# Robust and Effective Transformation of Letrec

Oscar Waddell
owaddell@cs.indiana.edu

Dipanwita Sarkar
dsarkar@cs.indiana.edu

R. Kent Dybvig
dyb@cs.indiana.edu

Computer Science Department
Indiana University
Bloomington, IN 47408

## ABSTRACT

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

## 1. INTRODUCTION

Scheme's `letrec` permits the definition of mutually recursive procedures and, more generally, mutually recursive objects that contain procedures [2]. It is also a convenient intermediate-language representation for internal definitions and local modules [10]. When used for this purpose, the values bound by `letrec` are often a mix of procedures and nonprocedures.

A `letrec` expression has the form

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

where each $x$ is a variable and each $e$ is an arbitrary expression, often but not always a `lambda` expression. The Revised$^5$ Report on Scheme [2] defines `letrec` via the following transformation into more primitive constructs.

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

where $t_1$ ... $t_n$ are fresh temporaries.

Permission to make digital or hard copies, to republish, to post on servers or to redistribute to lists all or part of this work 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 otherwise copy or redistribute requires prior specific permission.
Third Workshop on Scheme and Functional Programming. October 3, 2002, Pittsburgh, Pennsylvania, USA.
Copyright 2002 Oscar Waddell, Dipanwita Sarkar, and R. Kent Dybvig.

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

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

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

This paper presents an alternative transformation of `letrec` that attempts to minimize the number of introduced assignments. It enables the compiler to generate efficient code while preserving the semantics of the revised report transformation. The alternative transformation is shown to eliminate most of the introduced assignments and to improve run time dramatically. The transformation incorporates a mechanism for detecting all violations of the `letrec` restriction that, in practice, has virtually zero overhead. The transformation assumes that an earlier pass of the compiler has recorded for each variable binding whether it has been referenced or assigned, and no other information is required.

This paper also investigates the implementation of a variant of `letrec`, which we call `letrec*`, that evaluates the right-hand sides from left to right and assigns each left-hand side immediately to the value of the right-hand side. It is often assumed that this would result in less efficient code; however, we show that this is not the case in practice. While there are valid software engineering reasons for leaving the evaluation order for `letrec` unspecified, `letrec*` would be a useful addition to the language and a reasonable intermediate representation for internal definitions, where left-to-right evaluation is often expected anyway.

The remainder of this paper is organized as follows. Section 2 describes our transformation in three stages, starting with a basic version, adding an assimilation mechanism for nested bindings, and adding valid checks for references and assignments to left-hand-side variables. Section 3 introduces the `letrec*` form and describes its implementation. Section 4 presents an analysis of the effectiveness of the various transformations. Section 5 describes related work. Finally, Section 6 summarizes the paper and presents our conclusions.

## 2. THE TRANSFORMATION

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

The transformation expects that bound variables in the input program are uniquely named. It also assumes that an earlier pass of the compiler has recorded information about references and assignments of the bound variables. In our implementation, these conditions are met by running input programs through the `syntax-case` macro expander [1]. If this were not the case, a simple flow-insensitive pass to perform alpha conversion and record reference and assignment information could be run prior to the transformation algorithm.

The transformation is implemented in two passes. The first performs the transformation proper, and the second introduces the code that detects violations of the `letrec` restriction.

### 2.1 Basic transformation

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

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

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

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

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

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

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

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

***simple*** if $x$ is referenced but not assigned and $e'$ is a simple expression;

***lambda*** if $x$ is referenced but not assigned and $e'$ is a lambda expression;

***unreferenced*** if no references to $x$ appear in the program;

***complex*** if it does not fall into any of the other categories.

A simple expression contains no occurrences of the variables bound by the `letrec` expression and must not be able to obtain its continuation via `call/cc`, either directly or indirectly. The former restriction is necessary because simple expressions are placed outside the scope of the bound variables. Without the latter restriction, it would be possible to detect the fact that the bindings are created after the evaluation of a simple right-hand-side expression rather than before. To enforce the latter restriction, our implementation simply rules out all procedure calls except those to certain primitives (not including `call/cc`).

A `fix` expression is a variant of `letrec` that binds only unassigned variables to `lambda` expressions. It represents the subset of `letrec` expressions that can be handled easily by later passes of a compiler. In particular, no assignments through external variables are necessary to implement mutually recursive procedures bound by `fix`. Instead, the closures produced by a `fix` expression can be block allocated and "wired" directly together. This leaves the `fix`-bound variables unassigned for the duration, thus simplifying optimizations such as inlining and loop recognition. `fix` is identical to the `labels` operator handled by Steele's Rabbit compiler [9] and the `Y` operator of Kranz's Orbit compiler [4, 3] and Rozas' Liar compiler [7, 8].

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

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

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

An unreferenced binding `[x e']` may be dropped if $e'$ is simple or a `lambda` expression, although the code generated is the same if a later pass eliminates such expressions when they are used only for effect, as is the case in our compiler.

## 2.2 Assimilating nested binding forms

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

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

is translated into

```
(let ([f (void)])
  (let ([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'd prefer a translation into the following equivalent expression.

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

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

When $e'$ is a `fix` expression `(fix (`$[\overline{x_l}\ \overline{e_l}]$` ...) `$\overline{body}$`)`, the bindings $[\overline{x_l}\ \overline{e_l}]$ ... are simply added to the *lambda* partition and the binding $[x\ \overline{body}]$ is added to the set of bindings to be partitioned.

Essentially, this transformation treats the nested bindings as if they had originally appeared in the enclosing `letrec`. For example,

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

is treated as

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

When $e'$ is a `let` expression `(let (`$[\overline{x}\ \overline{e}]$` ...) `$\overline{body}$`)` and the set of bindings $[\overline{x}\ \overline{e'}]$ ... can be fully partitioned into a set of *simple* bindings $[\overline{x_s}\ \overline{e_s}]$ ... and a set of *lambda* bindings $[\overline{x_l}\ \overline{e_l}]$ ..., we add $[\overline{x_s}\ \overline{e_s}]$ ... to the *simple* partition, $[\overline{x_l}\ \overline{e_l}]$ ... to the *lambda* partition, and $[\overline{x}\ \overline{body}]$ to the set of bindings to be partitioned.

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

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

is treated as

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

If during this process we encounter a binding $[\overline{x}\ \overline{e}]$ where $\overline{x}$ is unassigned and $\overline{e}$ is a `let` or `fix` expression, we simply fold the bindings in and continue.

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

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

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

We are prevented from assimilating cases like this because it may be possible to detect the separation of the creation of the (mutable) binding for `n` from the evaluation of the body of the nested `let` by invoking a continuation created in another of the `letrec` bindings that causes the body of the nested `let` to be evaluated multiple times. The separation cannot be detected in the given example, however, since the body of the nested `let` is a `lambda` expression, and assimilated bindings of `lambda` expressions are evaluated only once.

Because it is desirable not to penalize such uses of local state, we add an additional case to handle this situation. When $e'$ is a `let` expression `(let (`$[\overline{x}\ \overline{e}]$` ...) `$\overline{body}$`)` and the set of bindings $[\overline{x}\ \overline{e}]$ ... can be fully partitioned into a set of *simple* bindings $[\overline{x_s}\ \overline{e_s}]$ ... and a set of *lambda* bindings $[\overline{x_l}\ \overline{e_l}]$ ..., except that one or more of the variables $\overline{x_s}$ ... is assigned, and $\overline{body}$ is a `lambda` expression, we add $[\overline{x_s}\ \overline{e_s}]$ ... to the *simple* partition, $[\overline{x_l}\ \overline{e_l}]$ ... to the *lambda* partition, and $[x\ \overline{body}]$ to the set of bindings to be partitioned.

For example, when $e_a$ is a `lambda` or simple expression, $a$ is assigned, and $e_g$ is a `lambda` expression,

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

is treated as

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

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

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

## 2.3 Valid checks

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

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

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

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

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

We never liked this behavior, which fell out of an earlier version of the partitioning algorithm.

We believe it is better for an implementation to detect and report errors rather than to give meaning to technically meaningless programs. Reporting these errors also helps users create more portable programs. Fortunately, it turns out that these errors can be detected with practially no overhead, as we describe in this section.

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

It is possible to analyze the right-hand sides to determine the set of variables referenced or to perform an interprocedural flow analysis to determine the set of variables that might be undefined when referenced or assigned, by monitoring the flow of the undefined values. With this information, we could perform the transformations described in Sections 2.1 and 2.2 for all but those variables that might be undefined when referenced or assigned.

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

This flag is introduced as a binding of a fresh variable, `valid?`, wrapped around the code that evaluates the *unreferenced* and *complex* expressions. If a `letrec` has no *unreferenced* or *complex* bindings, no valid flag need be introduced. This flag is checked at each point where a valid check is deemed to be necessary. It is set initially to false, meaning that references to left-hand-side expressions are not allowed, and changed to true once control enters the body of the `letrec`.

```
(let ([x_s e_s] ... [x_c (void)] ...)
  (fix ([x_l e_l] ...)
    (let ([valid? #f])
      e_u ...
      (let ([x_t e_c] ...)
        (set! x_c x_t)
        ...)
      (set! valid? #t))
    body'))
```

In a naive implementation, valid checks would be inserted at each reference and assignment to one of the left-hand-side variables within the *unreferenced* and *complex* expressions. A valid check simply tests `valid?` and signals an error if `valid?` is false. For each valid check for a variable `x`, the valid check appears as follows.

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

No checks need to be inserted in the body of the `letrec`, since the bindings are necessarily valid once control enters the body. No checks are required within the right-hand sides of *lambda* bindings, since control cannot enter the body of one of these `lambdas` except by way of a reference to the corresponding left-hand-side variable. *Simple* bindings contain no references to the left-hand-side variables.

We can do even better than to limit the valid checks to the right-hand sides of *unreferenced* and *complex* bindings. To do so, we introduce the notion of *protected* and *unprotected* references. A reference (or assignment) to a variable is protected if it is contained within a `lambda` expression that cannot be evaluated and invoked during the evaluation of an expression. Otherwise, it is unprotected.

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

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

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

This handles well situations such as

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

in which `f` is a sort of locative [6] for `x`. Since `cons` does not invoke its arguments, the references appearing within the `lambda` expressions are protected.

It doesn't handle situations such as the following.

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

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

The original `letrec` expressions no longer exist by the time the second pass runs, so the first pass must leave behind sufficient information to allow the second pass to know which are the original `letrec`-bound variables and which expressions may require the insertion of valid checks. The actual output of the first pass is therefore as follows

```
(let ([xs es] ... [xc (void)] ...)
  (fix ([xl el] ...)
    (bind-valid-flag (x ...)
      eu ...
      (let ([xt ec] ...)
        (valid-set! xc xt)
        ...))
    body'))
```

where $x$ `...` is the original list of `letrec`-bound variables. The `bind-valid-flag` expression expands into a `let` expression binding the variable `valid?` if any valid checks were inserted, otherwise it expands into the code in its body. It also inserts the assignment to set the valid flag true at the end of its body if the valid flag is introduced. The `valid-set!` expression is used in place of `set!` for the introduced assignments to the *complex* variables; this tells the second pass that this is already known to be valid so that no valid check is inserted for the assignment.

## 3. FIXED EVALUATION ORDER

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

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

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

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

One can nest binding contours to order bindings, but this is often inconvenient and prevents the sequenced bindings from being mutually recursive. It is therefore interesting to consider a variant of `letrec` that performs its bindings in a left-to-right fashion. Scheme provides a variant of `let`, called `let*`, that sequences evaluation of `let` bindings; we therefore call our version of `letrec` that sequences `letrec` bindings `letrec*`. The analogy to `let*` is imperfect, since `let*` also nests scopes whereas `letrec*` maintains the mutual recursive scoping of `letrec`.

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

```
(letrec* ([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` binding a set of temporaries to the right-hand-side expressions. This transformation would be incorrect for `letrec`, since the assignments are not all in the continuation of each right-hand-side expression, as in the revised report transformation. Thus, `call/cc` could be used to expose the difference between the two transformations.

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

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

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

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

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

On the other hand, a substantial change to the introduction of valid flags is necessary to handle the different semantics of `letrec*`. This change is to introduce one valid flag for each *unreferenced* and *complex* right-hand side, in contrast to one per `letrec` expression. The valid flag for a given expression represents the validity of references and assignments to the corresponding variable and all subsequent variables bound by the `letrec`. This may result in the introduction of more valid flags but should not result in the introduction of any additional valid checks. Due to the nature of `letrec*`, in fact, there will likely be fewer valid checks and possibly fewer actual valid-flag bindings.

As with `letrec`, the first pass of the transformation algorithm inserts `bind-valid-flag` expressions to tell the second pass where to insert valid flags and checks. If there are no unreferenced bindings, the output is as follows

```
(let ([x_s e_s] ... [x_c (void)] ...)
  (fix ([x_l e_l] ...)
    (set! x_c
      (bind-valid-flag (x_c+ ...)
        e_c))
    ...
    body'))
```

where $x_c+$ ... represents the sublist of original left-hand-side variables from $x_c$ on. If there are *unreferenced* bindings, the right-hand sides are inserted into the code in the proper sequence, each wrapped in a `bind-valid-flag` expression that lists all variables from the next referenced variable on.

The second pass operates as before: no changes are needed to support `letrec*`.

# 4. RESULTS

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

We measured the performance of the benchmark programs using several transformations:

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

Not surprisingly, the benchmark programs still run in the system that treats `letrec` as `letrec*`, since none contain code that detects the failure of that system to be faithful to the Revised$^5$ Report transformation. (Some of the tests in our test suite did fail, but only because they were there to keep our compiler honest in this regard.)

We compare these systems along several dimensions: run time, compile time, code size, number of introduced assignments, number of valid checks, and numbers of bindings classified as *lambda*, *complex*, *simple*, and *unreferenced*. Run times were determined by averaging three runs for each benchmark; programs were configured so that each run required at least two seconds. Code size was determined by recording the size of the actual code objects written to compiled files. Compile times were recorded for a single compilation of each benchmark, with the exception of the compiler bootstrapping benchmark (chezscheme), where three such runs were averaged. With the exception of `chezscheme`, `similix`, and `texer`, each benchmark was placed within a `module` form, converting top-level definitions to internal definitions. A few programs that relied on left-to-right evaluation of top-level definitions were edited so that they could run successfully in all of the systems.

The results are given in Tables 1–4. Programs in these tables are listed in sorted order, with larger programs (in terms of object code) after smaller ones. The run-time results show that the transformation is successful in reducing run-time overhead in many cases and never increases overhead, even with valid checks enabled. Using the "easy" transformation to catch pure `letrec` expressions is also effective, but our transformation is even more effective, with noticible improvements on several benchmarks, including `lattice-jw`, `ray`, `maze`, and `conform`.

Using our algorithm, run times are almost identical with or without valid checks, so strict enforcement of the `letrec` restriction is achieved with practically no overhead. Most of the benchmarks require no valid flags and few require a substantial number of valid checks. In contrast, naive valid checks significantly reduce the performance of the $R^5RS$ and "easy" transformations in some cases.

For our compiler, the most substantial program in our test suite, assimilating nested bindings allows the transformation to decrease the number of introduced assignments by 17%. Moreover, this allows the transformation to eliminate all of the valid checks that would otherwise be inserted. Assimilation of nested bindings does not seem to benefit run times, however. This is somewhat disappointing, but may simply indicate that few of the benchmarks try to express scoping relationships more tightly, perhaps even because of a fear that the resulting code would not be as efficient. We believe it is an important optimization, nevertheless, as one of many "bullets in [the compiler's] gun" [5] that are not generally applicable but are very useful in certain circumstances.

Compile time increases are modest for our algorithm, with or without valid checks and assimilation. In many cases, the compile times are less, even though more effort is clearly expended in the new passes than is required to do the $R^5RS$ transformation. This is because our transformation enables more optimizations by later passes, leading to smaller code and an overall reduction in compile times.

The numbers for `letrec*` indicate that there is no overhead in practice for fixing the order of evaluation, even though our compiler reorders expressions when possible to improve the generated code. This is likely due in part to the relatively few cases where our translation of `letrec*` actually introduces constraints on the evaluation order. In addition, almost no valid flags and checks are required for `letrec*`. So while the implementation of `letrec*` may require more valid flags in principle, it requires fewer in practice, since the fixed evaluation order eliminates the need for most valid checks and the flags used to support them.

As shown in Table 1, the "easy" algorithm, which is attractive for its simplicity, often introduces many more assignments than are necessary, since not all `letrec` bindings are `lambda` expressions. Naively enforcing the `letrec` restriction also introduces far more valid checks than necessary, even when pure `letrec` expressions are recognized.

Our algorithm identifies "simple" bindings in many of the benchmarks and avoids introducing assignments for these. Moreover, it avoids introducing assignments for pure `lambda` bindings that happen to be bound by the same `letrec` that binds a simple binding. In several cases, assimilating nested `let` and `letrec` bindings allows the algorithm to assign more of the bindings to the *lambda* or *simple* partitions.

## 5. RELATED WORK

Much has been written about generating efficient code for ideal recursive binding forms, like our `fix` construct or the `Y` combinator, that bind only `lambda` expressions. Yet virtually nothing has been written explaining how to cope with the reality of arbitrary `letrec` expressions, e.g., by transforming them into one of these ideal forms. Moreover, nothing has been written describing efficient strategies for detecting violations of the "`letrec` restriction."

Steele [9] developed strategies for generating good code for mutually recursive procedures bound by a `labels` form that is essentially our `fix` construct. Because `labels` forms are present in the input language handled by his compiler, he does not describe the translation of general `letrec` expressions into `labels`.

Kranz [4, 3] also describes techniques for generating efficient code for mutually recursive procedures expressed in terms of the `Y` operator. He describes a macro transformation of `letrec` that introduces assignments for any right-hand side that is not a `lambda` expression and uses `Y` to handle those that are `lambda` expressions. This transformation introduces unnecessary assignments for bindings that our algorithm would deem *simple*. His transformation does not attempt to assimilate nested binding constructs. The `Y` operator is a primitive construct recognized by his compiler, much as `fix` is recognized by our compiler.

Rozas [7, 8] shows how to generate good code for mutually recursive procedures expressed in terms of `Y` without recognizing `Y` as a primitive construct, that is, with `Y` itself expressed at the source level. He does not discuss the process of converting `letrec` into this form.

## 6. CONCLUSION

We have presented an algorithm for transforming `letrec` expressions into a form that enables the generation of efficient code while preserving the semantics of the `letrec` transformation given in the Revised[5] Report on Scheme [2]. The transformation avoids many of the assignments produced by the Revised[5] Report transformation by converting many of the `letrec` bindings into simple `let` bindings or into a "pure" form of `letrec`, called `fix`, that binds only unassigned variables to `lambda` expressions. `fix` expressions are the basis for several optimizations, including block allocation and internal wiring of closures. We have shown the algorithm to be effective at reducing the number of introduced assignments and improving run time with little compile-time overhead.

The algorithm also inserts "valid checks" to implement the `letrec` restriction that no reference or assignment to a left-hand-side variable can be evaluated in the process of evaluating the right-hand-side expressions. It inserts few checks in practice and adds practically no overhead to the evaluation of programs that use `letrec`. More importantly, it does not inhibit the optimizations performed by subsequent passes. Most Scheme implementations currently omit such checks, but this paper shows that the checks can be performed even in compilers that are geared toward high-performance applications.

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

While treating `letrec` expressions as `letrec*` clearly violates the Revised$^5$ Report semantics for `letrec`, we wonder if future versions of the standard shouldn't require that internal definitions be treated as `letrec*` rather than `letrec`. Left-to-right evaluation order of definitions is often what programmers expect and would make the semantics of internal definitions more consistent with external definitions. We have shown that there would be no significant performance penalty for this in practice.

# 7. REFERENCES


[1] Dybvig, R. K., Hieb, R., and Bruggeman, C. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation 5*, 4 (1993), 295–326.

[2] Kelsey, R., Clinger, W., and Rees, J. A. Revised$^5$ report on the algorithmic language Scheme. *SIGPLAN Notices 33*, 9 (1998), 26–76.

[3] Kranz, D. A. *Orbit, An Optimizing Compiler for Scheme*. PhD thesis, Yale University, May 1988.

[4] Kranz, D. A., Kelsey, R., Rees, J. A., Hudak, P., Philbin, J., and Adams, N. I. Orbit: an optimizing compiler for Scheme. *SIGPLAN Notices, ACM Symposium on Compiler Construction 21*, 7 (1986), 219–233.

[5] Peyton Jones, S. L., and Santos, A. A transformation-based optimiser for Haskell. *Science of Computer Programming 32*, 1–3 (1998), 3–47.

[6] Rees, J. A., Adams, N. I., and Meehan, J. R. *The T Manual*. Yale University, New Haven, Connecticut, USA, 1984. Fourth edition.

[7] Rozas, G. J. Liar, an Algol-like compiler for Scheme. S. B. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Jan. 1984.

[8] Rozas, G. J. Taming the Y operator. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming* (San Francisco, USA, June 1992), pp. 226–234.

[9] Steele Jr., G. L. Rabbit: a compiler for Scheme. MIT AI Memo 474, Massachusetts Institute of Technology, Cambridge, Mass., May 1978.

[10] Waddell, O., and Dybvig, R. K. Extending the scope of syntactic abstraction. In *Conference Record of the Twenty Sixth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1999), pp. 203–213.

| | Introduced assignments | | | | | Valid checks | | | | | # bindings in each partition | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | easy | | A | | | | N | | | |
| | $R^5RS$ | easy | A | N | S | $R^5RS$ | easy | A | N | S | $\lambda$ | $c$ | $\lambda$ | $c$ | $s$ | $u$ | $\lambda$ | $c$ | $s$ | $u$ |
| fxtak | 2 | – | – | – | – | 5 | – | – | – | – | 2 | – | 2 | – | – | – | 2 | – | – | – |
| tak | 2 | – | – | – | – | 5 | – | – | – | – | 2 | – | 2 | – | – | – | 2 | – | – | – |
| div-iter | 10 | – | – | – | – | 14 | – | – | – | – | 10 | – | 9 | – | – | 1 | 9 | – | – | 1 |
| cpstak | 3 | – | – | – | – | 5 | – | – | – | – | 3 | – | 3 | – | – | – | 3 | – | – | – |
| takl | 7 | 3 | 3 | 3 | 3 | 8 | – | – | – | – | 4 | 3 | 4 | 3 | – | – | 4 | 3 | – | – |
| ctak | 3 | – | – | – | – | 6 | – | – | – | – | 3 | – | 3 | – | – | – | 3 | – | – | – |
| mbrot | 8 | – | – | – | – | 7 | – | – | – | – | 8 | – | 8 | – | – | – | 8 | – | – | – |
| deriv | 4 | – | – | – | – | 8 | – | – | – | – | 4 | – | 4 | – | – | – | 4 | – | – | – |
| destruct | 9 | – | – | – | – | 8 | – | – | – | – | 9 | – | 9 | – | – | – | 9 | – | – | – |
| fxtriang | 13 | 7 | 4 | 4 | 4 | 6 | – | – | – | – | 6 | 7 | 6 | 4 | 3 | – | 6 | 4 | 3 | – |
| fft-f | 8 | – | – | – | – | 7 | – | – | – | – | 8 | – | 8 | – | – | – | 8 | – | – | – |
| fft-d | 11 | – | – | – | – | 12 | – | – | – | – | 11 | – | 11 | – | – | – | 11 | – | – | – |
| dderiv | 8 | – | – | – | – | 4 | – | – | – | – | 8 | – | 8 | – | – | – | 8 | – | – | – |
| triang | 13 | 7 | 4 | 4 | 4 | 6 | – | – | – | – | 6 | 7 | 6 | 4 | 3 | – | 6 | 4 | 3 | – |
| lattice | 22 | 14 | – | 1 | – | 37 | 29 | – | – | – | 8 | 14 | 21 | – | 2 | – | 19 | 1 | 2 | – |
| boyer | 25 | 23 | 2 | 2 | 2 | 50 | 48 | – | – | – | 2 | 23 | 22 | 2 | – | 1 | 22 | 2 | – | 1 |
| boyer-jw | 23 | 22 | 4 | 4 | 4 | 67 | 66 | – | – | – | 1 | 22 | 19 | 4 | – | – | 19 | 4 | – | – |
| browse | 20 | 8 | 1 | 1 | 1 | 33 | 21 | – | – | – | 12 | 8 | 19 | 1 | – | – | 19 | 1 | – | – |
| traverse | 47 | 38 | 5 | 5 | 5 | 69 | 60 | – | – | – | 9 | 38 | 39 | 5 | – | 3 | 39 | 5 | – | 3 |
| lattice-jw | 22 | 10 | – | 1 | – | 33 | 19 | – | – | – | 12 | 10 | 23 | – | – | – | 21 | 1 | – | – |
| fft-g | 11 | 4 | – | – | – | 9 | 2 | – | – | – | 7 | 4 | 8 | – | 3 | – | 8 | – | 3 | – |
| ray | 34 | 27 | 2 | 2 | 2 | 92 | 84 | – | – | – | 7 | 27 | 32 | 2 | – | – | 32 | 2 | – | – |
| fxpuzzle | 34 | 11 | 11 | 11 | 11 | 20 | – | – | – | – | 23 | 11 | 22 | 11 | – | 1 | 22 | 11 | – | 1 |
| graphs | 32 | – | – | – | – | 38 | – | – | – | – | 32 | – | 28 | – | – | 4 | 28 | – | – | 4 |
| tcheck | 35 | 32 | 3 | 3 | 3 | 84 | 79 | – | – | – | 3 | 32 | 22 | 3 | 12 | – | 22 | 3 | 10 | – |
| simplex | 32 | 1 | – | – | – | 67 | – | – | – | – | 31 | 1 | 31 | – | 1 | – | 31 | – | 1 | – |
| graphs-jw | 20 | – | – | – | – | 24 | – | – | – | – | 20 | – | 20 | – | – | – | 20 | – | – | – |
| maze | 83 | 62 | 1 | 1 | 1 | 183 | 161 | – | – | – | 21 | 62 | 68 | 1 | 3 | 11 | 68 | 1 | 3 | 11 |
| maze-jw | 85 | – | – | – | – | 37 | – | – | – | – | 85 | – | 74 | – | – | 11 | 74 | – | – | 11 |
| puzzle | 34 | 11 | 11 | 11 | 11 | 20 | – | – | – | – | 23 | 11 | 22 | 11 | – | 1 | 22 | 11 | – | 1 |
| earley | 75 | – | – | – | – | 117 | – | – | – | – | 75 | – | 73 | – | – | 2 | 73 | – | – | 2 |
| splay | 13 | – | – | – | – | 17 | – | – | – | – | 13 | – | 13 | – | – | – | 13 | – | – | – |
| matrix | 49 | 26 | – | 2 | – | 64 | 34 | – | – | – | 23 | 26 | 49 | – | – | 2 | 45 | 2 | – | 2 |
| conform | 104 | 82 | 5 | 5 | 5 | 261 | 240 | – | – | – | 22 | 82 | 91 | 5 | 5 | 3 | 91 | 5 | 5 | 3 |
| matrix-jw | 37 | 10 | – | 1 | – | 50 | 14 | – | – | – | 27 | 10 | 38 | – | – | – | 36 | 1 | – | – |
| peval | 55 | 41 | 3 | 3 | 3 | 175 | 134 | – | – | – | 14 | 41 | 44 | 3 | 8 | – | 44 | 3 | 8 | – |
| nucleic-sorted | 265 | 236 | 2 | 2 | 2 | 7 | 2 | – | – | – | 29 | 236 | 124 | 2 | 60 | 79 | 124 | 2 | 60 | 79 |
| nucleic-star | 265 | 260 | 5 | 5 | 5 | 743 | 738 | – | – | – | 5 | 260 | 124 | 5 | 57 | 79 | 124 | 5 | 57 | 79 |
| fxtakr | 101 | – | – | – | – | 401 | – | – | – | – | 101 | – | 101 | – | – | – | 101 | – | – | – |
| em-imp | 103 | 47 | 1 | 1 | 1 | 204 | 148 | – | – | – | 56 | 47 | 94 | 1 | 7 | 1 | 94 | 1 | 7 | 1 |
| nucleic-jw | 48 | 34 | 5 | 5 | 5 | 173 | 160 | 80 | 80 | – | 14 | 34 | 38 | 5 | 4 | 1 | 38 | 5 | 4 | 1 |
| em-fun | 102 | 62 | 1 | 1 | 1 | 264 | 224 | – | – | – | 40 | 62 | 94 | 1 | 7 | – | 94 | 1 | 7 | – |
| lalr | 349 | 292 | 3 | 6 | 3 | 303 | 245 | – | – | – | 57 | 292 | 186 | 3 | 16 | 163 | 166 | 6 | 15 | 163 |
| takr | 101 | – | – | – | – | 401 | – | – | – | – | 101 | – | 101 | – | – | – | 101 | – | – | – |
| nbody | 58 | 6 | – | – | – | 79 | – | – | – | – | 52 | 6 | 56 | – | 2 | – | 56 | – | 2 | – |
| interpret | 122 | 110 | 1 | 1 | 1 | 267 | 251 | – | – | – | 12 | 110 | 119 | 1 | 2 | – | 119 | 1 | 2 | – |
| dynamic | 201 | 187 | 2 | 2 | 2 | 561 | 548 | – | – | – | 14 | 187 | 144 | 2 | 41 | 14 | 144 | 2 | 41 | 14 |
| texer | 146 | 80 | 13 | 18 | 13 | 592 | 497 | – | – | – | 66 | 80 | 132 | 13 | 7 | – | 126 | 18 | 2 | – |
| similix | 527 | 141 | – | 1 | – | 1705 | 322 | – | – | – | 386 | 141 | 484 | – | 36 | 8 | 483 | 1 | 35 | 8 |
| ddd | 1161 | 550 | 14 | 45 | 14 | 3063 | 2164 | 2 | 2 | 2 | 611 | 550 | 1182 | 14 | 10 | 124 | 982 | 45 | 10 | 124 |
| softscheme | 1049 | 865 | 132 | 134 | 132 | 3382 | 2793 | – | 8 | – | 184 | 865 | 858 | 132 | 47 | 147 | 798 | 134 | 43 | 147 |
| chezscheme | 2411 | 1289 | 140 | 169 | 140 | 6051 | 4339 | – | 18 | – | 1122 | 1289 | 2137 | 140 | 110 | 114 | 2039 | 169 | 89 | 114 |

**Table 1: Number of introduced assignments and valid checks for the straightforward $R^5RS$ transformation, the modified $R^5RS$ transformation (easy) described in Section 4, and for the Assimilating (A), Non-assimilating (N), and Sequential `letrec*` (S) variants of our transformation. Also shown are the number of bindings in the *lambda* ($\lambda$), *complex* ($c$), *simple* ($s$), and *unreferenced* ($u$), partitions for the modified $R^5RS$ transformation and for our transformation with and without assimilation. (All bindings are *complex* in standard $R^5RS$ transformation.) Since assimilation incorporates both nested `let` and `letrec` bindings, the total number of bindings may be greater when assimilation is enabled.**

| | R$^5$RS | | R$^5$RS easy | | A | N | A | N | S |
|---|---|---|---|---|---|---|---|---|---|
| Checks: | no | naive | no | naive | yes | yes | no | no | yes |
| fxtak | 1.00 | 1.16 | .81 | .81 | .81 | .81 | .81 | .81 | .81 |
| tak | 1.00 | 1.12 | .87 | .87 | .87 | .87 | .87 | .87 | .87 |
| div-iter | 1.00 | 1.05 | .93 | .93 | .93 | .93 | .93 | .93 | .93 |
| cpstak | 1.00 | 1.03 | .85 | .85 | .85 | .85 | .85 | .85 | .85 |
| takl | 1.00 | 1.23 | .71 | .71 | .71 | .71 | .71 | .71 | .71 |
| ctak | 1.00 | 1.02 | .89 | .89 | .89 | .89 | .89 | .89 | .89 |
| mbrot | 1.00 | 1.01 | .99 | .99 | .98 | .99 | .99 | .98 | .99 |
| deriv | 1.00 | 1.02 | .97 | .97 | .96 | .96 | .96 | .96 | .96 |
| destruct | 1.00 | 1.05 | .81 | .82 | .81 | .81 | .81 | .81 | .81 |
| fxtriang | 1.00 | 1.15 | .87 | .87 | .81 | .80 | .80 | .80 | .81 |
| fft-f | 1.00 | 1.01 | .91 | .91 | .91 | .91 | .91 | .91 | .91 |
| fft-d | 1.00 | 1.00 | .99 | .99 | .99 | .99 | .99 | .99 | .99 |
| dderiv | 1.00 | 1.03 | .94 | .94 | .94 | .94 | .94 | .94 | .94 |
| triang | 1.00 | 1.11 | .90 | .91 | .85 | .85 | .85 | .85 | .85 |
| lattice | 1.00 | 1.04 | .53 | .54 | .52 | .53 | .52 | .53 | .52 |
| boyer | 1.00 | 1.13 | .88 | .92 | .86 | .86 | .86 | .86 | .86 |
| boyer-jw | 1.00 | 1.18 | 1.00 | 1.18 | .96 | .96 | .96 | .96 | .96 |
| browse | 1.00 | 1.01 | .97 | .97 | .94 | .94 | .94 | .94 | .94 |
| traverse | 1.00 | 1.10 | 1.05 | 1.09 | .97 | .97 | .97 | .97 | .97 |
| lattice-jw | 1.00 | 1.04 | .80 | .84 | .28 | .28 | .28 | .28 | .28 |
| fft-g | 1.00 | 1.01 | .88 | .89 | .88 | .88 | .88 | .88 | .88 |
| ray | 1.00 | 1.09 | .99 | 1.06 | .76 | .75 | .75 | .75 | .76 |
| fxpuzzle | 1.00 | 1.15 | .76 | .76 | .77 | .77 | .77 | .77 | .77 |
| graphs | 1.00 | 1.00 | .39 | .60 | .39 | .39 | .39 | .39 | .39 |
| tcheck | 1.00 | 1.01 | .99 | 1.00 | .96 | .96 | .96 | .96 | .96 |
| simplex | 1.00 | 1.06 | .55 | .56 | .54 | .54 | .54 | .54 | .54 |
| graphs-jw | 1.00 | 1.01 | .54 | .54 | .54 | .54 | .54 | .54 | .54 |
| maze | 1.00 | 1.12 | .79 | .83 | .55 | .55 | .55 | .55 | .55 |
| maze-jw | 1.00 | 1.03 | .70 | .70 | .70 | .70 | .70 | .70 | .70 |
| puzzle | 1.00 | 1.10 | .89 | .88 | .88 | .88 | .88 | .88 | .88 |
| earley | 1.00 | 1.03 | .73 | .73 | .73 | .73 | .73 | .73 | .73 |
| splay | 1.00 | 1.00 | .77 | .77 | .77 | .77 | .77 | .77 | .77 |
| matrix | 1.00 | .99 | .62 | .63 | .59 | .59 | .59 | .59 | .59 |
| conform | 1.00 | 1.13 | .92 | 1.09 | .38 | .38 | .38 | .38 | .38 |
| matrix-jw | 1.00 | 1.01 | .68 | .68 | .60 | .60 | .60 | .60 | .60 |
| peval | 1.00 | 1.08 | .93 | .98 | .78 | .78 | .78 | .78 | .78 |
| nucleic-sorted | 1.00 | .99 | .98 | .99 | .74 | .74 | .74 | .74 | .74 |
| nucleic-star | 1.00 | 1.08 | 1.00 | 1.09 | .76 | .76 | .76 | .76 | .76 |
| fxtakr | 1.00 | 1.56 | .72 | .73 | .73 | .73 | .72 | .73 | .73 |
| em-imp | 1.00 | 1.05 | .75 | .77 | .66 | .66 | .66 | .66 | .66 |
| nucleic-jw | 1.00 | 1.00 | 1.00 | 1.00 | .99 | .98 | .98 | .98 | .99 |
| em-fun | 1.00 | 1.04 | .77 | .81 | .69 | .69 | .69 | .69 | .69 |
| lalr | 1.00 | 1.03 | .89 | .90 | .82 | .81 | .82 | .81 | .82 |
| takr | 1.00 | 1.17 | .56 | .56 | .56 | .56 | .56 | .56 | .56 |
| nbody | 1.00 | 1.01 | .72 | .72 | .66 | .66 | .66 | .66 | .66 |
| interpret | 1.00 | 1.20 | 1.02 | 1.00 | .90 | .90 | .91 | .91 | .90 |
| dynamic | 1.00 | 1.01 | .97 | 1.01 | .93 | .93 | .93 | .93 | .93 |
| texer | 1.00 | .91 | .55 | .58 | .53 | .53 | .53 | .53 | .53 |
| similix | 1.00 | 1.02 | 1.00 | 1.01 | .97 | .97 | .96 | .96 | .96 |
| ddd | 1.00 | 1.03 | 1.00 | .99 | .97 | .96 | .97 | .96 | .98 |
| softscheme | 1.00 | 1.17 | .96 | 1.14 | .79 | .79 | .79 | .80 | .79 |
| chezscheme | 1.00 | 1.10 | .75 | .84 | .65 | .66 | .66 | .65 | .65 |

**Table 2: Run time of the code produced by the various algorithms, normalized to the R$^5$RS baseline.**

| | R$^5$RS | | R$^5$RS easy | | A | N | A | N | S |
|---|---|---|---|---|---|---|---|---|---|
| Checks: | no | naive | no | naive | yes | yes | no | no | yes |
| fxtak | 1.00 | 1.30 | .90 | .90 | .90 | .90 | .90 | .90 | .90 |
| tak | 1.00 | 1.24 | .91 | .91 | .91 | .91 | .91 | .91 | .91 |
| div-iter | 1.00 | 1.21 | .47 | .55 | .47 | .47 | .47 | .47 | .47 |
| cpstak | 1.00 | 1.18 | .93 | .75 | .93 | .93 | .93 | .93 | .93 |
| takl | 1.00 | 1.24 | .83 | .83 | .83 | .83 | .83 | .83 | .83 |
| ctak | 1.00 | 1.19 | .95 | .95 | .95 | .95 | .95 | .95 | .95 |
| mbrot | 1.00 | 1.15 | .72 | .80 | .72 | .72 | .72 | .72 | .72 |
| deriv | 1.00 | 1.19 | .96 | .96 | .96 | .96 | .96 | .96 | .96 |
| destruct | 1.00 | 1.12 | .68 | .74 | .68 | .68 | .68 | .68 | .68 |
| fxtriang | 1.00 | 1.10 | .84 | .86 | .77 | .77 | .77 | .77 | .77 |
| fft-f | 1.00 | 1.11 | .69 | .73 | .69 | .69 | .69 | .69 | .69 |
| fft-d | 1.00 | 1.17 | .82 | .84 | .82 | .82 | .82 | .82 | .82 |
| dderiv | 1.00 | 1.07 | 1.15 | 1.15 | 1.15 | 1.15 | 1.15 | 1.15 | 1.15 |
| triang | 1.00 | 1.08 | .88 | .90 | .83 | .83 | .83 | .83 | .83 |
| lattice | 1.00 | 1.29 | .94 | 1.20 | .61 | .64 | .61 | .64 | .61 |
| boyer | 1.00 | 1.39 | .99 | 1.37 | .63 | .63 | .63 | .63 | .63 |
| boyer-jw | 1.00 | 1.52 | 1.00 | 1.52 | .76 | .76 | .76 | .76 | .76 |
| browse | 1.00 | 1.24 | .91 | 1.07 | .76 | .76 | .76 | .76 | .76 |
| traverse | 1.00 | 1.35 | .99 | 1.22 | .57 | .57 | .57 | .57 | .57 |
| lattice-jw | 1.00 | 1.21 | .92 | 1.06 | .73 | .76 | .73 | .76 | .73 |
| fft-g | 1.00 | 1.07 | 1.05 | 1.08 | .98 | .98 | .98 | .98 | .98 |
| ray | 1.00 | 1.40 | .97 | 1.29 | .54 | .54 | .54 | .54 | .54 |
| fxpuzzle | 1.00 | 1.13 | .74 | .76 | .74 | .74 | .74 | .74 | .74 |
| graphs | 1.00 | 1.17 | .72 | .83 | .72 | .72 | .72 | .72 | .72 |
| tcheck | 1.00 | 1.43 | .98 | 1.38 | .77 | .77 | .77 | .77 | .77 |
| simplex | 1.00 | 1.27 | .60 | .65 | .59 | .59 | .59 | .59 | .59 |
| graphs-jw | 1.00 | 1.10 | .71 | .71 | .71 | .71 | .71 | .71 | .71 |
| maze | 1.00 | 1.46 | .94 | 1.29 | .46 | .46 | .46 | .46 | .46 |
| maze-jw | 1.00 | 1.10 | .46 | .46 | .46 | .46 | .46 | .46 | .46 |
| puzzle | 1.00 | 1.09 | .87 | .88 | .87 | .87 | .87 | .87 | .87 |
| earley | 1.00 | 1.28 | .49 | .52 | .49 | .49 | .49 | .49 | .49 |
| splay | 1.00 | 1.08 | .86 | .86 | .86 | .86 | .86 | .86 | .86 |
| matrix | 1.00 | 1.18 | .91 | 1.01 | .67 | .71 | .67 | .71 | .67 |
| conform | 1.00 | 1.49 | .94 | 1.41 | .51 | .51 | .51 | .51 | .51 |
| matrix-jw | 1.00 | 1.15 | .88 | .92 | .80 | .80 | .80 | .80 | .80 |
| peval | 1.00 | 1.40 | .95 | 1.26 | .76 | .76 | .76 | .76 | .76 |
| nucleic-sorted | 1.00 | 1.01 | .96 | .97 | .39 | .39 | .39 | .39 | .39 |
| nucleic-star | 1.00 | 1.56 | .99 | 1.55 | .40 | .40 | .40 | .40 | .40 |
| fxtakr | 1.00 | 1.54 | .59 | .59 | .59 | .59 | .59 | .59 | .59 |
| em-imp | 1.00 | 1.25 | .93 | 1.07 | .66 | .66 | .66 | .66 | .66 |
| nucleic-jw | 1.00 | 1.32 | .95 | 1.25 | .90 | .90 | .64 | .64 | .64 |
| em-fun | 1.00 | 1.33 | .99 | 1.24 | .68 | .68 | .68 | .68 | .68 |
| lalr | 1.00 | 1.18 | .92 | 1.06 | .53 | .54 | .53 | .54 | .53 |
| takr | 1.00 | 1.38 | .72 | .72 | .72 | .72 | .72 | .72 | .72 |
| nbody | 1.00 | 1.10 | 1.01 | 1.01 | .98 | .98 | .98 | .98 | .98 |
| interpret | 1.00 | 1.30 | .99 | 1.27 | 1.09 | 1.09 | 1.09 | 1.09 | 1.09 |
| dynamic | 1.00 | 1.34 | 1.02 | 1.36 | 1.16 | 1.16 | 1.16 | 1.16 | 1.16 |
| texer | 1.00 | 1.28 | .94 | 1.16 | .93 | .93 | .93 | .93 | .93 |
| similix | 1.00 | 1.19 | .94 | .98 | .91 | .91 | .91 | .91 | .91 |
| ddd | 1.00 | 1.21 | .93 | 1.08 | .81 | .87 | .78 | .87 | .81 |
| softscheme | 1.00 | 1.23 | .98 | 1.18 | .87 | .89 | .87 | .88 | .87 |
| chezscheme | 1.00 | 1.18 | .98 | 1.11 | .96 | .97 | .96 | .97 | .96 |

**Table 3: Size of the object code produced by the various algorithms, normalized to the R$^5$RS baseline.**

| | R⁵RS | | R⁵RS easy | | A | N | A | N | S |
|---|---|---|---|---|---|---|---|---|---|
| Checks: | no | naive | no | naive | yes | yes | no | no | yes |
| fxtak | 1.00 | 1.00 | .50 | 1.00 | .50 | .50 | 1.00 | .50 | .50 |
| tak | 1.00 | .50 | .50 | 1.00 | .50 | .50 | .50 | 1.00 | .50 |
| div-iter | 1.00 | .50 | 1.00 | .50 | 1.00 | .50 | .50 | .50 | .50 |
| cpstak | 1.00 | 1.00 | 1.00 | 1.00 | .50 | 1.00 | 1.00 | .50 | 1.00 |
| takl | 1.00 | 1.00 | .50 | 1.00 | 1.00 | 1.00 | .50 | .50 | 1.00 |
| ctak | 1.00 | .50 | .50 | 1.00 | .50 | .50 | 1.00 | .50 | 1.00 |
| mbrot | 1.00 | 1.00 | 1.00 | .67 | .67 | 1.00 | .67 | 1.00 | 1.00 |
| deriv | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | .50 | .50 | 1.00 |
| destruct | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| fxtriang | 1.00 | 1.00 | 1.00 | 1.50 | 1.50 | 1.00 | 1.00 | .50 | 1.00 |
| fft-f | 1.00 | .75 | .75 | .75 | .75 | .75 | .75 | .75 | .75 |
| fft-d | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | .50 | 1.00 | 1.00 |
| dderiv | 1.00 | 2.00 | 2.00 | 2.00 | 2.00 | 1.00 | 1.00 | 2.00 | 1.00 |
| triang | 1.00 | 1.50 | 1.50 | 1.00 | .50 | 1.00 | 1.00 | 1.00 | 1.50 |
| lattice | 1.00 | 1.33 | 1.00 | 1.00 | .67 | 1.00 | 1.00 | 1.00 | 1.00 |
| boyer | 1.00 | 1.25 | 1.00 | 1.00 | 1.00 | .50 | .75 | .50 | .50 |
| boyer-jw | 1.00 | 1.67 | 1.00 | 1.67 | 1.33 | 1.33 | .67 | .67 | 1.33 |
| browse | 1.00 | 1.33 | .67 | 1.00 | .67 | 1.00 | 1.00 | .67 | 1.00 |
| traverse | 1.00 | 1.25 | 1.00 | 1.25 | .75 | .75 | .75 | .50 | .75 |
| lattice-jw | 1.00 | .75 | 1.00 | .75 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| fft-g | 1.00 | 1.00 | 1.50 | 1.50 | 1.50 | 1.00 | 1.50 | 1.50 | 1.00 |
| ray | 1.00 | 1.17 | 1.00 | .83 | .67 | .83 | .83 | .83 | .83 |
| fxpuzzle | 1.00 | 1.33 | 1.33 | 1.33 | 1.33 | 1.33 | 1.00 | 1.33 | 1.33 |
| graphs | 1.00 | 1.00 | 1.00 | .80 | 1.00 | 1.00 | .80 | 1.00 | 1.00 |
| tcheck | 1.00 | 1.50 | 1.00 | 1.25 | 1.00 | 1.00 | 1.00 | 1.25 | 1.25 |
| simplex | 1.00 | 1.14 | .86 | 1.00 | 1.00 | .86 | .86 | .86 | .86 |
| graphs-jw | 1.00 | .83 | .83 | .83 | .83 | .83 | .83 | .83 | .67 |
| maze | 1.00 | 1.56 | 1.11 | 1.33 | .89 | .89 | .89 | .89 | .89 |
| maze-jw | 1.00 | .90 | .70 | .60 | .60 | .80 | .80 | .80 | .80 |
| puzzle | 1.00 | 1.67 | 1.33 | 1.67 | 1.67 | 1.67 | 1.33 | 1.67 | 1.33 |
| earley | 1.00 | 1.40 | .90 | 1.00 | .90 | 1.00 | .80 | .90 | 1.00 |
| splay | 1.00 | 1.29 | 1.00 | 1.14 | 1.14 | 1.14 | 1.00 | 1.00 | 1.14 |
| matrix | 1.00 | 1.14 | .86 | 1.14 | .86 | .86 | 1.00 | 1.00 | .86 |
| conform | 1.00 | 1.36 | .91 | 1.36 | .73 | .73 | .73 | .73 | .73 |
| matrix-jw | 1.00 | 1.17 | 1.17 | 1.17 | 1.17 | 1.17 | 1.17 | .83 | 1.17 |
| peval | 1.00 | 1.36 | 1.00 | 1.36 | 1.09 | 1.00 | 1.09 | 1.09 | 1.09 |
| nucleic-sorted | 1.00 | 1.00 | 1.03 | 1.07 | .77 | .77 | .77 | .77 | .73 |
| nucleic-star | 1.00 | 1.41 | 1.00 | 1.38 | .76 | .76 | .79 | .76 | .79 |
| fxtakr | 1.00 | 1.91 | 1.45 | 1.45 | 1.45 | 1.45 | 1.45 | 1.45 | 1.45 |
| em-imp | 1.00 | 1.25 | 1.00 | 1.12 | .88 | .88 | .81 | .75 | .75 |
| nucleic-jw | 1.00 | 1.09 | .95 | .95 | .95 | .95 | .91 | .91 | .82 |
| em-fun | 1.00 | 1.25 | 1.00 | 1.25 | .88 | .88 | .88 | .81 | .88 |
| lalr | 1.00 | 1.11 | 1.00 | 1.14 | .86 | .89 | .86 | .89 | .86 |
| takr | 1.00 | 1.29 | 1.06 | 1.06 | 1.06 | 1.06 | 1.06 | 1.06 | 1.06 |
| nbody | 1.00 | 1.06 | 1.00 | 1.06 | 1.12 | 1.12 | 1.12 | 1.00 | 1.06 |
| interpret | 1.00 | 1.17 | .96 | 1.12 | 1.08 | 1.00 | 1.08 | 1.08 | 1.00 |
| dynamic | 1.00 | 1.32 | 1.05 | 1.32 | 1.29 | 1.29 | 1.16 | 1.13 | 1.26 |
| texer | 1.00 | 1.24 | 1.02 | 1.16 | 1.07 | 1.07 | 1.02 | 1.02 | 1.09 |
| similix | 1.00 | 1.16 | 1.01 | 1.04 | .98 | 1.03 | .98 | .97 | .98 |
| ddd | 1.00 | 1.19 | .97 | 1.14 | .91 | .98 | .87 | .97 | .91 |
| softscheme | 1.00 | 1.30 | .99 | 1.27 | 1.01 | 1.02 | .96 | .96 | 1.01 |
| chezscheme | 1.00 | 1.17 | 1.01 | 1.16 | 1.10 | 1.10 | 1.09 | 1.08 | 1.10 |

**Table 4: Total compile times, normalized to the R⁵RS baseline. The coarse granularity of the timing mechanism gives us poor differentiation among many of the times, since compile times for most of the programs are very small.**