Speculative Inlining of Predefined Procedures in an R5RS Scheme to C Compiler

Marc Feeley

Dépt. d'informatique et de r.o., Université de Montréal, Canada

Abstract. The semantics of some dynamic programming languages, including Python, JavaScript, and R5RS Scheme, make it hard for a compiler to inline predefined procedures without compromising the semantics of the language. In the case of Scheme, many existing compilers can only achieve good execution speed by assuming that variables bound to predefined procedures are never mutated. This paper presents a *speculative inlining* approach which is portable and achieves good performance while fully conforming to the semantics of Scheme. It has been implemented in a mature Scheme to C compiler and we report on its performance on a large benchmark suite, both in execution speed and code size.

1 Introduction

Functional abstraction is useful for designing modular programs but the procedure call mechanism which implements the abstraction barrier has a run time cost for setting up parameters, directing the control flow to and from the procedure, and returning any results to the caller. The compiler can reduce the cost by *inlining* the procedure at the call site in the caller. This eliminates the need for the call/return control flow instructions, and it uncovers additional opportunities for optimization because the copy of the procedure body placed at the call site can be specialized for the actual parameters of that call.

We distinguish two kinds of inlinable procedures: *predefined procedures* provided by the language (e.g. `sqrt` and `map`), and *user procedures* whose definition must be given explicitly in the program. This classification covers language support operations, such as arithmetic, I/O, memory allocation and method dispatch, by treating them as inlinable predefined procedures. This paper addresses the problem of inlining predefined procedures in the R5RS Scheme language [11].

Some aspects of Scheme make inlining predefined procedures tricky. According to the semantics of Scheme the evaluation of `(+ x y)` decomposes into these steps: get the values of the variables `+`, `x`, and `y`, respectively t_1, t_2, and t_3, then check that t_1 is a procedure, and then call t_1 with the parameters t_2 and t_3. This usually adds `x` and `y` because the global variable `+` is initially bound to the addition procedure. During program execution it is possible to bind the variable `+` to a non-procedure value or to a different procedure, for example `(set! + list)`. After this assignment, the expression `(+ x y)` will in fact call the predefined procedure `list` and thus construct a two element list. This form of late

binding may be surprising, but it is sometimes useful as explained in Section 2. This problem is not specific to Scheme; indeed Python [14], JavaScript [2] and other scripting languages implement this form of late binding.

Predefined global variables may be mutated at run time in any part of the program, at the read-eval-print loop, in modules loaded dynamically using the predefined `load` procedure, and in S-expressions constructed and evaluated at run time by the predefined `eval` procedure. To achieve a better static analysis of programs, a Scheme compiler could adopt a static linking model by forbidding dynamic loading and `eval`, and not offering an interactive read-eval-print loop. This would allow a whole program analysis to determine conservatively that a given predefined variable is never mutated. Although this simplifies procedure inlining, it reduces the system's flexibility and it deviates from the Scheme semantics.

A popular alternative approach is the use of command-line flags and non-standard program annotations to force the compiler to assume the predefined variables contain their initial bindings. For example, the `(standard-bindings)` annotation of Gambit-C [3], the `--prim` flag of PLT Scheme's compiler [6], and the "benchmark mode" of Scheme48 [10]. This assumption is so common that it is the default compilation mode of CHICKEN [15], and the only compilation mode of Bigloo [13] which are both Scheme to C compilers.

Another aspect of Scheme which hinders the inlining of predefined procedures is the generic nature of fundamental operations such as `+` and `equal?`. Scheme supports a rich set of numerical types (infinite precision integers, rational, real, and complex) and the notion of exactness. Consequently most predefined procedures for performing arithmetic operations have non-trivial definitions which dispatch on the type and exactness of its arguments, type check the arguments, check for overflows, raise exceptions when appropriate, perform memory allocation, and so on. For space reasons it is unreasonable to fully inline arithmetic procedures. This is problematic because many programs need to do simple arithmetic on small exact integers for counting or indexing vectors.

To alleviate this problem many systems extend Scheme with a *fixnum* numerical type, which is a fixed-width exact integer type typically a few bits less than the natural word size of the machine, and procedures to operate on fixnums, such as `fx+` to add two fixnums, `fx<` to compare two fixnums, etc. Fixnum operations usually have a simple definition and they are fast because they do not require boxing and unboxing, and they have few special cases (overflow checking is typically the only special case). Some systems also provide a *flonum* numerical type, which is a fixed-precision inexact real type typically represented as a boxed machine floating point number. It is reasonable to inline fixnum and flonum operations because they take roughly the same space as a general procedure call.

The handling of fixnums and flonums varies considerably between systems, most notably in the width of fixnums and the handling of overflows (some systems detect fixnum overflows and signal an error, while others silently wraparound). For this reason it is difficult to write portable and fast programs even across systems that support fixnums and flonums. Moreover, fixnums cannot be used

in applications where the computations result in bignum integers that exceed the fixnum range even though most of the time the computations are within the fixnum range (financial calculations, number theoretic algorithms, etc).

A final concern in Scheme to C compilers, as considered here, is the high cost for implementing procedure calls. In order to correctly implement tail-calls the C code generated cannot in general directly translate Scheme calls into C calls. This is due to the fact that the global call graph of the program is partially known when a module is compiled (because of separate compilation, dynamic loading of modules, `eval` and mutation). Moreover, when compiling to C, some compilation techniques such as runtime code generation cannot be used.

To address these problems we have designed a speculative inlining algorithm which fully obeys the semantics of R5RS Scheme and does not rely on any program annotations, although it can take advantage of annotations to further improve performance. This algorithm has been integrated into the Gambit-C Scheme to C compiler. With no annotations, some programs approach the speed of hand tuned code with annotations and fixnum/flonum specific operations.

This paper describes the speculative inlining algorithm, how it integrates into the Gambit-C Scheme compiler and its performance. Section 2 explains situations where mutation of predefined global variables is useful. Section 3 discusses aspects of Gambit-C's compiler which interact with the speculative inlining algorithm which is described in detail in Section 4. Finally Section 5 reports on the performance both in execution speed and code space.

2 Mutation of Predefined Global Variables

The ability to mutate predefined global variables is consistent with Scheme's minimalistic philosophy. Using a single namespace for procedures and values (a "Lisp$_1$" [7]) is conceptually simpler than using two namespaces (a "Lisp$_2$"). Disallowing mutation of predefined global variables would increase the language's complexity by adding a special class of global variables. Moreover, mutation of predefined global variables is useful, as shown in the following examples.

Debugging – Scheme programs are often structured as a set of procedures defined at top-level. These procedures are bound to global variables and each call references the appropriate variable to get the procedure to call. Tracing calls to these procedures can be done by setting the variable to a new procedure which calls the old one and also displays the arguments and result of the call. This can be achieved by defining and using a `trace` macro as shown in Figure 1 (a).

Defining new types – There are no constructs in R5RS Scheme to define new types. Portable programs must represent new types using a predefined type, usually vectors. New types defined this way are not distinct because they cannot be distinguished from other new types and the vector used for their representation. A common solution is to use a unique tag at the head of the vector to identify the type unambiguously from other new types. Moreover, the `vector?` type predicate must be redefined to distinguish plain vectors from vectors representing the new types. Figure 1 (b) shows how a 2D point type can be defined.

```
> (define-syntax trace
    (syntax-rules ()
      ((trace var)
       (set! var (wrap 'var var)))))
> (define (wrap var proc)
    (lambda args
      (let ((r (apply proc args)))
        (write (cons var args))
        (display " ==> ")
        (write r)
        (newline)
        r)))
> (define (f n) (* (+ n 1) (+ n 2)))
> (trace f)
> (trace +)
> (f 10)
(+ 10 1) ==> 11
(+ 10 2) ==> 12
(f 10) ==> 132
132
```

(a) Debugging

```
(define old-vector? vector?)

(define (instance? obj tag)
  (and (old-vector? obj)
       (>= (vector-length obj) 1)
       (eq? (vector-ref obj 0) tag)))

(define pt (list 'pt)) ; unique tag
(define (make-pt x y) (vector pt x y))
(define (pt? obj) (instance? obj pt))
(define (pt-x p) (vector-ref p 1))
(define (pt-y p) (vector-ref p 2))

(set! vector?
      (lambda (obj)
        (and (old-vector? obj)
             (not (pt? obj)))))

(pt? (make-pt 11 22)) ⇒ #t
(vector? (make-pt 11 22)) ⇒ #f
```

(b) Defining new types

Fig. 1. Predefined global variable mutation examples

Overloading – Overloading of predefined procedures can be achieved easily with mutation. For example, `append` can be extended to allow concatenation of strings by setting the `append` variable to a procedure which either calls the `append` or `string-append` procedures depending on the type of the arguments.

Some scripting languages based on interpreters also support the redefinition of primitive functions. Figure 2 shows simple examples for the JavaScript and Python languages. In both cases the variable `abs`, whose initial binding is the function computing the absolute value, is modified to contain a different function.

```
Math.abs = Math.sqrt
alert(Math.abs(25)) // prints "5"
```

(a) JavaScript

```
abs = hex
print abs(25) # prints "0x19"
```

(b) Python

Fig. 2. JavaScript and Python examples

3 The Gambit-C System

3.1 System Architecture

The Gambit-C system [3] is a Scheme to C compiler based implementation of R5RS Scheme designed to be very portable while achieving good execution speed when programs are compiled.

A large part of the runtime system (roughly 50 kLOC), including the interpreter, debugger, bignum library, and all predefined procedures, is written in

Scheme. The compiler is also written in Scheme (roughly 25 kLOC). The rest of the system (roughly 40 kLOC), including the garbage collector, operating system interface, and foreign function interface is written in portable C.

Many macros which abstract away from the specifics of the platform are defined in the `gambit.h` header file: the use of the `gcc` C compiler and its extensions, the machine's natural word size and endianness, the width of each numeric type, the definition of virtual machine instructions, the representation of objects, etc. This header file plays a key role in the compilation process. The C files produced by the Gambit-C compiler are entirely composed of calls to macros defined in `gambit.h`. This allows a very late binding of the behavior of the generated code. Indeed, a C file produced by the Gambit-C compiler on a given machine does not have to be changed when it is compiled on a machine with a different C compiler, a different operating system, or different endianness and word size (for practical reasons the word size is currently limited to 32 and 64 bits). Porting to an unconventional C compiler typically only requires small changes to `gambit.h`.

Gambit-C supports separate compilation. In particular the runtime system's Scheme code is contained in 9 modules which are separately compiled. The modules of the runtime system and of the user can be statically linked to form an executable program. A running program can also load user modules dynamically and possibly more than once, to simplify debugging. Moreover, because the interpreter is written in Scheme [4], interpreted code and compiled code can freely call each other without compromising the Scheme semantics.

To implement tail-calls and continuations, Scheme calls cannot be translated directly into C calls. The runtime system manages a stack of Scheme continuation frames explicitly and independently from the C stack. The C code generated by the compiler is partitionned into a number of *host C procedures*. Depending on system build options, there is either a single host C procedure per Scheme module (*single host* mode) or one host C procedure per top-level Scheme procedure in the module (*multiple host* mode). Each host procedure contains a number of control points, which can either be procedure entry points or continuation return points. Trampolines are used to allow arbitrary jumps to a destination control point without C stack growth. Host procedures are only called from a dedicated *dispatcher* procedure. To jump to control point P, the current host returns to the dispatcher which then calls the host procedure containing P (i.e. the depth of the C call chain is never more than two). Upon entry to the host, a `switch` statement or a computed `goto`, if `gcc` is used, jumps to P in the host. There are a few optimizations to this basic approach which exploit locality, i.e. when P is in the same host. Regardless of the compilation mode and optimizations, calls to predefined procedures in the runtime are expensive because they necessarily require a non-local jump from the user program. The high cost of calling predefined procedures makes speculative inlining particularly attractive.

3.2 Extensions to Scheme

Gambit-C supports several extensions to R5RS Scheme. Some of the notable extensions are preemptive multithreading and a foreign function interface.

Low-Level Procedures Many low-level procedures meant primarily for the implementation of the runtime system are provided. These procedures typically perform a simple operation and are unsafe because they do not validate their arguments. For this reason, they are given an easily recognized name that is outside the standard Scheme identifier syntax (i.e. they cannot be found in an R5RS conformant program). These low-level procedures have names that begin with two hash signs. Here are a few examples:

- `##fx+` is the procedure which performs addition of fixnums. It does not check that the arguments are fixnums and whether there is a fixnum overflow.
- `##fx+?` is the procedure which performs addition of fixnums and checks for fixnum overflow. False (`#f`) is returned on overflow, otherwise the sum (a fixnum) is returned. It does not check that the arguments are fixnums.
- `##car` is a procedure which extracts a pair's `car` field. It does not check that the argument is a pair.
- `##cons` is the low-level procedure constructing pairs.

The duplication that occurs for `##cons` (which is identical to `cons`) and other low-level procedures is motivated by the need to easily distinguish internal low-level procedures from the procedures normally accessed by the user. This is convenient for the binding annotations explained in the next section.

User Annotations User annotations allow the programmer to force the compiler to assume certain properties about the code. This is useful when the programmer has knowledge that can help the compiler optimize the program. Annotations are specified inside the `declare` form. It is the programmer's responsibility to ensure that these annotations are correct; the compiler does not verify them. The `declare` form can appear anywhere a definition can appear. A `declare` at top-level has a lexical scope that extends until the end of the file. For a local `declare`, the lexical scope extends to the end of the enclosing binding form. Here is a typical use of user annotations:

```
(declare
  (standard-bindings) (extended-bindings) (block) (fixnum) (not safe))

(define z 0)

(define (iota n) (if (= n z) '() (##cons n (iota (- n 1)))))
```

The `(standard-bindings)` annotation asserts that a reference to a global variable predefined in R5RS will result in the corresponding predefined procedure. In other words in `iota` the calls to `=` and `-` will call the R5RS predefined procedures with those names. The `(extended-bindings)` annotation is similar but applies to Gambit-C extended procedures, such as `##fx+`, `##cons`, etc.

The `(block)` annotation asserts that all global variables defined in this file are only mutated in this file. Any global variable defined in a file and not mutated in that file can thus be treated like a constant. This enables constant propagation of global variables (e.g. replacing `z` in `iota` with 0), jump destination determination and inlining of user procedures defined at top-level (e.g. determining that the call to `iota` is a self-recursion).

The `(fixnum)` annotation asserts that all arguments and results of arithmetic procedures are fixnums. The `(not safe)` annotation tells the compiler that it is acceptable to generate unsafe code that could crash the program if some type checks fail. These two annotations in conjunction with the `(standard-bindings)` annotation allow the compiler to replace in `iota` the calls to `=` and `-` with unsafe calls to the fixnum specific procedures `##fx=` and `##fx-` respectively.

With carefully chosen annotations, programs can be made to run very fast, but at the price of safety. This is unacceptable in many situations. Moreover, annotations are brittle and are high maintenance. A small change in the program or in the dataset may invalidate the current set of annotations, but it is tedious and error-prone for the programmer to determine which ones.

In designing the speculative inlining algorithm, our goal was to improve the speed of execution without requiring that the programmer resort to annotations that are unsafe or otherwise change the semantics of the language (which includes all annotations described in this section).

3.3 Compiler Architecture

The compiler follows a conventional architecture. The source code is parsed and macros are expanded to produce an abstract-syntax tree (AST), which is transformed and annotated by subsequent passes. The AST is then traversed to generate the code for the Gambit Virtual Machine [5]. Low-level optimizations are then performed on this intermediate representation (dead code and common code elimination, instruction reordering, jump cascade removal, etc.) and finally it is expanded into C code in the form of calls to macros defined in `gambit.h`. The AST after all transformations is optionally pretty-printed as an S-expression.

The AST can represent expressions with no source code equivalent. Specifically, there is an AST node type representing procedure constants. These nodes are generated to refer to the procedure objects that exist at run time. Both predefined procedures and user procedures (but not closures) can be denoted. For example, the AST corresponding to this source code:

```
(let () (declare (standard-bindings)) (cons 11 22))
```

is transformed into an AST representing a call to a procedure constant denoting the predefined procedure `cons` (i.e. there is no longer a reference to the global variable `cons`). We use a box to denote procedure constants in the new AST:

```
(let () (declare (standard-bindings)) (cons 11 22)) → ('cons 11 22)
```

Note that $X \rightarrow Y$ will be used to mean "AST X is transformed into AST Y", where X and Y are external representations of ASTs possibly containing

procedure constants. The different passes which transform the AST are briefly explained below. They are executed in the order of presentation.

Assignment Conversion This pass introduces cells for local variables (including parameters) that are mutated. An assignment to a local variable is replaced with a mutation of the corresponding cell. This simplifies the implementation of closures and continuations which share mutated local variables. The remaining passes can assume that local variables are never mutated.

Beta Reduction This pass performs simple beta reductions of the code. The following transformations are done.

- **Constant and copy propagation**: When it is known that a variable V is never mutated and V is bound to X which is either a constant or a variable that is never mutated, references to V are replaced with X. For example:
  `(let ((x 5)) (let ((y x)) (+ y y)))` → `(+ 5 5)`
  `(let () (declare (standard-bindings)) +)` → `'+`
- **Constant folding**: When a constant predefined procedure is called, and all the arguments are constants of the correct type, and the procedure does not have side-effects, the call is replaced by a constant equal to the compile-time application of the procedure to the arguments. For example:
  `('+ 5 5)` → `10`
  There are subtle semantic issues which hinder constant folding. Calls to predefined procedures which allocate their result (e.g. `list` and `append`) are not constant folded because this would not preserve the uniqueness of the result (in the sense of `eq?`). Specifically, in Scheme:
  `(eq? (list 5) (list 5))` ≠ `(eq? '(5) '(5))`
  Because the target platform is not known at the time of the Scheme compilation, constant folding is tricky for procedures whose meaning is dependent on the target platform. This is specifically a problem for the fixnum operations because the width of a fixnum depends on the target machine's word size (30 bit fixnums on 32 bit machines, and 62 bit fixnums on 64 bit machines). An exact integer that does not fit in a 30 bit fixnum and that fits in a 62 bit fixnum is a bignum on 32 bit machines and a fixnum on 64 bit machines. So the `##fixnum?` procedure, which tests if its argument is a fixnum, is only constant folded when its argument is small enough to fit in a 30 bit fixnum or larger than fits in a 62 bit fixnum:
  `('##fixnum? 123)` → `#t`
  `('##fixnum? 1000000000000)` *is not constant folded*
  `('##fixnum? 2305843009213693952)` → `#f`
  Constant folding is also performed on conditional expressions, that is the `if`, `and`, and `or` special forms. For example:
  `(if (and #f (f 2)) 123 (g 3))` → `(g 3)`

– **Inlining of user procedures**: When it is known that a given variable V is never mutated and V is bound to a lambda-expression, calls to V are replaced by calls to a copy of the lambda-expression. As an additional condition, the size of the new call (measured in number of nodes in the AST) must not be larger that a certain factor F of the size of the original call in the source code. By default F is 3, but the programmer can modify this with the `(inlining-limit` F`)` user annotation. For example:

  `(let ((f (lambda (x) (+ x x)))) (f 5))` → `(+ 5 5)`

To improve the effectiveness of these beta reductions, processing generally starts at the leaves of the AST and progresses towards the root. For binding forms, the bound values are processed before the body. Finally, the top-level procedures are processed in the reverse order of their dependencies. If procedure `f` contains a call to procedure `g`, which contains a call to procedure `h`, then `h` is processed first, then `g`, and then `f`.

Lambda Lifting This pass transforms local user procedures using the lambda lifting transformation [9]. This eliminates the creation of closures for lambda-expressions bound to local variables when these local variables are only referenced in the operator position of calls. The lambda-expressions are modified so that they take their free variables as explicit parameters. All calls to these variables are also modified to pass the value of the free variables.

4 Speculative Inlining

Speculative inlining of predefined procedures is performed as an AST transformation pass just before assignment conversion.

4.1 Basic Approach

Our approach capitalizes on the high likelihood that predefined global variables contain the corresponding predefined procedure. When a predefined procedure is speculatively inlined, the inlined code must be guarded by a *run time binding test* to verify that the variable is indeed bound to the expected predefined procedure.

If the binding test fails, the inlined code is not appropriate and a normal procedure call using the global variable must be performed. For correct handling of tail-calls, this call must be a tail-call with respect to the original call.

If the binding test succeeds, the inlined code is executed. In the ideal case this code will perform the work required of the predefined procedure and return the required result. It is possible however that the inlined code encounters an exceptional case, such as an argument of the wrong type, or a complex case that would be too space inefficient to handle inline (such as a fixnum arithmetic operation overflowing into the bignum range). We will call these conditions the *inlining conditions* of the procedure. When the inlining conditions do not hold the execution can fall back to a normal procedure call. We require that the

inlined code only perform side-effects after verifying the inlining conditions. As an example, here is the speculative inlining of `car`:

```
(f (car (g 5))) → (f (let ((x (g 5)))
                       (if (and (' ##eq? car ' car )
                                (' ##pair? x))
                           (' ##car x)
                           (car x))))
```

Falling back to a normal procedure call is not only correct, it ensures that the behavior of a call to a predefined procedure is the same, except for execution speed, whether the procedure is inlined or not: the same exceptions are raised, the same continuation is used, etc. The inlining is purely a compiler optimization that is transparent to the programmer.

4.2 Inlining Scheme's Numeric Procedures

The inlining of Scheme's numeric procedures is problematic because most numeric operations are generic, they can accept several numeric types, and can accept mixed types. In Gambit-C, there are five representations for numbers: exact integers are represented with fixnums and bignums, exact rationals are represented as pairs of exact integers, inexact reals are represented as flonums (64 bit IEEE 754 floating point number), and complex numbers are represented as pairs of reals. Except for fixnums, these representations are memory allocated.

If we take addition as an example, the algorithm for adding two numbers depends on the representation of the numbers to add. It is necessary to dispatch on the type of both arguments to determine how to proceed. In the Gambit-C runtime all 25 cases are laid out to avoid needless representation conversions.

It is unreasonable to inline this much code routinely. Instead, the most likely case must be handled inline, with the less likely cases handled by the fall back. But what constitutes a likely case depends on the nature of the computation. There is a large class of algorithms which process small exact integers (e.g. counting and indexing vectors). On the other hand, scientific applications usually perform the bulk of their computations with inexact reals. The other numeric types (exact rationals and complex numbers) are less useful to handle inline in Gambit-C because the algorithms operating on them are complex and often require procedure calls (e.g. computing the GCD for normalizing rational results).

Consequently, there are two cases that are interesting to handle inline: when all the arguments are fixnums, and when all the arguments are flonums. A set of 5 user annotations is provided to allow the programmer to specify which case is most likely, and which cases to inline:

- `(mostly-fixnum)`: The fixnum case is more likely and is inlined.
- `(mostly-flonum)`: The flonum case is more likely and is inlined.
- `(mostly-fixnum-flonum)`: The fixnum case is more likely than the flonum case, but both are likely and are inlined. The fixnum case is checked first.
- `(mostly-flonum-fixnum)`: The flonum case is more likely than the fixnum case, but both are likely and are inlined. The flonum case is checked first.

– `(mostly-generic)`: The numeric procedures are not inlined.

These annotations are purely advisory. They affect the performance of the code but do not compromise the Scheme semantics. The following example shows the speculative inlining of `+` when the user annotation `(mostly-fixnum-flonum)` is in effect:

```
(let () (declare (mostly-fixnum-flonum)) (f (+ (g 2) (h 3))))
→
(f (let ((x (g 2)) (y (h 3)))
     (if (' ##eq? + ' +)
         (if (and (' ##fixnum? y) (' ##fixnum? x))
             (or (' ##fx+? x y) (+ x y))
             (if (and (' ##flonum? y) (' ##flonum? x))
                 (' ##fl+ x y)
                 (+ x y)))
         (+ x y))))
```

In the resulting AST, the `##fx+?` predefined procedure is used to perform the fixnum addition and overflow check. If the global variable `+` does not have its standard binding, or a fixnum overflow is detected, or the arguments are not both fixnum or both flonums, then a normal call to `+` is performed. The common code elimination optimization of the compiler will generate compact code by merging all three calls to `+`.

4.3 Inlining Recursive Procedures

The following recursive predefined procedures on lists are speculatively inlined: `assq`, `memq`, `map`, and `for-each`. Both `assq` and `memq` are worth inlining because many Scheme programs rely on them, their definitions are short, and they do not need to call procedures that are not easily inlined (`assv`, `assoc`, `memv`, and `member` are not inlined because they call `eqv?` and `equal?` whose definitions are considerably more complex than `eq?` which is called by `assq` and `memq`).

To avoid too much code expansion, the higher-order procedures `map` and `for-each` are only inlined when they are passed two arguments: the procedure argument and list. They are worth inlining not only because many Scheme programs rely on them but because the inlined code exposes optimization opportunities at the call to the procedure argument which can often avoid an expensive general call. If the procedure argument is a user procedure in the same file then a direct jump to the procedure can be performed (and without a parameter count if it does not take a rest parameter). The procedure argument is also a candidate for inlining, whether it is a user procedure or predefined procedure.

4.4 Interaction with Beta Reduction Pass

Implementing speculative inlining as an AST transformation has the advantage that subsequent transformations can further optimize the inlined code. In particular, the beta reduction pass may simplify the inlined code through constant

propagation and constant folding. Consider a slight variation on the previous example, where the second argument to `+` is the constant 1. The speculative inlining of `+`, followed by constant propagation will give:

```
(let () (declare (mostly-fixnum-flonum)) (f (+ (g 2) 1)))
→
(f (let ((x (g 2)))
     (if ('##eq? + '+)
         (if (and ('##fixnum? 1) ('##fixnum? x))
             (or ('##fx+? x 1) (+ x 1))
             (if (and ('##flonum? 1) ('##flonum? x))
                 ('##fl+ x 1)
                 (+ x 1)))
         (+ x 1))))
```

The calls `('##fixnum? 1)` and `('##flonum? 1)` will then be constant folded to `#t` and `#f` respectively, allowing both `and`s and the `if` guarding the flonum case to be constant folded:

```
(f (let ((x (g 2)))
     (if ('##eq? + '+)
         (if ('##fixnum? x)
             (or ('##fx+? x 1) (+ x 1))
             (+ x 1))
         (+ x 1))))
```

The constant propagation transformation can also make use of user annotations to further improve the code. If we add the annotation `(standard-bindings)` to the previous example, the code at the end of the AST transformations will be:

```
(f (let ((x (g 2)))
     (if ('##fixnum? x)
         (or ('##fx+? x 1) (+ x 1))
         (+ x 1))))
```

5 Evaluation

5.1 Experimental Setup

To evaluate the effectiveness of the speculative inlining approach, we compiled several Scheme benchmarks using the Gambit-C compiler with various user annotations. We are interested in measuring the impact of our approach on the execution speed and also on the code size. We used Gambit-C version 4.2.3 built with the configure options `--enable-single-host --enable-char-size=1` and, for comparison, Bigloo version 3.0d built with the configure option `--benchmark=yes`. All tests were performed on a Linux workstation (2 GHz Dual Core AMD Opteron with 16 GB SDRAM) and `gcc` 4.0.2 was used.

Version 4.2.3 of Gambit-C supports the speculative inlining of 101 R5RS predefined procedures, and 128 Gambit-C specific predefined procedures. The speculative inlining is performed on all the type predicates (e.g. `pair?`, `null?`),

most of the R5RS numeric procedures (e.g. `+`, `-`, `*`, `/`, `=`, `<`, `quotient`, `max`, `sqrt`), all the case-sensitive character comparison procedures (e.g. `char=?`), pair, string and vector constructor, accessor and mutator procedures (e.g. `cons`, `cadr`, `string-length`, `vector-set!`), miscellaneous procedures (e.g. `not`, `values`, `eq?`), and the recursive procedures `assq`, `memq`, `map`, and `for-each`.

The benchmarks contain programs representative of typical Scheme applications. There are 41 benchmarks in all. The largest are: `scheme` (Scheme interpreter in Scheme, 1 kLOC), `slatex` (Scheme to LaTeX formatter, 2.3 kLOC), `nucleic` (scientific application [8], 3.5 kLOC) and `compiler` (Scheme compiler, 11.7 kLOC).

To determine the size of the code generated by the Gambit-C compiler, we measured the size of the machine code produced by the C compiler and substracted the code size for an empty program. We did not measure the size of the program's data because it could not easily be isolated from the data of the Gambit-C runtime. The data size should not vary much between settings.

The benchmarks were also run with Bigloo to compare the execution time with a high-performance Scheme compiler. We used the Bigloo compiler options `-O6 -copt -O3 -copt -fomit-frame-pointer` and no explicit type information was used in the source code. This mode gives a semantics that is close to R5RS, but does not fully conform to it because it assumes that none of the predefined global variables are mutated and it does not check for arithmetic overflow.

For both Gambit-C and Bigloo, we set the same initial heap size when executing the compiled program (10 MB), and strings were represented using one byte per character.

Given our goal of achieving the best execution speed without compromising the Scheme semantics, one set of trials with Gambit-C avoided the annotation `(standard-bindings)`, but we tried each of the numeric user annotations: `(mostly-fixnum)`, `(mostly-flonum)`, etc. The `(mostly-fixnum-flonum)` case is used as the baseline because it corresponds to the default when the programmer does not supply any user annotations. Another set of trials was done with those user annotations combined with `(standard-bindings)`, in violation of the R5RS Scheme semantics. This is useful to evaluate the cost of the run time binding test.

We also tried the benchmarks with the set of user annotations that achieve the best speed which we call unsafe mode. That is the `(not safe)`, `(block)`, and `(standard-bindings)` user annotations were used, in addition to benchmark specific annotations for arithmetic (either `(fixnum)` or `(flonum)` as appropriate for the benchmark).

In addition, a trial was done with the speculative inlining transformation disabled. This situation approximates the Gambit-C compiler before the speculative inlining transformation was added. This trial and those using speculative inlining are the only ones which do not violate the Scheme semantics.

In all trials using Gambit-C, the inlining of user procedures was disabled. As a result the programs run slower than they would normally. This is particularly

noticeable for the unsafe mode where loop unrolling and data structure accessor inlining can boost performance relative to the baseline by 13% on average. This was done to avoid side effects of the user procedure inliner which would add noise to the code size measurements. Bigloo performs automatic inlining of primitives and user procedures.

5.2 Experimental Results

The results are given in Table 1. For each combination of benchmark and compilation mode, the execution time and code size are given and the code size is underlined. Lower values are better. To simplify comparison, all measurements are relative to the baseline (i.e. speculative inlining with `(mostly-fixnum-flonum)` but without `(standard-bindings)`). A value of 1 means the same time or space as the baseline. For lack of space in the table the columns for the baseline are omitted since they contain 1 everywhere for time and space. Moreover we omit the columns for `(mostly-flonum-fixnum)` because the time and space were within a few percent of the columns for `(mostly-fixnum-flonum)`. The table is ordered by increasing speedup of the baseline mode compared to the speculative inlining disabled mode.

By examining the speculative inlining disabled column we see that the benchmarks always execute faster with speculative inlining than without. The geometric mean speedup is 6.14, but in several cases the speedup is more than 10, and over 20 for `sum`. The code size is on average 79% larger when speculative inlining is used, and up to 4.5 times the size for `fft`. Overall we view these results positively since among the compilation modes which do not violate the R5RS semantics, speculative inlining is consistently faster while the code size is typically not unreasonably large.

If we now compare the `(mostly-fixnum)` and `(mostly-flonum)` modes with speculative inlining we see that the execution time for the `(mostly-fixnum)` case is better in general but worse on benchmarks which are floating point intensive (`nucleic`, `ray`, `fibfp`, `mbrot`, `sumfp` and `pnpoly`). The ratio can be up to 10 times in favor of `(mostly-flonum)` for `sumfp` and up to 21 times in favor of `(mostly-fixnum)` for `sum` (the same computation as `sumfp` but performed using small integers). In terms of code size the `(mostly-flonum)` case is normally better, by about 5% on average. This is probably due to the absence of an overflow check when operating on flonums.

Interestingly, `fft`, which uses a mix of operations on fixnums and flonums, is about the same speed with `(mostly-fixnum)` and `(mostly-flonum)`. This is explained by the fact that there is an equal number of fixnum and flonum operations, so the same number of non-local jumps result whether the fixnum case or the flonum case is inlined. The execution speed improves by a factor of over 5 when `(mostly-fixnum-flonum)` or `(mostly-flonum-fixnum)` are used. Considering all benchmarks these compilation modes give the best execution speed; 1.34 times faster than with `(mostly-fixnum)` on average and 2.19 times faster than with `(mostly-flonum)` on average. The `(mostly-fixnum-flonum)` mode gives marginally better performance which is why Gambit-C uses it as

the default compilation mode. The code size is consistently bigger than with `(mostly-fixnum)` and `(mostly-flonum)`, but by only 16-22% on average.

The cost of the run time binding tests can be evaluated by looking at the column for the `(standard-bindings)` + `(mostly-fixnum-flonum)` compilation mode. This mode generally yields code that is faster than the baseline, by about 13% on average. This mode also generally yields more compact code than the baseline, 38% smaller on average. Our view is that this is an acceptable cost for the run time binding tests which are required for conformance to the Scheme semantics.

The unsafe mode column indicates that with hand tuned user annotations and unsafe code, programs can run considerably faster, by a factor of about 2 on average but in some cases 4 times faster than the baseline (and even 8 times faster when user procedure inlining is enabled). Moreover the code is over 5 times more compact because there remains very few procedure calls in the code (and consequently fewer return points, continuation frame allocations and setup, stack overflow checks, etc. which all contribute to the total code). This shows in our view that speculative inlining does not completely eliminate the need for unsafe user annotations when very high performance and compact code are required. However, speculative inlining does contribute to lessen the urgency to resort to user annotations and promote a more maintainable coding style.

Finally, we can see that the performance of Gambit-C with speculative inlining is reasonably good in absolute terms. It is about 6% slower than Bigloo on average and about 21% slower when we ignore the `call/cc` intensive benchmarks `ctak` and `fibc`.

6 Related Work and Conclusion

Inlining has been used in other dynamically-typed programming languages to improve performance. Most notable is the compiler for Self [1], an object-oriented dynamically-typed programming language, which uses *message inlining* to speed up message sends by reducing the frequency of method lookups. On the first execution of the messsage send, a normal method lookup is performed to find the correct method to call based on the type of the receiving object. The message send is then *backpatched* to jump directly to this method and the type of the receiving object is saved. On subsequent message sends the method will be called if the type of the new receiving object is the same, otherwise the system reverts to a new method lookup and backpatch. *Selective recompilation* of the program is used when method definitions are changed. All of this requires a complex system architecture, the presence of the compiler in the runtime system, and runtime code generation. More recently the Java HotSpot VM [12] has used a similar inlining-with-recompilation approach and a complex runtime architecture.

Our approach is in comparison much simpler and can be applied in situations where runtime code generation is not an option such as in compilers which generate C code, in memory constrained systems, and embedded systems where the code must be stored in read-only memory. With an extensive experimental

evaluation using a mature Scheme system, we have shown that our approach can be used to correctly implement the R5RS semantics while achieving execution speeds comparable to other Scheme compilers which attain high-performance by violating the R5RS semantics.

Acknowledgements

This work was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada.

References

1. Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford, 1992.
2. ECMA. Ecma-262: Ecmascript language specification, 1999.
3. Marc Feeley. Gambit-C version 4.2.3. `http://gambit.iro.umontreal.ca`, 2008.
4. Marc Feeley and Guy Lapalme. Using closures for code generation. *Computer Languages*, 12(1):47–66, 1987.
5. Marc Feeley and James S. Miller. A Parallel Virtual Machine for Efficient Scheme Compilation. In *Proc. of the ACM Symposium on LISP and Functional Programming*, pages 119–130, 1990.
6. Matthew Flatt. PLT MzScheme: Language manual. Technical Report PLT-TR05-1-v300, 2005. `http://www.plt-scheme.org/techreports/`.
7. Richard P. Gabriel and Kent M. Pitman. Technical issues of separation in function cells and value cells. *Lisp and Symbolic Computation*, 1(1):81–101, June 1988.
8. P. H. Hartel, M. Feeley, M. Alt, L. Augustsson, P. Baumann, M. Beemster, E. Chailloux, C. H. Flood, W. Grieskamp, J. H. G. Van Groningen, K. Hammond, B. Hausman, M. Y. Ivory, R. E. Jones, J. Kamperman, P. Lee, X. Leroy, R. D. Lins, S. Loosemore, N. Röjemo, M. Serrano, J.-P. Talpin, J. Thackray, S. Thomas, P. Walters, P. Weis, and P. Wentworth. Benchmarking implementations of functional languages with "Pseudoknot", a float-intensive benchmark. *Journal of Functional Programming*, 6(4):621–655, 1996.
9. Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proc. of the conference on Functional Programming and Computer Architecture*, pages 190–203, New York, NY, USA, 1985.
10. R. Kelsey and J. Rees. The Incomplete Scheme 48 Reference Manual, 1999.
11. Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.
12. Michael Paleczny, Christopher A. Vick, and Cliff Click. The Java HotSpot Server Compiler. In *Java Virtual Machine Research and Technology Symposium*. USENIX, 2001.
13. M. Serrano and P. Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In *Proc. of the Static Analysis Symposium*, pages 366–381, 1995.
14. Guido Van Rossum. *The Python Language Reference Manual*. Network Theory Ltd., September 2003.
15. Felix L. Winkelmann. *CHICKEN - A practical and portable Scheme system*, 2008.

Table 1. Relative execution time and relative code size (underlined) for various compilation modes; baseline is speculative inlining and `(mostly-fixnum-flonum)`

	R5RS semantics obeyed						**R5RS semantics violated**							
	Speculative inlining						`(standard-bindings)`						Unsafe	Big-
Program	disabled		+ fix		+ flo		+ fix		+ flo		+ fix-flo		mode	loo
`dynamic`	1.54	.62	1.00	1.00	1.05	1.00	.96	.63	1.02	.63	.96	.63	.81 .29	.25
`slatex`	1.54	.61	.93	.96	.88	.90	.80	.60	.83	.56	1.01	.61	.74 .20	.45
`ctak`	1.75	.91	1.00	.97	1.55	.98	.90	.79	1.45	.75	.89	.81	.83 .48	55.03
`fibc`	2.23	.84	.99	.96	2.26	.95	.96	.80	2.24	.80	.96	.85	.71 .35	13.78
`conform`	2.78	.73	.98	1.00	.96	1.02	.87	.64	.89	.70	.87	.64	.60 .34	.21
`compiler`	2.99	.69	.99	.94	1.47	.90	.87	.67	1.34	.65	.87	.70	.52 .24	.29
`scheme`	3.77	.74	1.00	1.00	1.25	.97	.95	.71	1.16	.68	.92	.71	.69 .27	.43
`nucleic`	3.84	.54	2.48	.81	1.00	.82	2.34	.61	.87	.63	.90	.73	.34 .17	.87
`ray`	3.96	.47	3.32	.70	.99	.74	3.08	.49	.94	.58	.95	.71	.44 .21	1.24
`maze`	4.55	.55	1.02	.84	2.57	.70	.92	.53	2.49	.49	.92	.59	.34 .16	.40
`paraffins`	4.97	.38	1.00	.73	1.04	.72	.96	.44	.99	.42	.95	.53	1.00 .12	1.06
`deriv`	5.14	.54	1.00	1.00	1.16	1.04	.80	.56	.93	.61	.80	.56	.67 .26	1.23
`perm9`	5.34	.57	.99	.92	3.85	.83	.95	.53	3.73	.47	.93	.60	.85 .16	.80
`matrix`	5.36	.58	1.01	.90	2.39	.86	.86	.54	2.18	.53	.86	.62	.71 .28	.72
`dderiv`	5.48	.58	.98	1.00	1.13	1.03	.81	.58	.95	.60	.82	.58	.61 .28	.82
`fibfp`	5.70	.57	5.69	.61	1.02	.98	5.53	.49	1.00	.79	.93	.84	.64 .17	2.78
`lattice`	5.92	.69	.98	.99	.99	1.00	.83	.64	.84	.64	.82	.65	.71 .35	.32
`graphs`	6.17	.57	1.03	.91	2.31	.86	.91	.63	2.22	.61	.94	.66	.67 .23	.94
`earley`	6.20	.47	1.03	.90	2.89	.78	.89	.53	2.77	.43	.96	.55	.78 .17	.72
`peval`	6.43	.57	1.00	.96	1.03	.94	.76	.50	.81	.50	.76	.51	.49 .23	.31
`divrec`	6.44	.72	1.00	1.00	1.06	.89	.87	.64	.91	.55	.87	.64	.52 .16	.85
`simplex`	6.78	.36	1.99	.71	3.52	.55	1.92	.45	3.40	.35	.91	.52	.44 .11	.61
`primes`	6.94	.66	1.01	.99	3.18	.91	.89	.66	3.10	.55	.89	.68	.75 .16	1.11
`cpstak`	6.95	.81	1.01	.96	5.29	.92	.92	.74	5.18	.72	.93	.77	.84 .36	2.71
`browse`	7.16	.61	.97	.99	1.01	.98	.82	.50	.86	.46	.83	.51	.68 .19	.37
`fib`	7.37	.56	.93	.91	7.29	.69	.86	.65	7.03	.52	.93	.76	.38 .13	.62
`tak`	7.51	.67	1.00	.96	5.68	.86	.88	.65	5.70	.59	.89	.69	.34 .18	.50
`mazefun`	8.25	.66	.97	.98	5.70	.92	.89	.63	5.40	.59	.92	.66	.51 .24	.77
`nboyer`	8.30	.57	.98	.98	1.26	.94	.72	.42	.98	.41	.72	.42	.64 .21	.48
`mbrot`	9.18	.31	7.82	.55	2.37	.49	7.63	.39	2.28	.33	.95	.55	.57 .12	3.15
`takl`	9.30	.77	1.00	1.00	1.01	.94	.65	.66	.60	.59	.65	.66	.23 .17	.13
`triangl`	9.68	.48	.97	.88	6.69	.68	.93	.60	6.50	.42	.93	.60	.32 .14	.53
`sumfp`	9.70	.60	9.81	.55	1.03	.90	9.21	.45	.97	.78	.87	.89	.80 .10	4.75
`diviter`	9.75	.70	1.00	1.00	1.06	.89	.74	.62	.78	.53	.75	.62	.68 .16	1.09
`sboyer`	10.46	.57	.98	.98	1.30	.95	.67	.42	.96	.41	.66	.42	.57 .23	.35
`destruc`	11.16	.47	1.06	.85	6.84	.69	.86	.41	6.62	.34	.87	.41	.67 .12	1.05
`fft`	12.32	.22	5.14	.65	5.11	.46	5.04	.47	4.83	.33	.89	.61	.28 .05	2.03
`pnpoly`	12.85	.39	6.84	.70	3.80	.55	6.75	.50	3.56	.37	.89	.61	.23 .08	4.78
`puzzle`	13.29	.40	.98	.73	8.69	.65	.78	.52	8.25	.47	.83	.57	.35 .10	1.71
`nqueens`	14.12	.40	.96	.64	7.01	.65	.76	.33	6.54	.34	.76	.40	.52 .08	.83
`sum`	20.63	.64	.96	.84	21.05	.70	.75	.63	20.25	.51	.71	.79	.29 .07	1.68
geom. mean	6.14	.56	1.34	.86	2.19	.82	1.17	.56	1.99	.53	.87	.62	.54 .18	.94