Fixing Letrec (reloaded)

Abdulaziz Ghuloum
Indiana University
aghuloum@cs.indiana.edu

R. Kent Dybvig
Indiana University
dyb@cs.indiana.edu

Abstract

The Revised[6] Report on Scheme introduces three fundamental changes involving Scheme's recursive variable binding constructs. First, it standardizes the sequential recursive binding construct, `letrec*`, which evaluates its initialization expressions in a strict left-to-right order. Second, it specifies that internal and library definitions have `letrec*` semantics. Third, it prohibits programs from invoking the continuation of a `letrec` or `letrec*` init expression more than once. The first two changes increase the incentive for handling `letrec*` efficiently, while the third change gives the compiler more options for transforming `letrec` and `letrec*` expressions.

This paper extends an earlier effort of Waddell, Sarkar, and Dybvig to handle the Revised[5] Report `letrec` and the (then nonstandard) `letrec*` efficiently. It presents more aggressive transformations for `letrec` and `letrec*` that take advantage of the new prohibition on invoking the continuations of initialization expressions multiple times. The implementation employs Tarjan's algorithm for finding strongly connected components in a graph that encodes the dependencies among the bindings.

Keywords Scheme, recursive binding construct, internal definitions, mutual recursion, mutual definition, continuations, optimization, letrec

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Workshop on Scheme and Functional Programming '09 August 22, 2009, Cambridge, Massachusetts

1. Introduction

Scheme's `letrec` form, used to create recursive bindings, is easily transformed into a standard combination of `let` and `set!` expressions [4]. Unfortunately, the standard transformation introduces unnecessary assignments that may inhibit subsequent optimization of the form. An alternative transformation is described by Waddell, Sarkar, and Dybvig [9]. This transformation often succeeds in avoiding unnecessary assignments while maintaining the Revised[5] Report semantics for `letrec`. Waddell, et al., also define a variant of `letrec`, called `letrec*` by analogy to `let*`, that evaluates its initialization expressions from left to right, and they present a similar optimizing transformation for `letrec*`.

The Revised[6] Report on Scheme [5] changes the status quo by including `letrec*` as well as `letrec` in the language and, more importantly, by changing the semantics of internal defines so that `define`-bound variables behave as if bound by `letrec*` rather than `letrec`. An immediate consequence of this changes is that R[6]RS programs (on average) will contain more variables bound according to the `letrec*` semantics. Thus, optimizing `letrec*` has become even more important. Over time, we also expect programmers to take advantage of this change by using the values of earlier bindings for the purpose of initializing subsequent ones. Unfortunately, this will result in more of the so-called *complex* bindings for which the Waddell, et al., transformation, like the naive transformation, often produces unnecessary assignments.

The Revised[6] Report makes one other semantic change in its recursive binding constructs, which is to prohibit programs from invoking the continuation of a `letrec` or `letrec*` init expression more than once. This gives the implementation more flexibility to avoid producing assignments whose absence could otherwise

have been detected by invoking the continuation of an initialization expression multiple times.

This paper presents a new transformation algorithm that often produces fewer assignments than the Waddell, et al., transformation, especially for `letrec*`. At worst, it produces the same number of assignments as the Waddell, et al., transformation. It accomplishes this by an aggressive partitioning of bindings into minimal mutually dependent groups, aided by Tarjan's algorithm [6] for finding strongly connected components, and by taking advantage of the new prohibition against invoking the continuations of initialization expressions multiple times.

The rest of this paper is organized as follows. Section 2 gives an overview of the syntax and semantics of Scheme's `letrec` and `letrec*` forms along with the run-time restrictions imposed by the standard. Section 3 gives the naive but straightforward implementation of these forms based on source-level macro transformation. Section 4 summarizes the *fixing letrec* algorithm of Waddell, Sarkar, and Dybvig [9]. Section 5 motivates our work by showing simple examples where the original algorithm yields undesirable results. Section 6 describes the constraints that our transformation must follow, the analysis it requires, and how it encodes the information it needs in a graph form. Section 7 shows how the algorithm uses the strongly connected-components to perform the actual transformation. Finally, Section 8 provides some concluding remarks.

2. Semantics of `letrec` and `letrec*`

The syntax of the `letrec` and `letrec*` forms are identical, except for the opening keyword:

`(letrec ([`*var init*`] ...) `*body* `...)`

`(letrec* ([`*var init*`] ...) `*body* `...)`

The lexical scoping rules for `letrec` and `letrec*` are the same: all of *var*s are visible in all *init* expressions as well as in the *body* definitions and expressions. For both forms, evaluation proceeds as follows:

1. All *var*s are bound to fresh locations.
2. The *init* expressions are evaluated and the value of each *init* is assigned to the corresponding *var*.
3. The *body* is evaluated and the values of the last expression are returned.

The two forms differ in the specifics of item 2. For `letrec`, all *init* expressions are evaluated first, in some unspecified order, before all bindings are initialized to the computed values. For `letrec*`, the bindings are initialized sequentially: each *init* is evaluated, and the corresponding *var* is set before the next binding is initialized.

2.1 Restrictions

Although all *var*s are visible in all *init* expressions, the actual evaluation of the *init* expressions must obey additional restrictions. For `letrec`, each *init* expression must evaluate without referencing or assigning the value of any of the *var*s. During the evaluation of `letrec*` *init* expressions, however, references and assignments to previously initialized bindings (e.g., ones that appear earlier in the list of bindings) are permitted. The R^{6}RS requires that implementations *must* detect and report violations of this restriction [5], though some implementations currently ignore this requirement.

The R^{6}RS also prohibits programs from invoking the continuation (returning from the evaluation) of an *init* expression. The report specifies that implementation *should* detect and report such violations. Thus, portable programs cannot depend on the implementation's behavior when the program violates this prohibition.

Implementations can enforce both of these restrictions via a source code transformation that inserts additional checks.

A transformation that guards against the first restriction with minimal overhead is described by Waddell, et al. [9]. The transformation works as a pre-pass to the *fixing letrec* algorithm. It first inserts validity checks amounting to binding one *initialized?* variable per `letrec` expression, or one per `letrec*` binding, and inserting validity checks anywhere a possible violation of the restriction might occur. Because these checks are separate from the actual `letrec` or `letrec*` bindings, the transformation does not inhibit optimizations involving the bindings. Because it works independently, as a prepass, of the transformation of `letrec` and `letrec*` into more primitive forms, it is applicable regardless of the transformation used, whether it be the naive transformation, the one described by Waddell, et al., or the more sophisticated transformation described in this paper.

To enforce the second restriction, an implementation can transform an unchecked `letrec` or `letrec*` ex-

pression into a checked one via a simple transformation:

```
(define-syntax checked-letrec
  (syntax-rules ()
    [(_ ([var init] ...) b b* ...)
     (letrec ([var (once init)] ...)
        b b* ...)]))
```

where `once` is a primitive keyword that behaves according to the following definition:

```
(define-syntax once
  (syntax-rules ()
    [(_ expr)
     (let ([returned? #f])
       (let ([v expr])
         (when returned?
           (assertion-violation ---))
         (set! returned? #t)
         v))]))
```

The `once` wrapper should be omitted at least for *init* expressions that cannot possibly invoke their continuations multiple times, including constants, `lambda` expressions, variable references, and applications of most primitives to simple values. An implementation might also attempt to omit the `once` wrapper for other expressions it can prove do not invoke their continuations multiple times.

The remainder of this paper assumes that all uses of `letrec` and `letrec*` are either correct (i.e, do not violate the two restrictions of the report stated above) or that the required checks have already been inserted by a prior transformation.

3. Naive `letrec` / `letrec*` transformation

The task of the *fixing letrec* pass of the compiler is to rewrite the general Scheme `letrec` and `letrec*` forms into simpler forms that subsequent passes of the compiler can easily handle.

The simplest (and most naive) transformation for `letrec*` can be achieved at the source level via a simple macro:

```
(define-syntax letrec*
  (syntax-rules ()
    [(_ ([var init] ...) b b* ...)
     (let ([var #f] ...)
       (set! var init) ...
       (let () b b* ...))]))
```

It is easy to see how the output of this transformation performs the required `letrec*` evaluation semantics. (It enforces none of the restrictions, but we are assuming these are enforced by some earlier transformation.) The same transformation can be used for `letrec` expressions, but it unnecessarily forces the initialization expressions to be evaluated from left to right, preventing the compiler from subsequently choosing a different order that might result in more efficient code. The following definition of `letrec` avoids specifying a particular evaluation order:

```
(define-syntax letrec
  (lambda (stx)
    (syntax-case stx ()
      [(_ ([var init] ...) b b* ...)
       (with-syntax ([(tmp ...)
                      (generate-temporaries
                        #'(var ...))])
         #'(let ([var #f] ...)
             (let ([tmp init] ...)
               (set! var tmp) ...
               (values))
             (let () b b* ...)))]))))
```

These naive transformations have two unfortunate consequences. First, the compiler cannot always "undo" the transformation since the resulting code overspecifies the intended behavior. That is, the compiler cannot tell whether the resulting code is intended to behave according to the `letrec` / `letrec*` semantics, or whether it really means binding some variables to a constant (`#f`) followed by assigning these variables to some other values. Second, some optimizing compilers of Scheme cannot handle assigned variables as efficiently as unassigned ones. In addition to inhibiting inlining, copy-propagation, constant folding, direct jumps to local procedures, and other optimizations, assigned variables often end up being *boxed* in heap-allocated locations [1], thus introduce additional runtime overhead for heap overflow checks when the bindings are introduced, additional memory traffic when the bindings are used, and possibly additional garbage collection overhead.

4. Waddell's `letrec` transformation

The Waddell, et al., algorithm for *fixing letrec* works by transforming a `letrec` expression into a set of `let` and `fix` binding forms. The `fix` binding form is a

restricted form of `letrec` in which all bound *var*s are unassigned and all *init*s are `lambda` expressions. The transformation partitions the `letrec` bindings into four sets:

1. `[xu eu] ...` *unreferenced*
2. `[xs es] ...` *simple*
3. `[xl el] ...` *lambda*
4. `[xc ec] ...` *complex*

Unreferenced bindings are those evaluated for effect only: their computed values are never used in the program. Bindings that are unreferenced in the program are eliminated as are assignments to these unreferenced bindings. The *simple* bindings are those satisfying the following criteria: (1) the *var* is unassigned, (2) the *init* is not a `lambda` expression and does not contain a reference to any *var* bound in the same `letrec`, and (3) evaluating its *init* expression cannot capture and invoke its continuation more than once. (In R^{5}RS Scheme, an initialization expression is permitted to invoke its continuation more than once, but expressions that do are not considered *simple*.) The *lambda* bindings set contains the bindings where the *var* is unassigned and the *init* is a `lambda` expression. The *lambda* bindings are bound using `fix` in the output of the transformation. All other bindings are considered *complex*.

After partitioning the bindings, a `letrec` expression is transformed to:

```
(let ([xs es] ...       ; simple bindings
      [xc #f] ...)      ; complex bindings
  (fix ([xl el] ...)    ; lambda bindings
    eu ...              ; unreferenced
    (let ([xt ec] ...) ; complex values
      (set! xc xt) ...)
    body))
```

with the inner `let` expression omitted if no bindings are *complex*.

For `letrec*`, the bindings are similarly partitioned into *unreferenced*, *simple*, *lambda*, and *complex* bindings following the same criteria used for `letrec` expressions. The only difference in the transformation is that the *unreferenced init* expressions and the assignments to *complex* bindings must be interleaved in order to preserve the left-to-right evaluation order required for `letrec*`. Assuming no unreferenced bindings, a `letrec*` expression is transformed to:

```
(let ([xs es] ...       ; simple bindings
      [xc #f] ...)      ; complex bindings
  (fix ([xl el] ...)    ; lambda bindings
    (set! xc ec) ...    ; complex inits
    body))
```

Because some expressions considered *simple* for purposes of the `letrec` transformation might raise exceptions or perform side effects, the rules for simple expressions given above are too liberal to preserve the strict left-to-right evaluation order constraint for `letrec*` initialization expression. Thus, the Waddell, et al., transformation treats as *complex* any otherwise *simple* binding whose right-hand side is not effect free" [9]. For example, the expression `(car x)` is not considered *simple* for purposes of the `letrec*` transformation, since the `car` procedure may raise an exception if `x` is not a pair. In general, an expression cannot be considered effect free if it modifies state, exits from the program, raises an exception, or might not terminate. This has the unfortunate consequence of making many bindings *complex*, leading to all the drawbacks of assigned variables mentioned earlier. To be effective, *fixing letrec* should have a better story for handling such *init* expressions than to treat them as *complex*.

5. Why does it matter?

Consider the following simple program fragment:

```
(let ()
  (define q 8)
  (define f (lambda (x) (+ x q)))
  (define r (f q))
  (define s (+ r (f 2)))
  (define g (lambda () (+ r s)))
  (define t (g))
  t)
```

which should be understood in terms of a straightforward transliteration into `letrec*`, with the bindings appearing in the same order.

According to the Waddell, et al., partitioning algorithm, the binding for `q` is considered *simple*, the bindings for `f` and `g` are *lambda*, and the bindings for `r`, `s`, and `t` are *complex*. After *fixing letrec*, the code is transformed to:

```
(let ([q 8])
  (let ([r #f] [s #f] [t #f])
    (fix ([f (lambda (x) (+ x q))]
```

```
      [g (lambda () (+ r s))])
  (set! r (f q))
  (set! s (+ r (f 2)))
  (set! t (g))
  t)))
```

Because the variables `q`, `f`, and `g` are unassigned, they are straightforward targets for inlining, copy propagation, and other optimizations performed by a source optimizer such as the one described by Waddell and Dybvig [7, 8]. The assigned variables `r`, `s`, and `t` are not, so the resulting code after optimization might look like the following less-than-ideal code:

```
(let ([t #f] [s #f] [r #f])
  (set! r 16)
  (set! s (+ 10 r))
  (set! t (+ r s))
  t)
```

Compare this with the following program, which at the source level appears less efficient as it contains several more `lambda` expressions and procedure calls:

```
(let ()
  (define q (lambda () 8))
  (define f (lambda (x) (+ x (q))))
  (define r (lambda () (f (q))))
  (define s (lambda () (+ (r) (f 2))))
  (define g (lambda () (+ (r) (s))))
  (define t (lambda () (g)))
  (t))
```

It is discomforting and counterintuitive that the Waddell and Dybvig source optimizer boils this down to just the constant "42" while the simpler program shown at the beginning of this section does not, all because the `letrec*` transformation produces unnecessary assignments.

The goal of an efficient `letrec` and `letrec*` transformation is to reduce the number of emitted variable assignments (`set!`s) by turning as many of the bindings into simple `let` or `fix` bindings as possible. The original *fixing letrec* algorithm fails to do so, in essence, because it assumes that all of the bindings are possibly mutually dependent and thus must be grouped together in the output.

The new algorithm removes this restriction by grouping a set of bindings together only if they are mutually dependent. It can thus handle each nonrecursive complex binding in a group by itself without introducing an assignment. For example, it transforms the program given above into the following assignment-free program:

```
(let ([q 8])
  (fix ([f (lambda (x) (+ x q))])
    (let ([r (f q)])
      (let ([s (+ r (f 2))])
        (fix ([g (lambda () (+ r s))])
          (let ([t (g)])
            t))))))
```

6. Constraints, analysis, and encoding

Ideally, the transformation would place each `letrec` or `letrec*` binding in its own `let` or `fix` expression, each nested inside the previous ones, and so avoid all assignments. It cannot do so, however, due to semantic constraints that must be obeyed in order to handle the full generality of `letrec` and `letrec*`.

6.1 Constraints due to lexical scope

The lexical scope rule for `let` bindings requires that the right-hand-side expressions cannot reference any of the left-hand-side variables or any variables not bound in an outer scope. So, if `x` and `y` are two `letrec` bindings, we cannot place `x` in an outer `let` to `y` if `x`'s *init* expression refers to `y`.

The lexical scope rule for `fix` bindings allows for mutual recursion among `lambda` expressions: the variables at the left-hand-side can appear at the right-hand-side `lambda` expressions as well as in the `fix` body. Consider the following example:

```
(let ()
  (define f (lambda () (even? 5)))
  (define even?
    (lambda (x)
      (or (zero? x) (odd? (- x 1)))))
  (define odd?
    (lambda (x) (not (even? x))))
  (define t (f))
  t)
```

Because `even?` and `odd?` are mutually recursive, they must be bound by the same `fix`. The procedure `f` references `even?`, but neither `even?` nor `odd?` references `f`, so `f` is placed in an inner `fix`. (Since it is nonrecursive, it could be bound by `let` instead, but we bind it using `fix` to facilitate a later, independent transformation that combines nested `fix` expressions to simplify the block

allocation and wiring together of procedures.) The variable binding `t` is placed inside the binding for `f` since it references `f`, and none of the other bindings reference it. The final product of the transformation shows the effect of scoping constraints on the output of the transformation.

```
(fix ([even? (lambda (x) --- odd? ---)]
      [odd? (lambda (x) --- even? ---)])
  (fix ([f (lambda () (even? 5))])
    (let ([t (f)])
      t)))
```

The transformation cannot transform all `letrec` and `letrec*` forms into arbitrarily nested `let` and `fix` bindings and thus avoid ever producing an assignment. For example, the program

```
(letrec ([x (list (lambda () x))]) x)
```

necessarily requires the introduction of an assignment to establish the cyclic relationship between the procedure and the pair. It is thus transformed into the following equivalent program

```
(let ([x #f])
  (set! x (list (lambda () x)))
  x)
```

Similarly, an assignment may be introduced for more than one recursively defined nonprocedure binding. In the following example, `x` and `y` are mutually recursive nonprocedure bindings, both requiring an assignment. The binding for `f` refers to both `x` and `y` but it can appear in an inner binding because neither `x` nor `y` refers to it.

```
(let ()
  (define x (list (lambda () y)))
  (define f (lambda () (cons x y)))
  (define y (list (lambda () x)))
  (define t (f))
  t)
=>
(let ([x #f] [y #f])
  (fix ([f (lambda () (cons x y))])
    (set! x (list (lambda () y)))
    (set! y (list (lambda () x)))
    (let ([t (f)])
      t)))
```

If the initialization expressions for two complex bindings are not mutually recursive, however, the assignments can be avoided. For example, assuming `x`, `f`, and `g` are defined outside of the following expression and both `y` and `z` are referenced within `body`:

```
(let ()
  (define y (f x))
  (define z (g x))
  body)
```

is transformed into the following.

```
(let ([y (f x)])
  (let ([z (g x)])
    body))
```

This transformation would also be valid for the equivalent `letrec` expression, with the R^{6}RS semantics. It would not be valid for internal defines following the R^{5}RS `letrec` semantics, however, since a continuation catpured by `f` might be invoked multiple times, each causing a new location for `z` to be created, possibly holding a different value each time.

6.2 Constraints due to evaluation order

For `letrec*` (but not for `letrec`), an additional constraint on the transformation is forced by the requirement that the *init* expressions be evaluated sequentially from left to right. Evaluating an *init* expression may, however, cause side effects. The `letrec*` transformation therefore *must* preserve the order of observable side effects in the *init* expressions as they appear in the input program.

Whether an *init* expression causes an observable side effect is undecidable in general, so the analysis to determine whether it does so must be conservative. Our current implementation considers an *init* expression to have a side effect if it contains a procedure call or a `set!` expression occurring outside of a `lambda` expression. This gives the implementation the freedom to reorder the simple bindings comprising of constants, variable references, primitive references, `lambda` expressions, and the combination of simple expressions to form certain primitive calls, `if`, `begin`, `let`, `letrec`, and `letrec*` expressions.

6.3 Viewing constraints as a dependency graph

The first part of our algorithm works by encoding each `letrec` and `letrec*` instance as a directed graph G

in which the nodes are the set of vars and the arcs represent the dependencies between bindings.

The dependencies are derived from the lexical scope and (for `letrec*`) evaluation order constraints and control the overall structure of the result of the transformation. For example, the lexical scope rule dictates that at the output of the transformation, an $init$ in an outer `let` or `fix` binding must not reference a variable placed in an inner binding. Or, stated differently, if x_i appears free in some $init_j$ expression, x_i has to be bound before (or at the same time) $init_j$ is evaluated but not later. Thus, the binding for x_j is said to depend on the binding for x_i as in the following definition:

Dependency graph for `letrec`:
Given a set of bindings $\{\langle x_i, init_i\rangle *\}$, the dependency graph is $G = \langle V, E\rangle$ where
$V = \{x_i *\}$ and
$E = \{(x_j, x_i) : x_i \in FV(init_j)\}$.

Constraints due to the specified evaluation order for `letrec*` are encoded in a similar fashion. If two $init$ expressions might perform observable side effects, their order must be preserved in the output of the transformation. Thus, the edges of the dependency graph for `letrec*` must contain an arc $x_j \rightarrow x_i$ if $init_j$ must be evaluated after $init_i$.

Dependency graph for `letrec*`:
Augments the edges of the corresponding `letrec` graph with the set
$\{(x_j, x_i) : init_j$ and $init_i$ are *complex* and $j > i\}$.

Dependencies are transitive, e.g., if $(x_i, x_j) \in E$ and $(x_j, x_k) \in E$, then $(x_i, x_k) \in E$ and thus need not be encoded explicitly. In fact, our implementation adds at most $N - 1$ order-of-evaluation edges for a `letrec*` containing N bindings and not the $O(N^2)$ edges required if all pairs of dependent bindings are connected.

The dependency graphs for the programs presented so far are shown in figure 1. It is instructive to compare each dependency graph with the corresponding code after transformation.

6.4 Strongly connected components

Once the dependencies among bindings are determined, the resulting graph is partitioned into strongly connected components (SCCs) using Tarjan's algorithm [6]. An SCC in G is the largest subgraph of G in which every node is reachable when starting from every other node in the SCC.

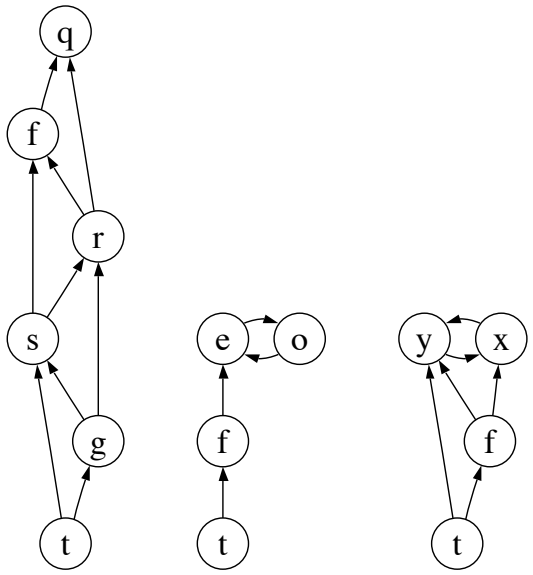

Figure 1. The dependency graphs for the first three source programs shown in Section 6. The left-most graph is for the program at the beginning of Section 5. The two shorter graphs are for 6.1 and 6.2. The edges in these graphs record the constraints due to lexical scope and evaluation order.

The Tarjan algorithm works by visiting all reachable nodes starting from some node x_s, in a depth-first-order traversal, ranking each node with its depth of traversal, and combining cycles as they are encountered.

In its simplest case, an SCC contains just a single node (which may or may not be pointing back to itself). If an SCC contains more than one node, these nodes are mutually dependent, either because they are mutually recursive or because one node references another that must follow it in the left-to-right evaluation order of `letrec*`.

The arcs of G not only determine the set of SCCs but also defines a partial order relation on the SCCs. Our implementation of Tarjan's algorithm returns an ordered list of SCCs such that SCC_i does not depend upon SCC_j if $i < j$, but SCC_j might depend on SCC_i.

7. Transformation based on SCCs

The `letrec` and `letrec*` transformation partitions the set of bindings into strongly connected components.

We only need to consider how to generate code for a single SCC. This is because the list of SCCs obtained from the Tarjan algorithm is ordered according to the required dependencies, so the bindings for each SCC need merely be nested within the bindings for previous SCCs.

Each SCC is handled in a manner similar to the way the Waddell, et al., transformation handles the entire set of `letrec` and `letrec*` bindings, except that we treat specially the case where an SCC contains only one binding.

Single bindings: Code generation for an SCC containing a single binding $\langle var, init\rangle$ is handled according to one of the following cases:

– If $init$ is a `lambda`, and var is unassigned:
`(fix ([`var $init$`]) `$rest$`)`

– If $var \notin FV(init)$:
`(let ([`var $init$`]) `$rest$`)`

– Otherwise, we resort to assignment:
`(let ([`var `#f])`
`  (set!` var $init$`)`
$rest$`)`

The $rest$ code in the transformation denotes the code for subsequent SCCs and the transformed $body$ expression.

Multiple bindings Code generation for an SCC containing multiple bindings is done by partitioning the bindings into two parts:

1. $\langle var_\lambda, init_\lambda\rangle$ if $init$ is a `lambda` expression and var is unassigned.
2. $\langle var_c, init_c\rangle$ otherwise.

Tarjan's algorithm does not guarantee an order of returned elements for each SCC, so, for `letrec*`, the complex bindings need to be sorted according to their occurrence in the original `letrec*` form. The two partitions are used to produce the following code:

```
(let ([var_c #f] ...)
  (fix ([var_λ init_λ] ...)
    (set! var_c init_c) ...
    rest))
```

The graphs shown in Figure 1 tell the whole story. The first graph shows a long chain of dependencies, but all SCCs are of size 1. This is why the transformed code (shown at the end of Section 5) has a deeply nested `let` and `fix` forms, each binding a single variable. The second graph (the `even?/odd?` example) shows two mutually recursive bindings in one SCC (producing a `fix` binding in the output) and two singleton SCCs (producing two `let` bindings). The last graph also shows an SCC with two bindings, but this time, `x` and `y` cannot be bound with `fix`, and thus an assignment is needed.

8. Conclusions

The `letrec` and `letrec*` transformation algorithm described in this paper improves on the handling of `letrec` and `letrec*` by Waddell, et al. [9], by producing fewer assignments, thus reducing direct overhead from assignments as well as the indirect overhead from the inhibition of certain optimizations. Each binding that would be considered *simple* by the Waddell, et al., transformation ends up in its own SCC. Since simple bindings cannot reference any of the left-hand-side variables, they are handled by the second single-binding case above, i.e., they are bound by `let` expressions. Similarly, all *lambda* bindings end up bound by `fix` expressions. The difference between the Waddell, et al., transformation and ours is in the treatment of bindings the former considers *complex*. While the Waddell, et al., transformation always introduces assignments for *complex* bindings, our transformation avoids assignments for those that are nonrecursive and end up in their own SCC, which appears in our initial experiments to be by far the most common case. For `letrec*`, some apparently *simple* bindings must be considered complex, such as primitive calls that might raise exceptions, so this improvement is particularly important for R^{6}RS, which employs `letrec*` semantics for internal definitions.

We have implemented the original and new algorithms and wired them into the Ikarus [3] and Chez Scheme [2] compilers, with a parameter to select which algorithm to run. Using both systems, we compared the effectiveness of the two algorithms in eliminating complex bindings while bootstrapping the compiler and run-time libraries, which for both systems involves thousands of `letrec` or `letrec*` bindings. 10% of Ikarus's bindings and 7.2% of Chez Scheme's bindings are considered complex by the original algorithm versus only .6% and .1% for the new. Thus, in both systems, the complex bindings are a substantial fraction of

all `letrec`/`letrec*` bindings using the original algorithm, but an insignificant fraction using the new.

Although the bootstrapping times for Ikarus improve by just under 11% when switching from the original to new algorithms, the bootstrapping times for Chez Scheme do not improve significantly. This is likely due, in part, to the fact that most of the code that affects Chez Scheme's compile-time was written before `letrec*` was introduced and internal definitions were changed to use `letrec*` semantics. Furthermore, most of the code was developed with an even less effective algorithm for handling `letrec` than the Waddell, et al., algorithm, and this has caused the developers to shy away from potentially complex bindings in time-critical portions of the system.

Based partly on our experience, we believe that programmers will take advantage of the `letrec*` semantics for internal definitions, especially with the knowledge that `letrec*` can be implemented effectively, and that the improvement afforded by the new transformation will become increasingly more valuable over time. It will be interesting to test this hypothesis once a large corpus of programs written specifically for R^{6}RS becomes available.

References

[1] R. Kent Dybvig. *Three Implementation Models for Scheme*. PhD thesis, University of North Carolina, Chapel Hill, April 1987.

[2] R. Kent Dybvig. *Chez Scheme Version 7 User's Guide*. Cadence Research Systems, 2005.

[3] Abdulaziz Ghuloum. Ikarus Scheme User's Guide, 2009.

[4] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[5] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten (editors). Revised^6 report on the algorithmic language Scheme. 2007.

[6] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[7] Oscar Waddell. *Extending the Scope of Syntactic Abstraction*. PhD thesis, Indiana University Computer Science Department, August 1999.

[8] Oscar Waddell and R. Kent Dybvig. Extending the scope of syntactic abstraction. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 203–213, New York, NY, 1999.

[9] Oscar Waddell, Dipanwita Sarkar, and R. Kent Dybvig. Fixing letrec: A faithful yet efficient implementation of Scheme's recursive binding construct. *Higher Order Symbol. Comput.*, 18(3-4):299–326, 2005.