing in one of the arguments.

For each variable reference and assignment, a validity check is inserted for the protectable and unprotected variables but not for the protected variables.

This handles well situations such as

```
(letrec ([x 0]
         [f (cons (lambda () x)
                  (lambda (v) (set! x v)))])
  body)
```

in which `f` is a sort of locative [10] for `x`. Since `cons` does not invoke its arguments, the references appearing within the `lambda` expressions are protected.

Handling situations such as the following is more challenging.

```
(letrec ([x 0]
         [f (let ([g (lambda () x)])
              (lambda () (g)))])
  body)
```

In general, we must treat the right-hand side of a `let` binding as unsafe, since the left-hand-side variable may be used to invoke procedures created by the right-hand-side expression. In this case, however, the body of the `let` is a `lambda` expression, so there is no problem. To handle this situation, we also record for each `let`- and `letrec`-bound variable whether it is protectable or unprotected and treat the corresponding right-hand side as an unsafe or safe context depending upon whether the variable is referenced or not. For `letrec` this involves a demand-driven process, starting with the body of the `letrec` and proceeding with the processing of any unsafe right-hand sides.

1. Process the body of the `letrec` with the left-hand-side variables in the protectable state. As the body is processed, mark unsafe the right-hand side of any binding whose left-hand-side variable is referenced in a protectable or unprotected state.
2. If any right-hand side has been marked unsafe, process it with each of the outer protectable variables, i.e., protectable variables bound by enclosing `letrec` expressions, in the unprotected state and each of the left-hand-side variables in the protectable state. As each right-hand side is processed, mark unsafe the right-hand side of any binding whose left-hand-side variable is referenced, and insert valid checks for any left-hand-side variable referenced in a protectable or unprotected state. Repeat this step until no unprocessed unsafe right-hand sides remain.
3. Process the remaining right-hand sides with the left-hand-side variables in the protectable state. As each right-hand side is processed, insert valid checks for any left-hand-side variable referenced in a protectable or unprotected state.

3. Fixed evaluation order

The Revised[5] Report translation of `letrec` is designed so that the right-hand-side expressions are all evaluated before the assignments to the left-hand-side variables are performed. The transformation for `letrec` described in the preceding section loosens this structure, but in a manner that cannot be detected, because an error is signaled for any program that prematurely references one of the left-hand-side variables and because the lifted bindings are immutable and cannot be (detectably) reset by a continuation invocation.

From a software engineering perspective, the unspecified order of evaluation is valuable because it allows the programmer to express lack of concern for the order of evaluation. That is, when the order of evaluation of two expressions is unspecified, the programmer is, in effect, saying that neither counts on the other being done first. From an implementation standpoint, the freedom to determine evaluation order may allow the compiler to generate more efficient code.

It is sometimes desirable, however, for the values of a set of `letrec` bindings to be established in a particular order. This seems to occur most often in the translation of internal variable definitions into `letrec`. For example, one might wish to define a procedure and use it to produce the value of a variable defined further down in a sequence of definitions.

```
(define f (lambda ...))
(define a (f ...))
```

This would, however, violate the `letrec` restriction. One can nest binding contours to order bindings, but nesting cannot be used for mutually recursive bindings and is inconvenient in other cases.

It is therefore interesting to consider a variant of `letrec` that performs its bindings in a left-to-right fashion. Scheme provides a variant of `let`, called `let*`, that sequences evaluation of `let` bindings; we therefore call our version of `letrec` that sequences `letrec` bindings `letrec*`. The analogy to `let*` is imperfect, however, since `let*` also nests scopes whereas `letrec*` maintains the mutually recursive scoping of `letrec`.

`letrec*` can be transformed into more primitive constructs in a manner similar to `letrec` using a variant of the Revised[5] Report transformation of `letrec`.

```
(letrec* ([x1 e1] ... [xn en]) body)
 →  (let ([x1 undefined] ... [xn undefined])
      (set! x1 e1)
      ...
      (set! xn en)
      body)
```

This transformation is actually simpler, in that it does not include the inner `let` that binds a set of temporaries to the right-hand-side expressions. This transformation would be incorrect for `letrec`, since the assignments are not all in the continuation of each right-hand-side expression, as in the revised report transformation. That is, `call/cc` could be used to expose the difference between the two transformations.

The basic transformation given in Section 2.1 is also easily modified to implement the semantics of `letrec*`. As before, the expressions $e \ \ldots$ and *body* are converted to produce $e' \ \ldots$ and *body*′, and the bindings are partitioned into *simple*, *lambda*, *unreferenced*, and *complex* sets. One difference comes in the structure of the output code. If there are no *unreferenced* bindings, the output is as follows

```
(let ([xs es] ... [xc (void)] ...)
  (fix ([xl el] ...)
    (set! xc ec)
    ...
    body′))
```

where the assignments to x_c are ordered as the bindings appeared in the original input.

If there are *unreferenced* bindings, the right-hand sides of these bindings are retained, for effect only, among the assignments to the *complex* variables in the appropriate order.

The other differences involve *simple* bindings with effects. A right-hand-side expression cannot be considered *simple* if it has effects and follows any *complex* or *unreferenced* right-hand side with effects in the original `letrec*` expression. Furthermore, the original order of *simple* bindings with effects must be preserved. This can be accomplished by producing a set of nested `let` expressions in the output to preserve their ordering. Our implementation currently takes an easier approach to solving both problems, which is to treat as *complex* any otherwise simple binding whose right-hand side is not effect free.

The more elaborate partitioning of `letrec` expressions to implement assimilation of nested bindings as described in Section 2.2 is compatible with the transformation above, so the implementation of `letrec*` does not inhibit assimilation.

On the other hand, a substantial change to the introduction of valid flags is necessary to handle the different semantics of `letrec*`. This change is to introduce (at most) one valid flag for each `letrec*` binding, in contrast with (at most) one per `letrec` expression. The valid flag for a given variable represents the validity of references and assignments to that variable.

This may result in the introduction of more valid flags but should not result in the introduction of any additional validity checks. Due to the nature of `letrec*`, in fact, there may be fewer validity checks and possibly fewer actual valid-flag bindings.

The processing of `letrec*` by the check-insertion pass is also more complicated, because left-hand-side references within right-hand-side `lambda` expressions are not necessarily safe. For example, the apparently valid reference to `c` in the right-hand side of `a` is actually invalid because of the (valid) call to `a` on the right-hand side of `b`.

```
(letrec* ([a (lambda () c)]
          [b (a)]
          [c 7])
  b)
```

If this were `letrec` rather than `letrec*`, the reference to `a` on the right-hand side of `b` would be checked so that control would never reach the reference to `c`. We therefore process `letrec*` bindings as follows:

1. Each binding is processed in turn, with its left-hand-side variable and remaining left-hand-side variables, i.e., those for later bindings, in the unprotected state and a fresh dummy variable δ in the protectable state. Processing occurs as usual, except that:
   - References to left-hand-side variables are recorded (for Step 2).
   - If a `lambda` expression is encountered while δ is still in the protectable state, the `lambda` expression is marked *deferred* and not processed.
   - Any expression containing a deferred expression is also marked deferred. (A deferred expression may have both processed and deferred subexpressions.)
2. If the left-hand-side variable of any binding with deferred right-hand side has been marked referenced, process the deferred portions of the right-hand side with the left-hand-side variable and remaining left-hand-side variables in the unprotected state. Repeat this step until no referenced left-hand-side variables with deferred right-hand sides remain.
3. Process the `letrec*` body and deferred portions of the remaining elements of the deferred list in a manner analogous to the body and right-hand sides of a `letrec` expression, as described at the end of Section 2.

4. Results

We have implemented the complete algorithm described in Section 2 and incorporated it as two new passes into the *Chez Scheme* compiler. The first pass inserts the validity checks described in Section 2.3, and the second performs the transformations described in Sections 2.1 and 2.2. We have also implemented `letrec*` as described in Section 3 and a compile-time parameter (compiler flag) that allows internal variable definitions (including those within modules) to be expanded into `letrec*` rather than `letrec`.

benchmark	*nodes*	*description*
`matrix`	4,825	A program that tests whether a matrix is maximal among all matrices obtained by reordering rows and columns
`conform`	5,300	A program that manipulates lattices and partial orders
`splay`	5,337	A program that builds splay trees
`earley`	5,902	Earley's algorithm for generating parsers for context-free grammars
`peval`	6,182	A small Scheme partial evaluator
`em-fun`	6,230	EM clustering algorithm in functional style
`em-imp`	6,521	EM clustering algorithm in imperative style
`nbody`	7,247	A program that computes gravitational forces using the Greengard multipole algorithm
`nucleic-star`	7,361	Feeley's 3D nucleic acid structure determination program [5] with top-level wrapped in `letrec*`
`nucleic-sorted`	7,369	Same as nucleic-star, but wrapped in nested `letrec` expressions
`interpret`	7,636	A Scheme interpreter evaluating the takl [6] benchmark
`dynamic`	9,889	A dynamic type inferencer applied to itself
`lalr`	9,995	Ashley's LALR parser generator in mix of imperative and functional style
`texer`	13,733	Bruggeman's Scheme pretty-printer with TEX output
`similix`	28,657	Self-application of the Similix [1] partial evaluator
`ddd`	32,278	A hardware derivation system [2] deriving a Scheme machine [3]
`softscheme`	134,209	Wright's soft typer [16] checking its pattern matcher
`chezscheme`	285,710	Chez Scheme compiling itself

Figure 1. Benchmarks ordered by abstract syntax tree size, measured following macro expansion.

We measured the performance of the set of benchmark programs described in Figure 1 using versions of our compiler that perform the following transformations:

- the standard Revised^5 Report (R^{5}RS) transformation;
- a modified R^{5}RS transformation (which we call "easy") that treats "pure" (`lambda` only) `letrec` expressions as `fix` expressions and reverts to the standard transformation for the others;
- versions of R^{5}RS and "easy" with naive validity checks;
- our transformation with and without assimilation and with and without validity checks; and
- our transformation with and without assimilation and validity checks, treating all `letrec` expressions as `letrec*` expressions.

We have compared these systems along several dimensions, including numbers of introduced assignments, indirect references, and validity checks performed at run time, as well as run times, compile times, and code sizes. Indirect references are references to assigned variables for which our compiler creates shared locations; introduced indirect references are those that result when a transformation causes references to become indirect by converting an unassigned variable into an assigned variable. We also compared the systems according to the numbers of bindings classified as *lambda*, *complex*, *simple*, and *unreferenced*.

With the exception of `chezscheme`, the code for each benchmark is wrapped in a `module` form so that all definitions are internal, which gives the compiler more to work with. A few that were originally written using top-level definitions and relied on left-to-right evaluation of these definitions were edited slightly so that they could run successfully in all of the systems. For the `nucleic` benchmarks we did the editing after macro expansion, replacing the resulting top-level `letrec` expression with a `letrec*` expression (`nucleic-star`) or a set of nested `letrec` expressions (`nucleic-sorted`). We used a topological sort of the `letrec` bindings to help with the latter transformation.

Run times for each benchmark were determined by averaging three runs of n iterations, where n was determined during a separate calibration run to be the minimum number of runs necessary to push the run time over two seconds. Compile times were determined similarly. Code size was determined by recording the size of the actual code objects written to compiled files. The results are given in Figures 2–14. Programs in these figures are displayed in sorted order, with larger programs (in terms of abstract syntax tree size) following smaller ones.

Figure 2 demonstrates that our transformation is successful in reducing run-time overhead. Using the "easy" transformation to catch pure `letrec` expressions is also effective but not as effective. Figures 3 and 4 indicate the reason why: programs produced by our transformation execute far fewer introduced assignments and indirect references.

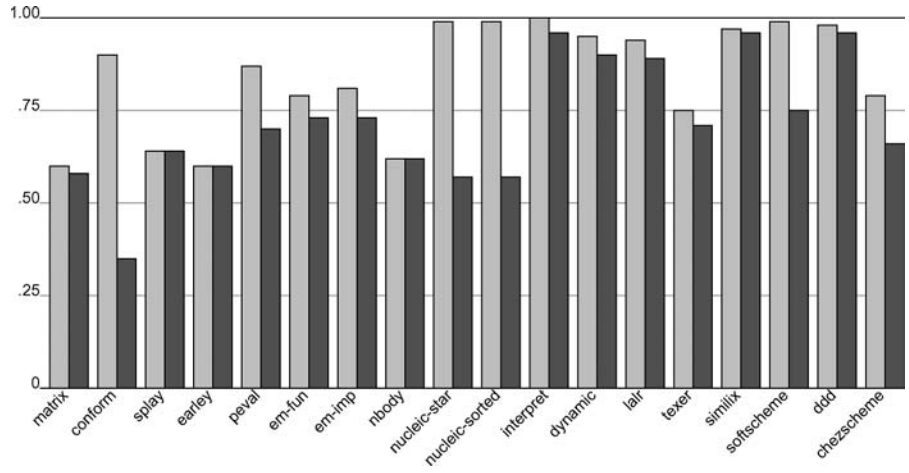

Figure 2. Run times of the code produced by the modified R^{5}RS transformation (easy) and our transformation, both normalized to the R^{5}RS run times.

	R^{5}RS	easy	ours
matrix	32,055	24	–
conform	193,149	64,622	–
splay	100,004	–	–
earley	51,756	–	–
peval	6,789	834	532
em-fun	28,755,774	61	–
em-imp	10,172,457	45	–
nbody	20,577,260	6	–
nucleic-star	9,060	181	5
nucleic-sorted	9,060	159	2
interpret	205	109	–
dynamic	1,632	173	1
lalr	6,697,716	129	1
texer	321,149	85	–
similix	152,141	463	–
ddd	423,841	439	8
softscheme	150,989	28,513	73
chezscheme	9,340,077	198,231	178

Figure 3. Numbers of introduced assignments executed dynamically by programs compiled using the R^{5}RS transformation, the modified R^{5}RS transformation (easy), and our transformation. Zero is denoted by " – " for readability.

	R^{5}RS	easy	ours
matrix	99,754	9,569	–
conform	3,642,137	3,406,487	–
splay	721,001	–	–
earley	262,440	–	–
peval	186,488	120,349	532
em-fun	129,217,011	62,106,671	–
em-imp	44,578,003	19,348,334	–
nbody	100,286,079	459,623	–
nucleic-star	8,162,206	8,068,422	7
nucleic-sorted	8,162,206	7,912,792	1
interpret	2,303,211	2,302,989	–
dynamic	425,146	266,048	39,929
lalr	302,232,257	52,775,817	297,917
texer	3,488,037	1,093,092	–
similix	10,926,477	8,392,639	–
ddd	12,534,905	6,433,532	384
softscheme	7,664,358	7,289,954	176,932
chezscheme	148,528,501	91,757,704	1,132,600

Figure 4. Numbers of introduced indirect references executed dynamically by programs compiled using the R^{5}RS transformation, the modified R^{5}RS transformation (easy), and our transformation.

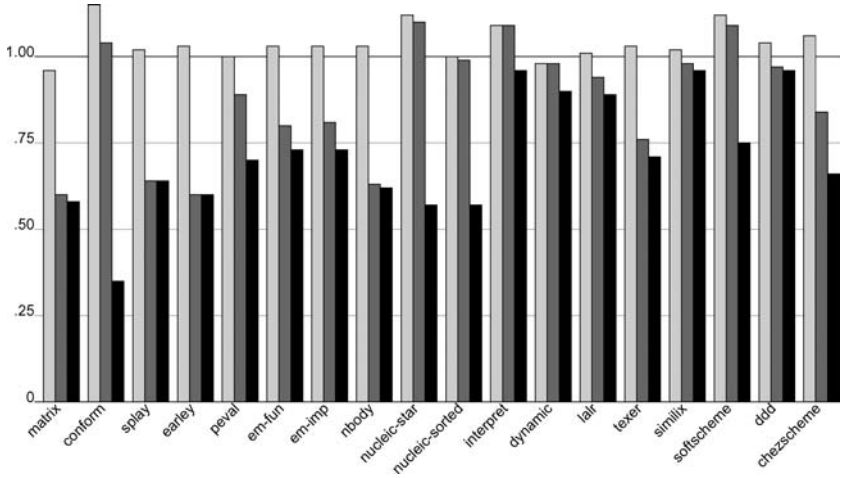

Figure 5. Run times of the code produced by the R[5]RS transformation with naive validity checks, the modified R[5]RS transformation (easy) with naive validity checks, and our transformation (with validity checks), all normalized to the R[5]RS (no validity checks) run times.

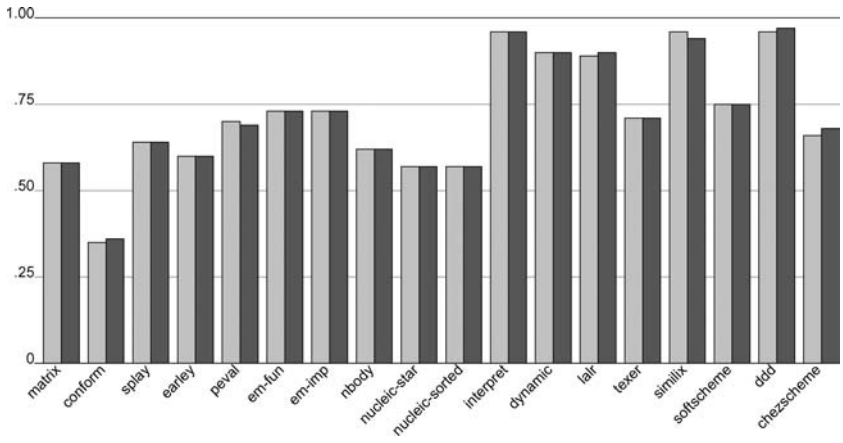

Figure 6. Run times of the code produced by our transformation, and our transformation without validity checks, both normalized to the R[5]RS run times.

Figure 5 shows that naive validity check insertion significantly reduces the performance of the R[5]RS and "easy" transformations in some cases. On the other hand, Figure 6 shows that run-times for our transformation are almost identical with and without validity checks, so our transformation achieves strict enforcement of the `letrec` restriction with practically no overhead. As indicated by Figure 7, naively enforcing the `letrec` restriction introduces far more validity checks than necessary, even when pure `letrec` expressions are recognized. Our transformation not only causes fewer validity checks to be performed (none in many cases) but also permits optimizations based on `letrec` bindings to proceed uninhibited with its use of separate valid flags.

	R^5RS	easy	ours
matrix	67,722	9,568	–
conform	3,489,951	3,382,561	–
splay	204,000	–	–
earley	210,641	–	–
peval	180,983	119,592	–
em-fun	100,464,744	62,110,117	–
em-imp	34,409,037	19,351,780	–
nbody	62,247,187	–	–
nucleic-star	350,309	265,404	256,307
nucleic-sorted	346,808	261,903	–
interpret	2,303,118	2,302,988	–
dynamic	336,101	265,934	–
lalr	290,745,438	51,786,365	–
texer	2,587,755	507,034	–
similix	5,259,949	2,885,779	–
ddd	12,111,948	6,437,306	7
softscheme	701,463	446,846	152,609
chezscheme	128,991,149	83,077,540	–

Figure 7. Numbers of validity checks executed dynamically by programs compiled using the R^5RS transformation, the modified R^5RS transformation (easy), and our transformation.

Figures 8 and 9 compare the performance of our algorithm with and without assimilation. For our compiler, the most substantial program in our test suite, assimilating nested bindings allows the transformation to decrease the number of introduced assignments and indirect references significantly. Assimilation does not seem to benefit run times, however. This is disappointing but may simply reflect the low overhead of the nonassimilating transformation, as evidenced by the already small number of introduced bindings and indirect references. Also, few of the benchmarks try to express scoping relationships more tightly, perhaps partly because programmers believe that the resulting code would be less efficient.

Figure 10 shows why our transformation is able to eliminate more assignments than the "easy" transformation. Our algorithm identifies "simple" bindings in many of the benchmarks and avoids introducing assignments for these. Moreover, it avoids introducing assignments for pure `lambda` bindings that happen to be bound by the same `letrec` that binds a simple binding. Figure 10 also shows that, in some cases, assimilating nested `let` and `letrec` bindings allows the algorithm to assign more of the bindings to the *lambda* or *simple* partitions.

Figures 11 and 12 show that fixing the order of evaluation has virtually no effect on run time or the number of introduced assignments, introduced indirect references, and validity checks, even though our compiler reorders expressions when possible to improve the generated code. This is likely due in part to the relatively few cases where constraints on the evaluation order remain following our translation of `letrec*`. We also measured the effect of enabling or disabling assimilation when `letrec` is replaced by `letrec*`; the results are virtually identical to those shown for `letrec` in Figures 8 and 9.

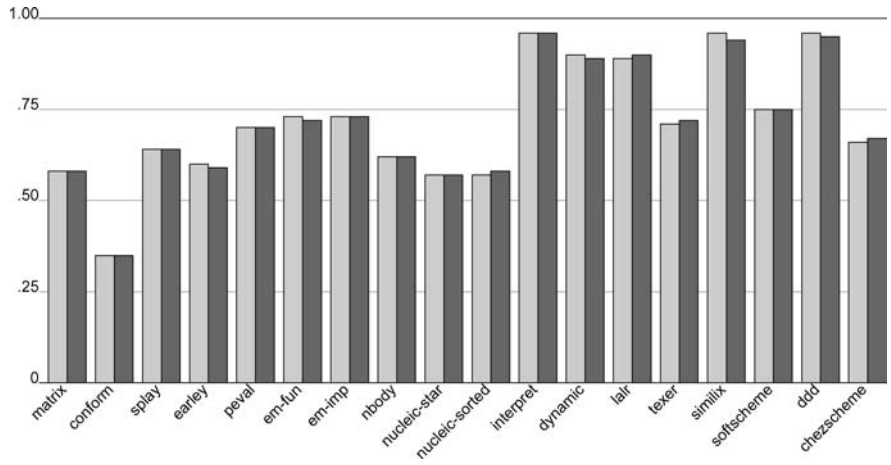

Figure 8. Run times of the code produced by our transformation and our transformation without assimilation, both normalized to the R^{5}RS run times.

	Assignments		References	
	Assimilating	Nonassimilating	Assimilating	Nonassimilating
matrix	–	2	–	4,533
conform	–	–	–	–
splay	–	–	–	–
earley	–	–	–	–
peval	532	532	532	532
em-fun	–	–	–	–
em-imp	–	–	–	–
nbody	–	–	–	–
nucleic-star	5	5	7	7
nucleic-sorted	2	2	1	1
interpret	–	–	–	–
dynamic	1	1	39,929	39,929
lalr	1	4	297,917	298,536
texer	–	5	–	34,448
similix	–	2	–	103
ddd	8	44	384	2,423
softscheme	73	75	176,932	176,932
chezscheme	178	419	1,132,600	2,280,902

Figure 9. Numbers of introduced assignments and indirect references executed dynamically by programs compiled using our transformation with and without assimilation.

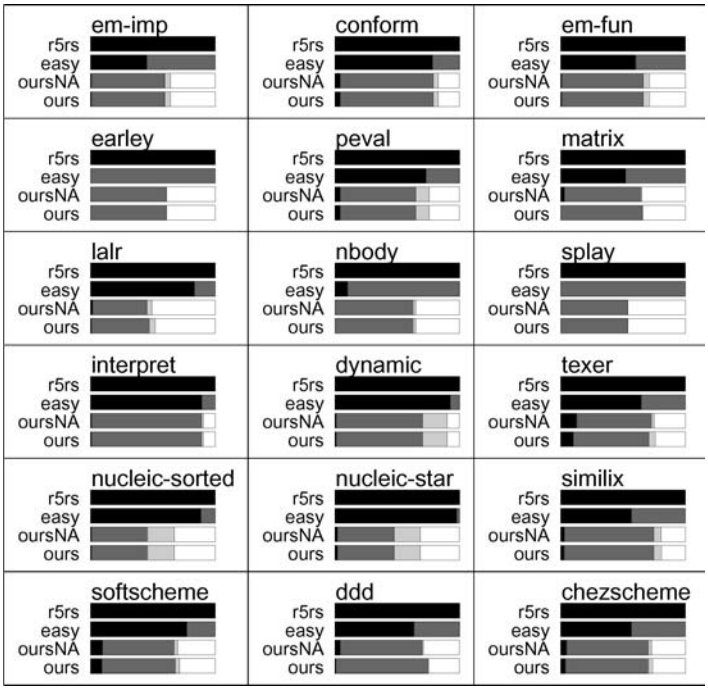

Figure 10. Proportion of bindings classified as ■ *complex*, ■ *lambda*, ■ *simple*, and □ *unreferenced* for the R^{5}RS transformation, the modified R^{5}RS transformation (easy), our transformation without assimilation (oursNA), and our transformation.

Comparing the performance of our algorithm on the two `nucleic` benchmarks (`nucleic-star`, wrapped in a single `letrec*` expression, and `nucleic-sorted`, wrapped in nested `letrec` expressions), we can see that fixing the evaluation order of the formerly top-level definitions has little effect on run-time performance (Figure 2), even though it has a slight effect on the number of introduced assignments and indirect references (Figures 3 and 4) and a more substantial effect on validity checks (Figure 7).

Figure 13 shows that code size is mostly smaller with our transformation, with a few exceptions. Code size and compile time (Figure 14) are affected directly by the transformation and indirectly by optimizations enabled by the transformation. In particular, our transformation opens up more inlining opportunities, and inlining can either reduce or increase code size and may reduce or increase compile time. Not surprisingly, compile time is roughly correlated with code size; when code size goes down, so does compile time, since the burden on the later stages of compilation is smaller.

5. Related work

Much has been written about generating efficient code for restricted recursive binding forms, like our `fix` construct or the `Y` combinator, that bind only `lambda` expressions. Yet, little

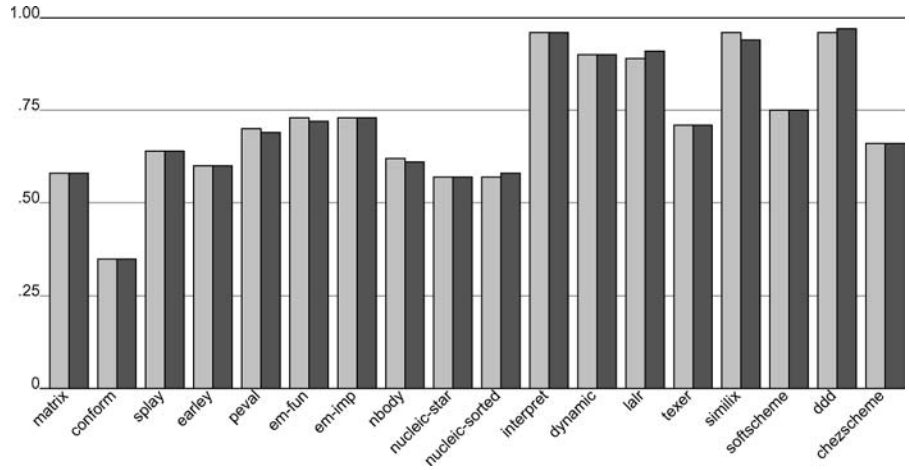

Figure 11. Run times of the code produced by our `letrec` transformation and our `letrec*` transformation, both normalized to the R^{5}RS run times.

	Assignments		References		Validity checks	
	`letrec`	`letrec*`	`letrec`	`letrec*`	`letrec`	`letrec*`
matrix	–	–	–	–	–	–
conform	–	–	–	–	–	–
splay	–	–	–	–	–	–
earley	–	–	–	–	–	–
peval	532	532	532	532	–	–
em-fun	–	–	–	–	–	–
em-imp	–	–	–	–	–	–
nbody	–	–	–	–	–	–
nucleic-star	5	5	7	7	256,307	256,307
nucleic-sorted	2	2	1	1	–	–
interpret	–	–	–	–	–	–
dynamic	1	1	39,929	39,929	–	–
lalr	1	1	297,917	297,917	–	–
texer	–	–	–	–	–	–
similix	–	**1**	–	**95**	–	–
softscheme	73	73	176,932	176,932	152,609	152,609
ddd	8	8	384	384	7	7
chezscheme	178	178	1,132,600	1,132,600	–	–

Figure 12. Numbers of introduced assignments, introduced indirect references, and validity checks executed dynamically by programs compiled using the `letrec` and `letrec*` versions of our transformation. The results are identical in each case, except for Similix (shown in bold).

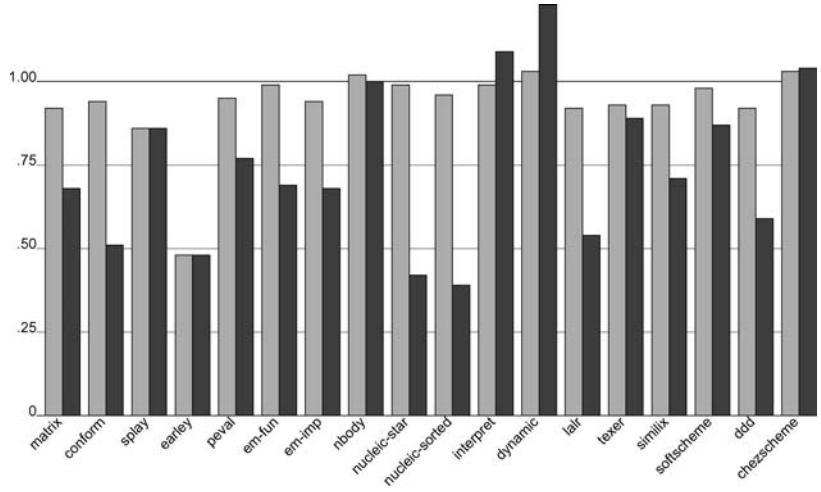

Figure 13. Size of the object code produced by the modified R^{5}RS transformation (easy) and our transformation, both normalized to the R^{5}RS code sizes.

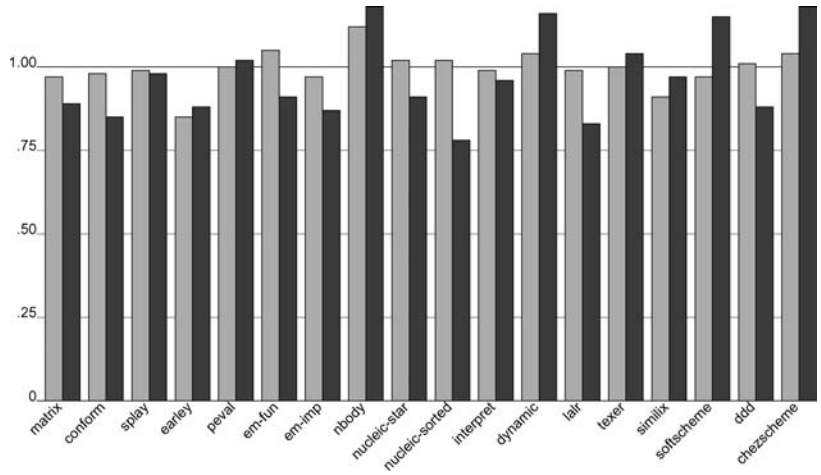

Figure 14. Compile times with the modified R^{5}RS transformation (easy) and our transformation, both normalized to the R^{5}RS compile times.

has been written explaining how to cope with the reality of arbitrary Scheme `letrec` expressions, e.g., by transforming them into one of these restricted forms. Moreover, we could find nothing in the literature describing efficient strategies for detecting violations of the `letrec` restriction.

Steele [13] developed strategies for generating good code for mutually recursive procedures bound by a `labels` form that is essentially our `fix` construct. Because `labels` forms are present in the input language handled by his compiler, he does not describe the translation of general `letrec` expressions into `labels`.

Kranz [8, 9] also describes techniques for generating efficient code for mutually recursive procedures expressed in terms of the `Y` operator. He describes a macro transformation of `letrec` that introduces assignments for any right-hand side that is not a `lambda` expression and uses `Y` to handle those that are `lambda` expressions. This transformation introduces unnecessary assignments for bindings that our algorithm would deem *simple*. His transformation does not attempt to assimilate nested binding constructs. The `Y` operator is a primitive construct recognized by his compiler, much as `fix` is recognized by our compiler.

Rozas [11, 12] shows how to generate good code for mutually recursive procedures expressed in terms of `Y` without recognizing `Y` as a primitive construct, that is, with `Y` itself expressed at the source level. He does not discuss the process of converting `letrec` into this form.

6. Conclusion

We have presented an algorithm for transforming `letrec` expressions into a form that enables the generation of efficient code while preserving the semantics of the `letrec` transformation given in the Revised[5] Report on Scheme [7]. The transformation avoids most of the assignments produced by the revised report transformation by converting many of the `letrec` bindings into simple `let` bindings or into a "pure" form of `letrec`, called `fix`, that binds only unassigned variables to `lambda` expressions. `fix` expressions are the basis for several optimizations, including block allocation and internal wiring of closures. We have shown the algorithm to be effective at reducing the number of introduced assignments and improving run time with little compile-time overhead.

The algorithm also inserts *validity checks* to implement the `letrec` restriction that no reference or assignment to a left-hand-side variable can be evaluated in the process of evaluating the right-hand-side expressions. It inserts few checks in practice and adds practically no overhead to the evaluation of programs that use `letrec`. More importantly, it does not inhibit the optimizations performed by subsequent passes. We are unaware of any other Scheme implementation that performs such checks, but this article shows that validity checks can be introduced without compromising performance even in compilers that are geared toward high-performance applications.

We have also introduced a variant of `letrec`, called `letrec*`, that establishes the values of each variable in sequence from left-to-right. `letrec*` may be implemented with a straightforward modification of the algorithm for implementing `letrec`. We have shown that, in practice, our implementation of `letrec*` is as efficient as `letrec`, even though later passes of our compiler take advantage of the ability to reorder right-hand-side expressions. This is presumably due to the relatively few cases where constraints on the evaluation order remain following our translation of `letrec*`, but in any case, debunks the commonly held notion that fixing the order of evaluation hampers production of efficient code for `letrec`.

While treating `letrec` expressions as `letrec*` clearly violates the Revised[5] Report semantics for `letrec`, we wonder if future versions of the standard should require that internal variable definitions be treated as `letrec*` rather than `letrec`. Left-to-right evaluation order of definitions is often what programmers expect and would make the

semantics of internal variable definitions more consistent with top-level variable definitions. We have shown that there would be no significant performance penalty for this in practice.

We have noticed that some expressions treated as complex involve only data constructors, especially `cons` and `vector`, and `lambda` expressions. Of the cases we have seen, the most common are vectors of procedures used to implement dispatch tables. It may be worth generalizing `fix` to handle data constructors as well as `lambda` expressions. This would be a straightforward addition to later passes of the compiler, which could just as easily wire together groups of pairs, vectors, and procedures as procedures alone, and it would cut down on the number of expressions treated as complex by the `letrec` transformation. While the impact on most programs would be insignificant, programs that rely heavily on dispatch tables and similar structures might benefit substantially.

Appendix

This appendix contains a formal description of the basic letrec transformation and validity check insertion algorithms presented in Sections 2.1 and 2.3. The structure of the formal description mirrors the actual compiler structure, with two passes. The first pass, $\mathcal{C}$, is a source-to-source transformation that inserts validity flags and checks. The second, $\mathcal{T}$, partitions `letrec` bindings and transforms `letrec` into simpler forms.

Assimilation and fixed evaluation order (`letrec*`) can be handled within the structure presented below with straightforward modifications of the two passes. Handling `letrec*` involves adding a `letrec*` case to each pass as described in Section 3. Handling assimilation requires no changes to the first pass and changes only in the partitioning phase of the second pass as described in Section 2.2.

The language of expressions handled by $\mathcal{C}$ consists of a set of standard core forms.

$x \in$ *Vars* ; $e, body \in$ *Input Expressions* ::=

(`quote` *datum*)	*constants*
x	*variable references*
(`set!` x e)	*assignments*
(`begin` $e_1 \ldots e_n$)	*sequencing*
(`if` e_1 e_2 e_3)	*conditionals*
(`lambda` ($x_1 \ldots x_n$) *body*)	*abstractions*
(*pureprim* $e_1 \ldots e_n$)	*pure prim app.*
(e_0 $e_1 \ldots e_n$)	*applications*
(`letrec` (`[`x_1 e_1`]` ... `[`x_n e_n`]`) *body*)	*recursive binding*

In addition to the input expression, $\mathcal{C}$ takes as input the sets of unprotectable and protectable variables and a mapping from `letrec` variables to valid flags. The set of protected variables is implicit. In addition to the output expression, which is in the same language as the input expression, $\mathcal{C}$ returns the set of variables referenced unsafely (while unprotected or protectable) and the set of all variables referenced.

$\mathcal{C}$: *Input Expressions* $\times$ *Unprotectable* $\times$ *Protectable* $\times$ *Flags* $\rightarrow$
Input Expressions $\times$ *Unsafe* $\times$ *Referenced*

$$
\begin{aligned}
u \in \mathit{Unprotectable} &= \mathcal{P}(\mathit{Vars}) \\
p \in \mathit{Protectable} &= \mathcal{P}(\mathit{Vars}) \\
v \in \mathit{Flags} &= \mathcal{P}(\mathit{Vars} \times \mathit{Vars}) \\
d \in \mathit{Unsafe} &= \mathcal{P}(\mathit{Vars}) \\
r \in \mathit{Referenced} &= \mathcal{P}(\mathit{Vars})
\end{aligned}
$$

$\mathcal{C}[\![\,(\texttt{quote}\ \mathit{datum})\,]\!]upv \to \langle [\![\,(\texttt{quote}\ \mathit{datum})\,]\!], \phi, \phi\rangle$

$\mathcal{C}[\![x]\!]upv \to$
  if $x \in (u \cup p)$ then
    if $\langle x, x_f\rangle \in v$ then
      $\langle \mathit{check}[\![x]\!]x_f, \{x\}, \{x, x_f\}\rangle$
    else
      $\langle [\![x]\!], \{x\}, \{x\}\rangle$
  else
    $\langle [\![x]\!], \phi, \{x\}\rangle$

$\mathcal{C}[\![\,(\texttt{set!}\ x\ e)\,]\!]upv \to$
  let $\langle e', d, r\rangle = \mathcal{C}[\![e]\!]upv$ in
    if $x \in (u \cup p) \wedge \langle x, x_f\rangle \in v$ then
      $\langle \mathit{check}[\![\,(\texttt{set!}\ x\ e')\,]\!]x_f, d, (r \cup \{x_f\})\rangle$
    else
      $\langle [\![\,(\texttt{set!}\ x\ e')\,]\!], d, r\rangle$

$\mathcal{C}[\![\,(\texttt{begin}\ e_1 \ldots e_n)\,]\!]upv \to$
  let $\langle e_1', d_1, r_1\rangle = \mathcal{C}[\![e_1]\!]upv$
    $\vdots$
    $\langle e_n', d_n, r_n\rangle = \mathcal{C}[\![e_n]\!]upv$
  in
    $\left\langle [\![\,(\texttt{begin}\ e_1' \ldots e_n')\,]\!], \bigcup_{i=1}^{n} d_i, \bigcup_{i=1}^{n} r_i \right\rangle$

$\mathcal{C}[\![\,(\texttt{if}\ e_1\ e_2\ e_3)\,]\!]upv \to$
  let $\langle e_1', d_1, r_1\rangle = \mathcal{C}[\![e_1]\!]upv$
    $\langle e_2', d_2, r_2\rangle = \mathcal{C}[\![e_2]\!]upv$
    $\langle e_3', d_3, r_3\rangle = \mathcal{C}[\![e_3]\!]upv$
  in
    $\langle [\![\,(\texttt{if}\ e_1'\ e_2'\ e_3')\,]\!], (d_1 \cup d_2 \cup d_3), (r_1 \cup r_2 \cup r_3)\rangle$

$\mathcal{C}[\![\,(\texttt{lambda}\ (x_1 \ldots x_n)\ \mathit{body})\,]\!]upv \to$
  let $\langle \mathit{body}', d, r\rangle = \mathcal{C}[\![\mathit{body}]\!]u\phi v$ in
    $\langle [\![\,(\texttt{lambda}\ (x_1 \ldots x_n)\ \mathit{body}')\,]\!], d, r\rangle$

$\mathcal{C}[\![\,(\mathit{pureprim}\ e_1 \ldots e_n)\,]\!]upv \to$
  let $\langle e_1', d_1, r_1\rangle = \mathcal{C}[\![e_1]\!]upv$
    $\vdots$
    $\langle e_n', d_n, r_n\rangle = \mathcal{C}[\![e_n]\!]upv$
  in

$$\left\langle [\![\texttt{(}\mathit{pureprim}\ \ e_1'\ldots e_n'\texttt{)}]\!], \bigcup_{i=1}^{n} d_i, \bigcup_{i=1}^{n} r_i \right\rangle$$

$\mathcal{C}[\![\texttt{(}e_0\ \ e_1\ldots e_n\texttt{)}]\!]upv \rightarrow$
  let $\langle e_0', d_0, r_0\rangle = \mathcal{C}[\![e_0]\!](u \cup p)\phi v$
    $\langle e_1', d_1, r_1\rangle = \mathcal{C}[\![e_1]\!](u \cup p)\phi v$
    $\vdots$
    $\langle e_n', d_n, r_n\rangle = \mathcal{C}[\![e_n]\!](u \cup p)\phi v$
  in
    $\left\langle [\![\texttt{(}e_0'\ \ e_1'\ldots e_n'\texttt{)}]\!], \bigcup_{i=0}^{n} d_i, \bigcup_{i=0}^{n} r_i \right\rangle$

$\mathcal{C}[\![\texttt{(letrec ((}x_1\ \ e_1\texttt{)}\ldots\texttt{(}x_n\ \ e_n\texttt{)) }\mathit{body}\texttt{)}]\!]upv \rightarrow$
  let $\langle body', d_0, r_0\rangle = \mathcal{C}[\![body]\!]u(p \cup \{x_1, \ldots, x_n\})v$
    $x_f = \text{fresh}(\cdot)$
    $v' = v \cup \{\langle x_1, x_f\rangle, \ldots, \langle x_n, x_f\rangle\}$
  in
    $a \leftarrow \{x_1, \ldots, x_n\}$
    $w \leftarrow d_0 \cap a$
    for $x_i \in w$ do
      $\langle e_i', d_i, r_i\rangle \leftarrow \mathcal{C}[\![e_i]\!](u \cup p)\{x_1, \ldots, x_n\}v'$
      $a \leftarrow a - \{x_i\}$
      $w \leftarrow (w \cup r_i) \cap a$
    end
    for $x_i \in a$ do
      $\langle e_i', d_i, r_i\rangle \leftarrow \mathcal{C}[\![e_i]\!]u(p \cup \{x_1, \ldots, x_n\})v'$
      $a \leftarrow a - \{x_i\}$
    end
    if $x_f \in r$ then
      $\left\langle [\![\begin{array}{l} \texttt{(let ((}x_f\ \texttt{\#f))} \\ \quad \texttt{(letrec ((}x_1\ \ e_1'\texttt{)}\ldots\texttt{(}x_n\ \ e_n'\texttt{)) (set! }x_f\ \texttt{\#t)} \\ \quad\quad body'\texttt{))} \end{array}]\!], \bigcup_{i=0}^{n} d_i, \bigcup_{i=0}^{n} r_i \right\rangle$
    else
      $\left\langle [\![\texttt{(letrec ((}x_1\ \ e_1'\texttt{)}\ldots\texttt{(}x_n\ \ e_n'\texttt{)) }body'\texttt{)}]\!], \bigcup_{i=0}^{n} d_i, \bigcup_{i=0}^{n} r_i \right\rangle$

The *check* function called in the variable reference and assignment cases inserts a conditional expression that signals an error if the value of the flag variable (x_f) is false.

$$check[\![e]\!]x_f \rightarrow [\![\texttt{(begin (if (not }x_f\texttt{) (error "undefined")) }e\texttt{)}]\!]$$

where `(if` e_1 e_2`)` is syntactic sugar for `(if` e_1 e_2 `(void))`. We do not use the more obvious transformation

$$check[\![e]\!]x_f \rightarrow [\![\texttt{(if }x_f\ e\ \texttt{(error "undefined"))}]\!]$$

since this would complicate inlining when e is a variable naming a procedure.

The second pass, $\mathcal{T}$, transforms its input into an output language that is nearly identical to the input language. The only difference is that `letrec` is replaced with `fix`, which binds only `lambda` expressions.

e', $body' \in$ *Output Expressions* ::=

`(quote` *datum*`)`	*constants*
x	*variable references*
`(set!` x e'`)`	*assignments*
`(begin` $e_1' \ldots e_n'$`)`	*sequencing*
`(if` e_1' e_2' e_3'`)`	*conditionals*
λ'	*abstractions*
`(`*pureprim* $e_1' \ldots e_n'$`)`	*pure prim app.*
`(`e_0' $e_1' \ldots e_n'$`)`	*applications*
`(fix (` $[x_1\ \lambda_1']\ \ldots\ [x_n\ \lambda_n']$ `)` $body'$`)`	*recursive binding*

$\lambda' \in$ *Output Lambda Expressions* ::= `(lambda (`$x_1 \ldots x_n$`)` $body'$`)`

The input expression is the only input to $\mathcal{T}$, and the output expression is the only output.

$$\mathcal{T} : \textit{Input Expressions} \rightarrow \textit{Output Expressions}$$

$\mathcal{T}[\![$`(quote` *datum*`)`$]\!] \rightarrow [\![$`(quote` *datum*`)`$]\!]$
$\mathcal{T}[\![x]\!] \rightarrow [\![x]\!]$
$\mathcal{T}[\![$`(set!` x e`)`$]\!] \rightarrow$
  let $e' = \mathcal{T}[\![e]\!]$ in
    $[\![$`(set!` x e'`)`$]\!]$
$\mathcal{T}[\![$`(begin` $e_1 \ldots e_n$`)`$]\!] \rightarrow$
  let $e_1' = \mathcal{T}[\![e_1]\!], \ldots, e_n' = \mathcal{T}[\![e_n]\!]$ in
    $[\![$`(begin` $e_1' \ldots e_n'$`)`$]\!]$
$\mathcal{T}[\![$`(if` e_1 e_2 e_3`)`$]\!] \rightarrow$
  let $e_1' = \mathcal{T}[\![e_1]\!], e_2' = \mathcal{T}[\![e_2]\!], e_3' = \mathcal{T}[\![e_3]\!]$ in
    $[\![$`(if` e_1' e_2' e_3'`)`$]\!]$
$\mathcal{T}[\![$`(lambda (`$x_1 \ldots x_n$`)` $body$`)`$]\!] \rightarrow$
  let $body' = \mathcal{T}[\![body]\!]$ in
    $[\![$`(lambda (`$x_1 \ldots x_n$`)` $body'$`)`$]\!]$
$\mathcal{T}[\![$`(`*pureprim* $e_1 \ldots e_n$`)`$]\!] \rightarrow$
  let $e_1' = \mathcal{T}[\![e_1]\!], \ldots, e_n' = \mathcal{T}[\![e_n]\!]$ in
    $[\![$`(`*pureprim* $e_1' \ldots e_n'$`)`$]\!]$
$\mathcal{T}[\![$`(`e_0 $e_1 \ldots e_n$`)`$]\!] \rightarrow$
  let $e_0' = \mathcal{T}[\![e_0]\!], e_1' = \mathcal{T}[\![e_1]\!], \ldots, e_n' = \mathcal{T}[\![e_n]\!]$ in
    $[\![$`(`e_0' $e_1' \ldots e_n'$`)`$]\!]$
$\mathcal{T}[\![$`(letrec ((`x_1 e_1`) ... (`x_n e_n`)) (`$body$`)`$]\!] \rightarrow$
  partition $\langle x_1, \mathcal{T}[\![e_1]\!]\rangle \ \ldots \ \langle x_n, \mathcal{T}[\![e_n]\!]\rangle$ into

$\langle x_u, e_u{}'\rangle \ldots$ if *unreferenced?*(x_u)
$\langle x_s, e_s{}'\rangle \ldots$ if *unassigned?*(x_s) and *simple?*$(e_s', \{x_1, \ldots, x_n\})$
$\langle x_l, e_l{}'\rangle \ldots$ if *unassigned?*(x_l) and *lambda?*(e_l')
$\langle x_c, e_c{}'\rangle \ldots$ otherwise

in

let $body' = \mathcal{T}[\![\texttt{body}]\!], \langle x_t, \ldots\rangle = \langle \text{fresh}(x_c), \ldots\rangle$ in

$[\![$ `(let ((`x_s $e_s{}'$`) ... (`x_c `(void))...)`
`(fix ((`x_l $e_l{}'$`)...)`
`(begin`
$e_u{}'\ldots$
`(let ((`x_t $e_c{}'$`)...) (set!` x_c x_t`)...)`
$body'$`)))` $]\!]$

Partitioning involves a set of tests on the left-hand-side variables and right-hand-side expressions. The *unreferenced?* and *unassigned?* checks inspect flags incorporated into the representation of variables by an earlier pass. The *lambda?* check is trivial, and *simple?* is given below.

$$\textit{simple?} : \textit{Output Expressions} \times \textit{LHS Vars} \to \textit{boolean}$$

$$v \in \textit{LHS Vars} = \mathcal{P}(\textit{Vars})$$

simple?$([\![$ `(quote` *datum*`)` $]\!], v) \to$ true
simple?$([\![x]\!], v) \to$ if $x \in v$ then false else true
simple?$([\![$ `(set!` x e'`)` $]\!], v) \to$ false
simple?$([\![$ `(begin` $e_1{}' \ldots e_n{}'$`)` $]\!], v) \to \bigwedge_{i=1}^{n}$*simple?*$([\![e_i{}']\!], v)$
simple?$([\![$ `(if` $e_1{}'$ $e_2{}'$ $e_3{}'$`)` $]\!], v) \to$
simple?$([\![e_1{}']\!], v) \wedge$ *simple?*$([\![e_2{}']\!], v) \wedge$ *simple?*$([\![e_3{}']\!], v)$
simple?$([\![$ `(lambda (`$x_1 \ldots x_n$`)` $body'$`)` $]\!], v) \to$ false
simple?$([\![$ `(pureprim` $e_1{}' \ldots e_n{}'$`)` $]\!], v) \to \bigwedge_{i=1}^{n}$*simple?*$([\![e_i{}']\!], v)$
simple?$([\![$ `(`$e_0{}'$ $e_1{}' \ldots e_n{}'$`)` $]\!], v) \to$ false
simple?$([\![$ `(fix ((`x_1 $e_1{}'$`) ... (`x_n $e_n{}'$`)) ` $body'$`)` $]\!], v) \to$ false

Acknowledgments

Insightful comments by Julia Lawall and the anonymous reviewers led to many improvements in the presentation. Dipanwita Sarkar is supported by a gift from Microsoft Corporation.

References

1. Bondorf, A. *Similix Manual, System Version 5.0*. DIKU, University of Copenhagen, Denmark, 1993.
2. Bose, B. DDD—A transformation system for digital design derivation. Technical Report 331, Computer Science Dept., Indiana Univ., Bloomington, Ind., May 1991.

3. Burger, R.G. The Scheme machine. Technical Report 413, Computer Science Dept., Indiana Univ., Bloomington, Ind., August 1994.
4. Dybvig, R. K., Hieb, R., and Bruggeman, C. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, **5**(4) (1993) 295–326.
5. Feeley, M., Turcotte, M., and Lapalme, G. Using Multilisp for solving constraint satisfaction problems: An application to nucleic acid 3D structure determination. *Lisp and Symbolic Computation*, **7** (1994) 231–247.
6. Gabriel, R.P. *Performance and Evaluation of LISP Systems*. MIT Press Series in Computer Systems. MIT Press, Cambridge, Massachusetts, 1985.
7. Kelsey, R., Clinger, W., and Rees, J. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, **11**(1) (1998) 7–105.
8. Kranz, D.A., Orbit, An optimizing compiler for scheme. PhD thesis, Yale University, May 1988.
9. Kranz, D.A., Kelsey, R., Rees, J.A., Hudak, P., Philbin,J., and Adams, N.I. Orbit: an optimizing compiler for Scheme. *SIGPLAN Notices, ACM Symposium on Compiler Construction*, **21**(7) (1986) 219–233.
10. Rees, J.A., Adams, N.I., and Meehan, J.R. *The T Manual*. Yale University, New Haven, Connecticut, USA, 1984. Fourth edition.
11. Rozas, G.J. Liar, an Algol-like compiler for Scheme, S.B. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1984.
12. Rozas, G.J. Taming the Y operator. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, San Francisco, USA, June 1992, pp. 226–234.
13. Guy L. Steele Jr. Rabbit: A compiler for Scheme. MIT AI Memo 474, Massachusetts Institute of Technology, Cambridge, Mass., May 1978.
14. Waddell, O. and Dybvig, R.K. Extending the scope of syntactic abstraction. In *Conference Record of the Twenty Sixth Annual ACM Symposium on Principles of Programming Languages*, January 1999, pp. 203–213.
15. Waddell, O., Sarkar, D., and Dybvig, R.K. Robust and effective transformation of letrec. In *Scheme 2002: Proceedings of the Third Workshop on Scheme and Functional Programming*, October 2002.
16. Wright, A.K. and Cartwright, R. A practical soft type system for scheme. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, 1994, pp. 250–262.