SINTL: A Strongly-Typed Generic Intermediate Language for Scheme

Mark DePristo

15 May 2000

Abstract

This paper describes SINTL, a strongly-typed generic intermediate language for Scheme. The paper begins by outlining the motivation for developing a sophisticated intermediate language for a new Scheme compiler. We consider two novel aspects of SINTL as an intermediate language for Scheme, a declarative type system and type inference algorithm, and a register to stack machine conversion algorithm. We then demonstrate the effectiveness of these two techniques by discussing the JVM-SINTL backend, a fully functional backend to the Java Virtual Machine. Following a discussion of the representational choices and compilation techniques used in JVM-SINTL, we compare the performance of the JVM-SINTL backend relative to several popular Scheme compilers. Ultimately, this thesis demonstrates that a sophisticated compiler with well-selected analyses and optimizations can generate high-quality code running on the JVM, and acheive performance an order of magnitude better than current Scheme compilers to the JVM.

Contents

1 Introduction and Motivation **3**
- 1.1 Using the Features of The Target Machine . . . 3
- 1.2 Support Many Diverse Targets Machines . . . 4
- 1.3 A Completely Generic Runtime System . . . 4

2 The Design of SINTL **5**
- 2.1 The Instruction Set . . . 5
- 2.2 The Declarative Type System . . . 9
- 2.3 The Assembler Phases . . . 12

3 Type Inference in SINTL **13**
- 3.1 Instruction Types . . . 14
- 3.2 Block Level Type Inference . . . 15
- 3.3 Instruction Level Type Inference . . . 16

4 The Reg→Stack Instruction Selection Algorithm **18**
- 4.1 DAG Value Dependency Analysis . . . 19
- 4.2 Local Variable Assignment . . . 20
- 4.3 Stack Machine Instruction Selection . . . 20

5 The JVM Backend **22**
- 5.1 Why the JVM . . . 22

6 An Extended Compilation Example **26**
- 6.1 Front End Compilation . . . 26
- 6.2 Initial SINTL Phases . . . 27
- 6.3 Type Inference . . . 28
- 6.4 Reg→Stack Instruction Selection . . . 29
- 6.5 JVM Source Code . . . 29

7 Performance Results for JVM-SINTL **33**

8 Conclusion **41**

1 Introduction and Motivation

SINTL (Scheme INTermediate Language) is a strongly-typed, linear instruction stream register transfer language and assembler backend for the PM compiler [1]. It was designed to use a minimum instruction set of only twenty-two instructions with well-defined, simple semantics. The instruction set is not a machine language, however, remaining intermediate between Scheme and machine assembly code. It is similar in flavor to the Scheme48 Virtual Machine instruction set [13] and MacScheme machine code.

Our principal goals for the PM compiler expanded as the scope of the project grew. After several prototype compilers were implemented and their difficulties identified, we settled on the following major goals:

1.1 Using the Features of The Target Machine

Our principle goal for the PM compiler is to use the features of a target machine to directly implement the features of Scheme. Although many systems backend to a target machine not specifically designed for the source language (the JVM for instance), most compiler writers choose to drastically simulate their runtime system on the target machine. For instance, implementing call-with-current-continuation in Scheme is notoriously difficult because with call/cc the lifetime of a stack frame does not always end when the frame returns to its caller. Languages with call/cc choose one of several standard implementation options. One option, taken by Scheme48 [13], MzScheme [5], and Standard ML of New Jersey [11], is to heap allocate all stack frame and hence 'garbage collect' the stack. This is an attractive option when one is able to design and implement both the compiler and virtual machine, because it gives reasonable performance and makes the implementation of call/cc extremely easy and efficient. Another option, discussed by Dybvig in [4], is to use a traditional stack for call frames, and implement call/cc by copying the current stack into the heap. This approach gives excellent performance for most Scheme programs, but heavily penalizes the use of call/cc. Unfortunately, neither of these approaches are viable for systems on top of the JVM. Since the JVM and similar systems were designed for languages without call/cc, they have a strict call stack discipline and do not provide functionality for copying the call stack into the heap. Our approach to this problem is similar to that taken in Kawa [3] in only supporting upward, single return continuations. Such continuations are semantically like throw/catch exceptions in C++ and Java, and hence well supported systems like the JVM.

By choosing to only support upward continuations and optionally full tail-calls, we can implement almost all of Scheme's features immediately on the machinery provided by

[1]The PM compiler is a Scheme front end compiler under development in the Northwestern Unversity Autonomous Mobile Robotics Group. PM is a temporary name given to the compiler, formed from the names of its principal designers Pinku Surana and Mark DePristo.

the JVM. This means that we use the target machine's argument passing, return value, and local variable mechanisms. In some cases the underlying machine simply cannot support the semantics of Scheme (such as lexical scoping); in such cases, we chose to implement the feature in the least intrusive way possible.

This might seem like an odd goal, but it has at least two substantial benefits. First, systems like the JVM are often just-in-time compiled from the portable virtual machine instruction set to native instructions. Such compilers are reasonably effective, but by their very nature must be fast and hence cannot do sophisticated program analysis. By compiling Scheme to code using the conventions of the target machine, it is much more likely that a JIT will effectively translate our code in native machine code. Second, it eases inter-operability between the source language and other languages on the target machine. Since our compiler generates code that follows the conventions of the target machine, it is easier for systems in our language to access features of the target machine and for programs in other languages to call Scheme programs.

1.2 Support Many Diverse Targets Machines

Our experience with our initial prototype compilers was that features of the intended target inexorably crept into all phases of our compiler. Once assumptions about the target machine were embedded in the compiler, changing any part of the system invariably lead to violated assumptions. Given that we were unsure how to efficiently implement Scheme on the JVM, the ability to experiment with representations, analyses, and optimizations was and is still critical.

1.3 A Completely Generic Runtime System

Writing a large runtime system for many target machines is extremely tedious and error-prone. There are simply not enough hands available to write three or four efficient runtime systems, even for an minimalist language like Scheme. We thus chose to write a single completely generic runtime system in Scheme using a minimal set of primitive operations. We thus pushed the work of implementing the RTS efficiently on the target system onto the compiler, and concentrated our efforts on improving the quality of the compiler. Additionally, once the RTS proved itself on a single backend, it become substantially easier to port the compiler to other platforms. This is especially important when retargeting the compiler to targets under development since helps to isolates failures in the relatively small instruction selector or backend specific runtime system.

2 The Design of SINTL

It became clear that building a compiler with genericity and portability required a strong division of labor between the front end Scheme compiler and the backend code generator. SINTL was designed to encapsulate all the information needed to compile a sequence of instructions into a target language. The SINTL instructions comprise an abstract register transfer language with only twenty-two instructions. The instruction stream is strongly-typed in a simple, Scheme specific type system described in section 2.2.

Deciding to write the entire runtime system in Scheme posed a host of difficult design questions for the SINTL system. Since much of the time of an average Scheme program is spent in the runtime system, it is essential that we effectively compile the runtime system. By writing the RTS as clean Scheme code, we moved the work of optimizing the RTS from the RTS programmer to the compiler writer. We anticipated that work improving the compiler would be more profitable in the long run, since the general compiler optimizations would be applicable to all Scheme code and would be beneficial to us (the writers of the RTS) and to our users (Scheme programmers). Further, by writing a generic RTS which is then compiled onto the target language, we ease the burden on the person porting the compiler to a new backend. Instead of writing a large and difficult to debug backend and RTS, retargeting SINTL to a new backend involves describing the way to access low-level primitives and writing a simple instruction selector from SINTL code to the target language.

2.1 The Instruction Set

The instruction set provided by SINTL is shown in table 2.1.

The argument specification contains the descriptors:

reg A register containing a value.

name A symbolic name. For a label or conditional instruction, the name must correspond to the label within the template. For a primitive instruction, the name must be the name of a primitive known to SINTL for the particular backend in use.

lit A literal object of any type permissible in R5RS Scheme.

type A SINTL type in the currently selected backend.

global A symbolic name for a global variable.

The instructions are of the form (result-register arguments ...). There are several instructions warranting elaboration.

Instruction	Operands	Description
literal	(reg lit)	Load a literal
global-ref	(reg global)	Load global variable
global-set!	(global val)	Set global variable
global-define!	(global val)	Define global variable
make-closure	(reg env template)	Create a closure with env and template
this-closure	(reg)	Load current closure
closure-env	(reg closure)	Load closure environment
make-env	(reg parent length)	Make an env with parent env of length
env-parent	(reg env)	Load an env's parent
env-ref	(reg env index)	Load env[index]
env-set!	(env index val)	env[index] $\leftarrow$ val
set!	(dst src)	Copy src into dst
call	(reg proc arg ...)	Call proc with arguments
tail-call	(reg proc arg ...)	Tail-call proc with arguments
return	(reg)	Return result
label	(label)	Label instruction stream
branch	(label)	Branch to label
branch-false?	(test label)	Conditional branch on test to label
false?	(reg val)	Is value Scheme false?
type-cast	(reg val type)	Cast value to type
ensure-type	(reg type)	Ensure value is of type
primitive	(reg name arg ...)	Invoke a primitive routine

Table 1: The SINTL Instruction Set

- (`type-cast reg2 reg1 type`): The type-cast instruction accepts a register reg1 and a type and produces a value in reg2, converting the value in reg1 into a new value of type and placing it in reg2. The value in reg1, say of type type1, must have a castable or boxable relation to type. For instance, if reg1 contains an unboxed integer <ubox-integer>, then (`type-cast reg2 reg1` <integer>) would convert the box the unboxed integer and place its boxed equivalent in reg2. On the other hand, the instruction (`type-cast reg2 reg1` <ubox-real>) would cast the unboxed integer in reg1 into its unboxed real equivalent.

- (`ensure-type reg type`): The ensure-type instruction performs a type-check operation on reg, ensuring that once control has passed though this instruction, reg contains a value of type until its next assignment. The type type must be a subtype of <unit> (a boxed type), since these are the only values with runtime type information.

- (`primitive reg-res name reg1 ... regN`): The primitive instruction is used to access low-level resources provided by the current SINTL backend. The primitive instruction acts like a function call to a known procedure explicitly typed arguments and return type. The R5RS runtime system is implemented using a small set of about 30 of primitives that are easily mapped onto any target machine.

  The primitives provided by a backend SINTL conform to a rigorous syntax, and are easily entered using a simple macro. The primitives of a SINTL are defined using the `DEFINE-PRIMITIVE` syntax. Each primitive has a name, a SINTL type signature, a calling convention, an implementation, and an optional type implication. The type signature of a primitive is defined as:

type-signature	:=	((spec *) → spec)	[a function type]
spec	:=	type	[a SINTL type]
	|	...	[any number of previous]
	|	#f	[ignore type]

  where the (type ...) states that the primitive accepts an arbitrary number of arguments, all with a type type. For instance, the primitive for integer addition in Java is defined as:

```
(int+
 ((<ubox-integer> <ubox-integer>) -> <ubox-integer>)
 (inline ((iadd))))
```

  Which means that `int+` is a primitive of type unboxed integer →unboxed integer →unboxed integer and is implemented by the inline JVM instruction iadd.

Most of the primitives provided by a backend fit into this model; however, some primitives have a more sophisticated semantics that requires a more expressive definitional language. SINTL also accepts schema primitives (similar to the axiom schemas of first-order logic). A primitive schema effectively describes an infinite number of primitives of a regular form. The schema language expresses the primitive in terms of unification variables, allowing a scheme to be generalized over some of its arguments. For instance, the type testing predicates in SINTL are expressed using a simple schema:

```
(schema (?result type? ?type ?arg) (reg type? type reg)
  (type ((#f <unit>) → <ubox-boolean>)
        (inline (lambda (result type arg)
                    `(,@(stack-instructions arg)
                      (instanceof type))))
        (implies ?arg ?type)))
```

These declaration means that primitives of the instructions of the form (`primitive result type? arg` <integer>) are primitives from an ignored argument →<unit> →<ubox-boolean>, and are implemented by inserting inline the JVM instructions produced by the call to the lambda form. The lambda uses the JVM `instanceof` instruction to test whether the value produced by `arg` is an instanceof the class `type`. Further, the (`implies ?arg ?type`) expression informs the type inference engine that the register `?arg` has type `?type` in the true branch of a `branch-false?` instruction on the `?result` register of a `type?` primitive.

The simple primitive form and the primitive schema are sufficient to describe all of the low-level operations needed to implement an R5RS runtime system. Further, this primitive system nicely describes most of procedures provided by the target language, and can be easily adapted as a foreign function interface. By expanding the type system to include the standard types of the JVM and including a simple Scheme interface to declare the types and implementation of the Java libraries, we can easily inter-operate with native procedures, such as the libraries available on the JVM.

The fundamental assemble object of SINTL is the template. A template is a named sequence of SINTL instructions with a potentially empty list of typed arguments. An example sequence of instructions for the recursive version of Fibonacci:

```
(define (fib n)
  (if (< n 2)
     1
     (+ (fib (- n 1)) (fib (- n 2)))))
```

are the following, the first the result of compiling vanilla Scheme and the second with standard binding:

```
(define-template (fib (n <unit>))
  (global-ref lt <)
  (call t lt n 2)
  (branch-false? t recur)
  (return 1)
  (label recur)
  (global-ref minus -)
  (call t1 minus n 1)
  (global-ref f fib)
  (call t2 f t1)
  (call t3 minus n 2)
  (call t4 f t3)
  (global-ref plus +)
  (call t5 plus t2 t4)
  (return t5))

(define-template (fib2 (n <unit>))
  (primitive t int< n 2)
  (branch-false? t recur)
  (return 1)
  (label recur)
  (primitive t1 int- n 1)
  (global-ref f fib)
  (call t2 f t1)
  (primitive t3 int- n 2)
  (call t4 f t3)
  (primitive t5 int+ t2 t4)
  (return t5))
```

2.2 The Declarative Type System

The SINTL language is strongly-typed, meaning that every every register can be assigned a type in the SINTL type system. Because Scheme is a dynamically typed language, the SINTL type system must be flexible enough to express the need for runtime type checking. Further, since target machines typically distinguish between boxed values (which contain enough information to recover the type of their value at runtime) and unboxed values, the raw representation of a value without runtime type information, SINTL includes both boxed and unboxed types. Since the particular details of the type system is often system

dependent, SINTL uses a completely declarative type system to describe the domain of values for each particular backend. A SINTL backend is required to support the core set of types needed to compile Scheme code, but may extend or specialize the way the SINTL assembler handles instructions, types, and values.

A SINTL type system is a set of disjoint, discrete types (represented as symbols) with simple relations between the elements of this set. SINTL currently understands three kinds of relationships between types:

Subtyping If <A> is a subtype of <B>, then values of type <A> are permissible anywhere values of type <B> are required. Conversely, if a value of type <A> is required and a register reg is known to have type <B>, then register reg may in fact contain a value of type <A>. To ensure that the register reg contains such a value, an (`ensure-type reg` <A>) instruction should be inserted before using reg. A typical subtype relation occurs between <integer> and <number> and between <boolean>, <pair>, etc and <unit>, the supertype of all boxed types in SINTL.

Boxing If there is a boxed relation between <U> and <B>, where <U> is the unboxed version of <B>, then it is possible convert values of type <U> into and from values of type <B>. In a typical backend, the target machine can only directly operate on unboxed values, so it is critical to represent and reason about the transformations between boxed and unboxed values. To convert between <U> and <B>, a type-cast instruction is used. If reg1 contains a value of type <B>, then the instruction (`type-cast reg2 reg1` <U>) places the unboxed version of <B> in reg2. A typical boxed relation occurs between <ubox-integer> and <integer> and between <ubox-boolean> and <boolean>.

Castable Some values can be transformed into other values of a different type through a method qualitatively different than boxing. The types of such values have a castable relation between them, such that if <A> and <B> are castably related, then it is possible to convert a value of type <A> into a not necessarily equivalent value of type <B>. Since castable relations are often lossy, SINTL resorts to castable relations when converting from <A> to <B> only when no other relationship can convert <A> to <B>.

SINTL requires that the declared relations among types satisfy the following requirements. First, no two types can be mutual subtypes. Second, a type cannot be a subtype of multiple types; that is, SINTL only supports single inheritance of types. Although future versions of SINTL might permit multiple supertypes of a type, it was not an essential feature for the current project.

The current SINTL type system for the Java backend uses the set of types described in table 2.2.

<nil>	The type of '()
<env>	The type of a closure's environment
<template>	The type of a instruction sequence
<closure>	The type of a closure
<boolean>	The type of a boxed boolean value used in Scheme
<ubox-boolean>	The type of an unboxed boolean in target language
<symbol>	The type of Scheme symbols
<real>	The type of inexact numbers
<ubox-real>	The unboxed type of inexact numbers
<integer>	The boxed type of exact integer numbers
<ubox-integer>	The unboxed type of exact integer numbers
<number>	The superclass of boxed numbers
<pair>	The type of a cons cell
<char>	The type of a boxed character
<ubox-char>	The type of an unboxed character
<string>	The type of Scheme strings
<vector>	The type of Scheme vectors
<unit>	The supertype of all boxed types

Table 2: The SINTL Types

The SINTL type system is sufficiently simple that we can determine properties among types easily and efficiently. We can view the types system as a directed graph where the types are vertices and the relations between types are labeled edges. We can then determine the properties of the type system using standard graph algorithms on its corresponding type graph. Determining whether and how a value of type <A> can be converted into a value of type <B> is performed by running Dijkstra on the type graph, and selecting the best path between <A> and <B> if at least one exists. A best path between two nodes is a shortest path containing the a minimum casts and box/unbox edges.

2.3 The Assembler Phases

The current version of SINTL, which is under active development, accepts a list of templates and assembles them into JVM instructions or C procedures. The phases of the SINTL assembler are mutual independent; the actual phases and their ordering is controlled by the current SINTL backend. However, the most common phases required by backends are in order:

- Phase 1: Parsing of the templates into Scheme data structure. The templates are represented as a directed graph of basic blocks holding SINTL instruction and register data structures.

- Phase 2: Unreachable blocks are eliminated. Instruction sequences are cleaned up, adding labels to the start of every block and unconditional branches to fall-through blocks, among other simple canonicalizing transformations. Trivial blocks are copied back into incident blocks. For instance, blocks containing only unconditional branches to other blocks are propagated into their incident blocks and eliminated. Blocks containing only return instructions are copied back into their incident blocks, which improves type inference and code generation on backends permitting multiple return sites.

- Phase 3: Alias analysis and elimination. Unnecessary (SET! ...) instructions are found and eliminated from the instruction stream. The removal of all unnecessary register aliases cleans up the instruction stream, improving all later optimizations and analyses.

- Phase 3: Intra-block optimizations. A fixed point interaction of constant propagation and dead code elimination.

- Phase 4: Type inference to ensure instruction stream type safety. See section 3.

- Phase 6: Boxed literal collection. Literals that have both a boxed and unboxed representation in SINTL are assumed to be unboxed during type inference. However, many literals are immediately boxed by the type inference algorithm to satisfy type constraints imposed by the instructions. This phase determines which literals are used only once in an immediate (`type-cast` ...) to their boxed equivalents. Such literals are replaced by boxed literal equivalents and the `type-cast` instruction is replaced by the boxed literal instruction.

- Phase 5: Another round of intra block optimization, eliminating temporaries and literals introduced by type inference that are no longer needed.

- Phase 6: DAG Value Dependency Analysis. See section 4.1

- Phase 7: Backend code generation. Control passes to a backend specific generation algorithm which converts the optimized SINTL instruction stream into the backend's target language. Block tracing and instruction selection occur at this stage.

3 Type Inference in SINTL

Type inference in SINTL is used to ensure the type safety of the template instruction stream. In essence, SINTL accepts type unsafe templates and, through type inference, inserts ensure-type and type-cast instructions to make the template type safe. The principal goals of SINTL type inference are to insert the minimum number of type checks into the template to guarantee every instruction's type signature is satisfied and to transform values through boxed and castable types into those needed by calls, tail-calls, and primitive calls. The insight into the SINTL type inference algorithm is that it is often easier to insert type checks to ensure safety in an unsafe template than to remove unnecessary type checks in a safe template.

Many Scheme systems use type inference to improve the performance of their generated code, since type inference can prove that many type checks in scheme procedures are unnecessary. The SINTL type inference algorithm is compatible with Scheme level inference system, and in fact can improve their performance since SINTL provides access to partially unchecked primitives. Future versions of SINTL will be expanded to accept type annotation on registers and call instructions (such as those produced by soft-typing [15] or polyvariant analysis [8]), and will include a set based analysis, not the current singleton analysis. Nevertheless, the current type inference algorithm provides respectable performance and ease of use, especially when coupled with aggressive inlining on the front end.

3.1 Instruction Types

literal	literal-signature
global-ref	((name) → <unit>)
global-set	((name <unit>) → #f)
global-define	((name <unit>) → #f)
make-closure	((<env> name) → <closure>)
this-closure	(() → <closure>)
closure-env	((<closure>) → <env>)
make-env	((<env> index) → <env>)
env-parent	((<env>) → <unit>)
env-ref	((<env> index) → <unit>)
env-set	((<env> index <unit>) → #f)
set	set!-signature
call	((<closure> <unit> ...) → <unit>)
tail-call	((<closure> <unit> ...) → <unit>)
return	((<unit>) → #f)
label	((name) → #f)
branch	((name) → #f)
branch-false	((<ubox-boolean> name) → #f)
false	((<unit>) → <ubox-boolean>)
type-cast	type-cast-signature
ensure-type	ensure-type-signature
primitive	primitive-signature

where the five special cases are defined as:

(ensure-type r type) => #f

An ensure-type instruction does not have a type signature. It is handled specially by the instruction inference system. When an ensure-type instruction is encountered, the type of register r's T in the current tenv is checked against type. If type $\subseteq T$, then processing continues with r's type bound to type. Otherwise a type error has occurred, and the user is signaled.

(type-cast r2 r1 type) => (((typeof r1 tenv) #f) → type)

That is, a type-cast from r1 to r2 has a type signature of the type of r1 in the current tenv to the type cast type.

(literal r lit) => ((#f) → (typeof lit))

A literal instruction has a type signature of an ignored argument to the type of the literal.

(set! r2 r1) => ((#f) → (typeof r1 tenv))

The type of SET! is an ignored type argument to the type of r1 in the current type environment.

(`primitive res prim args` ...) => (`typeof prim args` ...)

Finally, the type of a primitive operation is the type of the primitive with respect to its arguments. Although most primitives have a simple type signature (such as int+: ((<ubox-integer> <ubox-integer>) → <ubox-integer>)), SINTL supports schema primitives whose type signature is potentially a function of its arguments.

3.2 Block Level Type Inference

The type inference algorithm is divided into two separate modules: basic blocks inference and instruction inference. The basic block inference algorithm uses the the instruction level type inference algorithm. The algorithm uses a work queue of blocks in need of inference, and maintains an input and output type environment for each block in the template. The algorithm begins by initializing the entry block of the template to a type environment in which the formal arguments to the template are bound to their declared types. The algorithm then loops while the work queue is not empty. Let B be the head block of the queue and let $tenv_w$ be a tenv. If input tenv of B, $tenv_in_b$, and the appropriate output tenvs of the predecessors of B, are unified. The output tenv of a block B_p used for B depends on the relationship between B and B_p. If $B_p \rightarrow B$ by an unconditional branch, then the output tenv is simply that of B_p. If $B_p \rightarrow B$ by an conditional branch, then the type implications in the output tenv of B_p must be processed in a copy of the output tenv. The type implications are notes attached to the bindings in the tenv that state the behavior of the bindings as control moves through a conditional branch. The unification attempts to unify the bindings in each tenv against the bindings in all others, returning the most unified type environment, if one exists. If the blocks cannot be unified, then a type error has been found and the algorithm halts. If the unification succeeds, then the unified tenv is assigned to $tenv_w$. If all of the tenvs where trivially equivalent, and hence unification did not produce a new tenv, there are two chooses for $tenv_w$. If the block B has not been visited by the algorithm yet, then B is processed with $tenv_w = tenv_in_b$. If B has already been visited, then since the input tenv of B is trivially equivalent to itself unified against all of the type environments of its predecessor, the output tenv of B will not change when inferred against $tenv_in_B$. In other words, block B has achieved quiescence in the fixed point, and hence we abort processing B and move on to the next block in the work queue.

Assuming that the block B needs to be processed and a valid tenv has been placed in $tenv_w$, we set the input tenv of B to $tenv_w$ and process the instructions in B in a copy of $tenv_w$. Once the instruction level infer is done, we are left with a new type environment reflecting type behavior of the instructions in B. We record this new tenv as B's output

tenv, and enqueue the successor blocks of B in the work queue. We then repeat the work loop.

3.3 Instruction Level Type Inference

The algorithm currently used by SINTL is a fixed point, block based algorithm. The unique entry block of the template is initialized to a type environment [tenv] (a mapping from register → type) populated by the types of the template's formal arguments. The block's instructions are processed in order, and the type of registers are recorded in the tenv after each instruction. At each instruction, a type signature for the instruction in the current tenv is calculated, and each instruction argument is checked against its desired type in the type signature. If the required type unifies with the current type in the tenv, then the argument is type safe at this instruction. If the argument does not unify, then their are several potential problems:

- The required type is an unboxed type but the current type is a boxed type. If their is a type path from the current type to the required unboxed type, then an instruction sequence is inserted before the current instruction converting the argument into its unboxed equivalent. The unboxed equivalent is placed in some register R and the argument A to this instruction is replaced with the register R. Type inference restarts at the begin to the inserted instructions. An example of this occurs when: $type_r$ = <ubox-integer>, $type_{arg}$ = <unit>, which produces the fixup sequence:

```
(ensure-type A <integer>)
(type-cast R A <ubox-integer>)
(I ... R ...)
```

- The required type is a boxed type but the current type is a unboxed type. Like the previous case, except that the current type is converted into its boxed equivalent if this is possible. This occurs when $type_r$ = <unit> and $type_{arg}$ = <ubox-boolean>, producing the fixup sequence (`type-cast R A <boolean>`).

- The required type is a unboxed type, and the current type is a different unboxed type. If there exists a sequence of casts between the required and current types, then the sequence of casts is inserted into the stream.

- The required type is boxed type, and the current type is a different boxed type. If the required type is a subtype of the current type, then an (`ensure-type A <required-type>`) is inserted. Otherwise, a type error has occurred, since the value in A cannot possible be of type required-type. The first case occurs

when $type_r$ = <integer> and $type_{arg}$ = <unit> with the sequence (**ensure-type A** <integer>). The second case occurs when $type_r$ = <integer> and $type_{arg}$ = <boolean>, which is a type error, since the value in A is a <boolean> and hence cannot ever contain a value of type <integer>.

Depending on its current mode, SINTL may attempt to fix assignments to a register of non-unifiable types. This stage is optional, as such assignments can be viewed as type errors from a strict point of view (as most strongly-typed languages do), or as implicit casts to a general, unifiable type.

If SINTL is attempting to fix the types of assigned registers, then an extra step occurs after instruction argument type inference has ensured the type safety of the instruction application. The return type of the instruction is checked against the previous type assigned to its result register. In most cases, the return type of the instruction will unify with the previous type, and type inference simply proceeds to the next instruction. In the case where the return instruction does not unify, SINTL assumes that the type of the result register should be boxed, and unifies the instruction's return type with the <unit> type. There are four cases to consider, depending the return type:

- The return type is a subtype of <unit>. The current instruction is type safe, but at least one of the previous assignments to the register must be fixed. This case occurs when a previous assignment had an unboxed type, and return type is a boxed type. The appropriate action to take is to boxed the previously unboxed assignment. For example, consider the following instruction stream:

(literal r 1)	((#f) → <ubox-integer>)
...	...
(call r proc arg1)	((<closure> <unit>) → <unit>)

  in which the first instruction loads an unboxed integer 1 into r, and the second instruction loads a <unit> into r. Such a situation is very common when translating procedures with an if expression, such as (**if** (< n 1) 1 (foo n)). The correct fix for such a sequence is

(literal t1 1)	((#f) → <ubox-integer>)
(type-cast r t1 <integer>)	((<ubox-integer> #f) → <integer>)
...	...
(call r proc arg1)	((<closure> <unit>) → <unit>)

  Although more efficient methods are available to find and fix the instructions assigning to the register, SINTL simply restarts type inference at the first instruction of the entry block of the template. Eventually the offending instruction will be reached, and appropriate action will be taken.

- The return type is not a subtype of <unit>, but there is a best path to convert a value of the return type to a value of a subtype of <unit>. In this case, an temporary register t1 is created, and the current instruction's result register r is replaced with t1, and a sequence of instructions is inserted converting t1 into a boxed value with the result in r. This occurs when the previous assignment was from a boxed type, and the current assignment is from an unboxed type.

- The return type is not a subtype of <unit>, and there is no path from the return type of a subtype of <unit>. In this case, type inference has detected an unrecoverable type error, and should signal the user.

Although I have enumerated each case in some detail, the process of converting a register A of type $type_{arg}$ to a type $type_r$ in register R can be implemented by running Dijkstra on the type graph and running a recursive conversion procedure over the edges of the path. Each case naturally falls out of the edge type between successive edges in the type graph. Further, since the SINTL type graph is very small, this routine can easily be precomputed or memorized to vastly improve performance. Although the current version of SINTL computes the path for every requested conversion, the algorithm contributes little to the overall runtime of the assembler.

The algorithm above is completely implemented and in daily use in the PM compiler. The implementation of the type inference algorithm and type environments requires 1007 lines of Scheme code, including comments.

4 The Reg→Stack Instruction Selection Algorithm

It may seem strange that SINTL, a register transfer language, is used as an intermediate language between Scheme and a stack machine like the JVM. Traditionally, compilers targeting register machines like native assembly or C use a register-based intermediate language, while those targeting stack machines like the JVM and Scheme48's VM use a stack-based language. Since we intend to use SINTL for both register and stack based targets, we needed an instruction representation compatible with both machine types. For simple code generation, stack representations are clearly better. In a stack representation, the arguments to all instructions are implicit and unnamed (they take their arguments off of the operand stack) and instruction results are pushed onto the operand stack. Scheme code generation can be accomplished with a simple walk of the Scheme expression. Further, since arguments and results are unnamed, a stack code generator need not worry about mapping expression results to names.

There are, however, substantial drawbacks to a stack machine representation for an optimizing code generator. A stack machine instruction can be viewed as an instruction with implicitly named arguments; that is, where the name of argument i is i, its

depth on the operand stack. An optimizing assembler frequently wants to compute the result dependencies among instructions, where instruction I depends on instruction J at i when at I, the value at depth i is generated by J. Unfortunately, the difficulty of recovering this information from stack machine instruction ranges from difficult to computational impossible (if the stack machine permits variable depth stacks across blocks). Without this information, an assembler is ham-strung, incapable of using many popular optimizations. Choosing a stack machine representation for an intermediate language is not a particularly good idea, as the first phase of most serious optimization algorithms will be to compute the dependencies among instructions, which is effectively converting the stack machine code into register transfer code. With these trade-offs in mind, the SINTL instruction set was specifically designed as a register transfer language.

There are several advantages to using an RTL representation in SINTL. First, it was easy to adapt most compiler analyses and optimizations to the SINTL instruction set, as these analyses are typically designed for RTLs or quad instruction form. Second, code generation to languages like C or MLRisc could be performed by a simple translation of the linear instruction stream. We would not need to simulate an operand stack or attempt to eliminate it from the instruction stream.

The major problem posed by choosing a RTL for SINTL was designing an algorithm that converts the RTL into near optimal stack machine code for targets like the JVM. In the following two sections, we present a novel linear time analysis and instruction selection algorithm that automatically converts a SINTL RTL instruction stream into stack machine code. The following algorithms assume that the target stack machine provides an arbitrary number of local variables which can be stored and loaded using two instructions (`LOCAL ?reg`) and (`SET-LOCAL! ?reg`) and values can be duplicated with (`DUP`). Local variables are assumed to be strongly typed, which implies that that values of unrelated types cannot be assigned to the same local variable.

Using an explicit conversion algorithm from a register architecture to a stack machine allows SINTL to support both C or MLRisc targets and JVM targets effectively. The C SINTL backend can treat SINTL as a RTL language similar to C and can use the C execution model directly. At the same time, the JVM SINTL backend can view SINTL as a stack machine language, and map the instruction set naturally onto the JVM stack machine code.

The algorithm operates on the basic block graph of a template. Conversion occurs in three phases: DAG value dependency analysis, local variable assignment, and stack machine instruction selection.

4.1 DAG Value Dependency Analysis

The DAG value dependency analysis uses a technique similar to DAG code generation described in [1] and value numbering [?]. For each basic block B in the basic block

graph, we walk the instruction stream forward to backward building a graph G. For each instruction I with return result r (if one exists) and register arguments $a_1, \ldots, a_n$, we create a node labeled r with value $\langle i, I \rangle$ in G, where i is the position of I in the instruction stream. For each argument a_k, we look for a node in G labeled a_k. If such a node I_{a_k} exists, we add an edge between I and I_{a_k} labeled k. If no such node exists, then the register a_k was produced outside block B. We create a new node in G labeled a_k with value $\langle -1, a_k \rangle$, and add an edge between this node and I with label k.

Once the walk of the instructions for each basic block B in the template is complete, we have a graph G_B describing the computational dependency among the instructions each block. Example DAGs are depicted in figures 6.4, 6.4, and 6.4.

4.2 Local Variable Assignment

Local variable assignment occurs as a linear walk over the basic block DAG produces by the value dependency analysis. First, the formal arguments of the template are marked as local variables. For each basic block DAG G_B, we walk the nodes of the graph, tagging registers as local when a node N:

1. If N is a node with value $\langle i, r \rangle$, where r is a register, then N represents an inter-block register. A inter-block register is a register whose value is computed in one block and whose value is used in at least one other block. We mark the register r as local.

2. If $in_degree(N) > 1$, then the value of N is used multiple times in its block B. To avoid computing any value more than once, the label register r of N is marked local.

3. If N is a node whose values is $\langle i,$ `(SET! r src)` $\rangle$ (a node produced by a `SET!` instruction), then the result register r of the `SET!` is marked local.

4.3 Stack Machine Instruction Selection

Instruction selection operates on the basic block DAG produced by the value dependency analysis, after local variables have been assigned. The instruction selection algorithm for a basic block B with DAG G_B begins by computing the dominating nodes in G_B. For the conversion process, a node N is a dominator of G_B if $in_degree(N) = 0$; that is, the value produced by N is not used anywhere in the basic block. We compute the set of dominating nodes of G_B and order the dominators by their index i stored in the node's value $\langle i, v \rangle$. For each dominator node N, we call the algorithm's main routine *Instructions* with N.

Instructions(N) examines N with value $\langle i, v \rangle$ and label r. If N has already been visited (implying N was a multiple use node) or if v is a register, we invoke the *Inst_Select*

function with the argument (`LOCAL r`). That is, we load the value stored in the local variable associated with r. If N has not been flagged, then v is an actual instruction that needs to be generated. We invoke $Inst_Select(v)$. If the label register r of N is marked as a local register, then we append (`DUP`) and (`SET-LOCAL! r`) instructions to the value returned by $Inst_Select(v)$.

The $Inst_Select(inst)$ function provided to the instruction selection algorithm accepts a SINTL instruction and the three extended instructions `DUP`, `LOCAL`, and `SET-LOCAL!`. It returns a sequence of instructions in the target stack machine language for *inst*. The function *Instructions* is reentrant, and generated occurs by $Inst_Select(inst)$ recursively *Instruction* on the arguments of *inst* to compute the stack instructions for each argument. With the stack instructions for each argument, the instruction selector returns a sequence of instructions for *inst*. In other words, *Instructions* and *Inst_Select* are mutually recursive co-routines functions, each calling the other to select stack instructions for the instructions in the basic block B.

Ultimately, the above three algorithms all operate in $O(n)$, where n is the number of instructions in the template. There are several aspects of these algorithms that are important to point out. First, the algorithms right now do not respect the evaluation order described by a Scheme program. In the following Scheme snippet:

```
(define (proc z)
  (let ((x (foo z)))
    (let ((y (bar z)))
      (+ y x))))
```

The call to `foo` always executes before the call to `bar`. It is possible that the reg→stack conversion algorithm, depending on the instruction selection function *Inst_Select*, will compute (`bar z`) before (`foo z`). In other words, the algorithm does not enforce Scheme's order of evaluation. However, R5RS Scheme does not specify the order of evaluation for nested applications, such as:

```
(define (proc z)
  (+ (bar z) (foo z)))
```

The reg→stack algorithm can select any evaluation order for such expressions. Unfortunately, the SINTL instruction set is not expressive enough to describe the order dependency of let expressions. Another approach is to mark `call` and `tail-call` instructions which an immovable marker, locking their evaluation order relative to other expressions. All in all, the conversion algorithm works quite successful, and is capable of compiling the entire R5RS (minus full continuations) correctly.

5 The JVM Backend

SINTL current includes three backends: the Java Virtual Machine, a currently undisclosable virtual machine, and C. Several additional backends are in development or in planning: C– [9], MLRisc [7], and to the Python VM. The JVM backend will be described in detail in the following sections because it is fully functional and includes several novel representation choices. This section will be concluded with performance measurements and comparison with other Scheme compilers.

5.1 Why the JVM

Several compilers for high-level programming languages already target Java or the JVM (see [12], [10], [3]) and have achieved reasonable performance. The JVM is an attractive target for high-level programming languages for many reasons. The JVM is highly portable, running on most modern operating systems and machine architectures. Like most virtual machines, the JVM achieves acceptable performance when interpreted; there are, however, many publicly available Just-In-Time compilers that vastly improve the performance of executing JVM code. Since the JVM includes a garbage collector, a high-level language targeting the JVM receives a reasonably high-quality garbage collector for free. Perhaps most significantly, a high-level language on the JVM should gain easy access to the multitude of standard Java libraries. Ideally, by compiling Scheme to the JVM, we should immediately be able to use a cross-platform windowing system, secure network communications, and applet capabilities. In many senses, a language is nothing more than the sum of its libraries; by targeting the JVM, high-level languages become more attractive to real programmers whose business is to accomplish specific tasks.

However, serious issues arise when compiling a high-level language like Scheme to the JVM. Scheme provides capabilities like `call-with-current-continuation`, lexical scoping, and first-class closures that require facilities not immediately present on the JVM. Consequently, a significant challenge for a Scheme compiler targeting the JVM is selecting appropriate representations for Scheme constructs that permit inter-operability between Scheme and JVM libraries without sacrificing too much performance. Our approach to the JVM backend is to use the capabilities of the JVM to implement as much of Scheme's functionality as possible, and avoid representations which would hinder inter-operabilty. These design decisions lead immediately to only support one-shot (or upward) continuations using the JVM's exception mechanism, since representing frames as heap-allocated records would be prohibitively slow. Worse yet, such a representation would interact badly with JITs, which are optimizing to produce good native code for the standard representations of Java constructs. We ultimate choose the following representations:

1. Closures. A closure would be represented as a class, with environments stored as

vectors in the private class fields. Since we do not support full continuations, we can use the JVM stack for computation and local variables to hold most Scheme variables. The only variables that the front end compiler puts into environment vectors are those shared between lexically scoped closures. The instructions implementing a closure are represented by apply methods in subclass of the closure superclass. The arguments to a procedure are passed using the JVM argument passing system, at least up to four arguments.[2] When the number of arguments to a procedure exceeds four, the extra arguments are passed as a list and destructured in the apply method of the closure class. This is similar to the argument passing mechanism used in Bigloo [14], where the number of argument dispatching resides in the virtual table lookup of the JVM. Variable number of argument procedures are implemented as an apply routine accepting up to four fixed arguments and a list of remaining arguments. The closure class of an vararg procedure includes apply methods to forward calls with zero to four arguemtns to the vararg apply method.

2. Applications. A application is a virtual call to an apply method on a closure object. Up to four arguments are passed directly, and all remaining arguments are passed as a list to the apply method.

3. The remaining types. Unboxed booleans, integers, reals, and characters are implemented using the JVM int and real types. Boxed integers and reals are java.lang.Integer and java.lang.Float objects. Pairs represented using a Scheme specific classes, and Scheme '() is represented using the JVM `null` object. Symbols are java.lang.Strings, Strings are java.lang.StringBuffers, and vectors are JVM object arrays.

There are several interesting extensions to Scheme that are directly supported on the JVM backend. A restricted form of the case-lambda syntax [?] is supported by the JVM backend. When the templates produced by a **case-lambda** form reach the JVM backend, they are translated apply methods on the *same* closure class. For instance, consider the realistic implementation of the Scheme + operator, simplified for integers only.

```
(define +
  (case-lambda
    (()   0)
    ((x)  x)
    ((x y) (primitive int+ x y))
    (rest  (let loop ((val (car r)) (r (cdr rest)))
             (if (pair? r)
```

[2] This is necessary since the JVM does not support variable number of argument procedures.

```
(loop (+ val (car r)) (cdr r))
val)))))
```

It uses **case-lambda** to implement fast paths for the common zero, one, and two argument cases. The PM compiler using JVM SINTL would translate this definition into the JVM equivalent of the following closure:

```
public proc_PLUS extends Closure {
    public apply() {
        return box_integer(0);
    }

    public apply(Object x) {
        return x;
    }

    public apply(Object x, Object y) {
        Integer x_i = (Integer)x;
        Integer y_i = (Integer)y;
        int r = unbox_integer(x_i) + unbox_integer(y_i);
        return box_integer(r);
    }

    public apply(Object a1, Object a2, Object a3) {
        return this.var_apply(cons(a1, cons(a2, cons(a3, nil))));
    }

    public apply(Object a1, Object a2, Object a3, Object a4) {
        return this.var_apply(cons(a1, cons(a2, cons(a3, cons(a4, nil)))));
    }

    public apply(Object a1, Object a2, Object a3, Object a4, List l) {
        return this.var_apply(cons(a1, cons(a2, cons(a3, cons(a4, l)))));
    }

    public var_apply(List rest) {
        Object val = car(rest);
        Object r = cdr(rest);
        while (r instanceof Pair) {
            val = this.apply(val, car(r));
            r = cdr(r);
```

```
        }
        return val;
    }
}
```

Supporting case-lambda at the Scheme level radically improves the performance of the R5RS runtime system. Using the **case-lambda** syntax, we can write efficient fast-paths for procedures like arithmetic operations and list operations, where the standard invocations uses two arguments, but R5RS defines the procedures as accepting a variable numbers of arguments. Additionally, procedures accepting optional arguments like the I/O routines can be optimized, such as in:

```
(define display
  (case-lambda
    ((obj) (display obj (current-input-port)))
    ((obj port) ...)))
```

Most Scheme implementations define these sorts of procedures as vararg procedures and explicitly check for the optional argument or dispatch to a fast-path in the body of the code. Although the check is relatively fast, such a definition forces memory allocation at each call to the routine for the vararg list, which can drastically hurt performance.

In R5RS Scheme, all procedures (even standard procedures like `cdr`, `car`, +) are redefinable. In most cases, a Scheme compiler cannot statically determine the mappings from identifiers to values. Consequently, it is critically important that global variables be resolved in an efficient manor, preferably in constant time with respect to accesses of the variable. The JVM backend resolves global variables at class construction time by placing global variable lookup in the class static constructor. The performance gains of such an optimization are enormous, even relative to doing variable lookup at class construction time. This behavior is understandable, given that many common procedures allocate escaping lambdas, such as

```
(define (incrementer by)
  (lambda (x)
    (+ x by)))
```

If global variable look occurs at closure construction time, every call to `incrementer` would allocate a closure and hence lookup the binding of +. The costs of variable lookup at each call site would be prohibitly expensive, and was never consider a viable implementation option.

We encountered several unexpected difficulties when compiling to the JVM. The type inference algorithm removes many type checks in the runtime system code. Unfortunately, the JVM verifier cannot handle some of the type safe operations that SINTL

proves are possible to remove. Consequently, our compiler produces two different code sequences for the JVM, a verifiable version with unnecessary checks to satisfy the verifier, and a non-verifiable without these checks. Performance results for both versions are presented in section 7.

6 An Extended Compilation Example

This section includes an extended compilation example to demonstrate the phases of the compiler and the utility of each phase of the SINTL assembler. We will consider the compilations of the classic recursive Fibonacci procedure:

6.1 Front End Compilation

```
(define (fib n)
  (if (< n 1)
     1
     (+ (fib (- n 2)) (fib (- n 1)))))
```

After front end macro expansion, a-normalization, and function inlining, we produce the following procedure for code generation to SINTL: [3]

```
(define fib
  (lambda (n_1)
    (let (v_131 (primitive int< n_1 1))
      (if v_131
         1
         (let (v_133 (primitive int- n_1 2))
           (let (v_135 (primitive int- n_1 1))
             (let (v_136 (fib v_133))
               (let (v_137 (fib v_135))
                 (primitive int+ v_136 v_137)))))))))
```

The output of the front end code generator from a-normal form to SINTL templates produces:

```
(define-template (fib (n_1 <unit>))
  (primitive t1 int< n_1 2)
  (branch-false? t1 case1)
```

[3]The front end compiler does not inline arithmetic operations yet. The fib code segment includes inlined arithmetic operations for clarity. Benchmarks presented in section 7 do *not* have inlined arithmetic operations.

```
  (primitive v_133 int- n_1 2)
  (primitive b_135 int- n_1 1)
  (global-ref p1 fib)
  (call v_136 p1 v_133)
  (global-ref p2 fib)
  (call v_137 p2 v_135)
  (primitive retval int+ v_136 v_137)
  (branch ret)
  (label case1)
  (set! retval 1)
  (branch ret)
  (label ret)
  (return retval))
```

6.2 Initial SINTL Phases

As you can see, not only is this code sequence suboptimal, the retval register is assigned two dissimilar types (an unboxed integer and a unit), the primitives are called with arguments of incorrect types (a <unit> where an <ubox-integer> is required), and there are no type checks on any of the operations.

After the first three phases of SINTL, during which parsing, intra-basic block optimizations, code expansions, and branch optimizations have occurred, the template looks like:

```
(define-template (fib (n_1 <unit>))
  (label ENTRY)
  (literal lit1 2)
  (primitive t1 int< n_1 lit1)
  (branch-false? t1 case1)

  (label LABEL1)
  (literal lit2 2)
  (primitive v_133 int- n_1 lit2)
  (literal lit3 1)
  (primitive b_135 int- n_1 lit3)
  (global-ref p1 fib)
  (call v_136 p1 v_133)
  (global-ref p2 fib)
  (call v_137 p2 v_135)
  (primitive retval int+ v_136 v_137)
  (return retval)
```

```
(label case1)
(literal lit4 1)
(return lit4))
```

The return instruction has been propagated back into its two incident blocks.

6.3 Type Inference

Now type inference begins, with the first instruction of the entry block in a type environment where n_1 →<unit>. After (literal lit1 2), the type environment also contains a binding from lit1 →<ubox-integer>.

The instruction (primitive t1 int< n_1 2) has a type signature ((<ubox-integer> <ubox-integer>) → <ubox-boolean>). Since n_1 is of type <unit>, we need to fix the instruction stream as follows:

```
(ensure-type n_1 <integer>)
(type-cast s1 n_1 <ubox-integer)
(primitive t1 int< s1 lit1)
```

in which we checked that n_1 contains an integer, and unboxed the value to satify the type requirements of the int+ primitive. The (branch-false? t1 case1) instruction passes type inference without a problem. Now the entry block has been processed, and we are left with a type environment:

```
n_1   →  <integer>
lit1  →  <ubox-integer>
s1    →  <ubox-integer>
t1    →  <ubox-boolean>
```

We begin processing the next block, LABEL1. Since n_1 is known now to be of type <integer>, each use of n_1 need only unbox its result, and need not check its type. After type inference, we are left with the type safe template:

```
(define-template (fib (n_1 <unit>))
  (label ENTRY)
  (literal lit1 2)
  (ensure-type n_1 <integer>)
  (type-cast s1 n_1 <ubox-integer>)
  (primitive t1 int< s1 lit1)
  (branch-false? t1 case1)

  (label LABEL1)
  (literal lit2 2)
  (type-cast s2 n_1 <ubox-integer>)
  (primitive v_133 int- s2 lit2)
```

```
(literal lit3 1)
(type-cast s3 n_1 <ubox-integer>)
(primitive v_135 int- s3 lit3)
(global-ref p1 fib)
(type-cast s8 v_133 <integer>)
(call v_136 p1 s8)
(global-ref p2 fib)
(type-cast s9 v_135 <integer>)
(call v_137 p2 s9)
(ensure-type v_136 <integer>)
(type-cast s4 v_136 <ubox-integer>)
(ensure-type v_137 <integer>)
(type-cast s5 v_137 <ubox-integer>)
(primitive retval int+ s4 s5)
(type-cast s6 retval <integer>)
(return s6)

(label case1)
(literal s7 1 <integer>)
(return s7))
```

At this point, we see that several further optimizations are possible. Common subexpression elimination, at either intra or inter block, would eliminate the many redundant unbox operations on `n_1`. The current version of SINTL, however, does not perform CSE; we expect to add CSE in the near future.

6.4 Reg→Stack Instruction Selection

The DAG dependency analysis is run on each basic block, producing the DAGs for the `ENTRY` block in figure 6.4, the `LABEL1` block in figure 6.4, and the `CASE1` block in figure 6.4.

At this point, control enters the JVM backend. The reg→stack instruction selection algorithm is run, the JVM instructions are peephole optimized, producing the following sequence of JVM instructions in Jasmin syntax:

6.5 JVM Source Code

```
.method public apply(Ljava/lang/Object;)Ljava/lang/Object;
    .limit locals 2
    .limit stack 4
    .var 1 is n_1 Ljava/lang/Object;
```

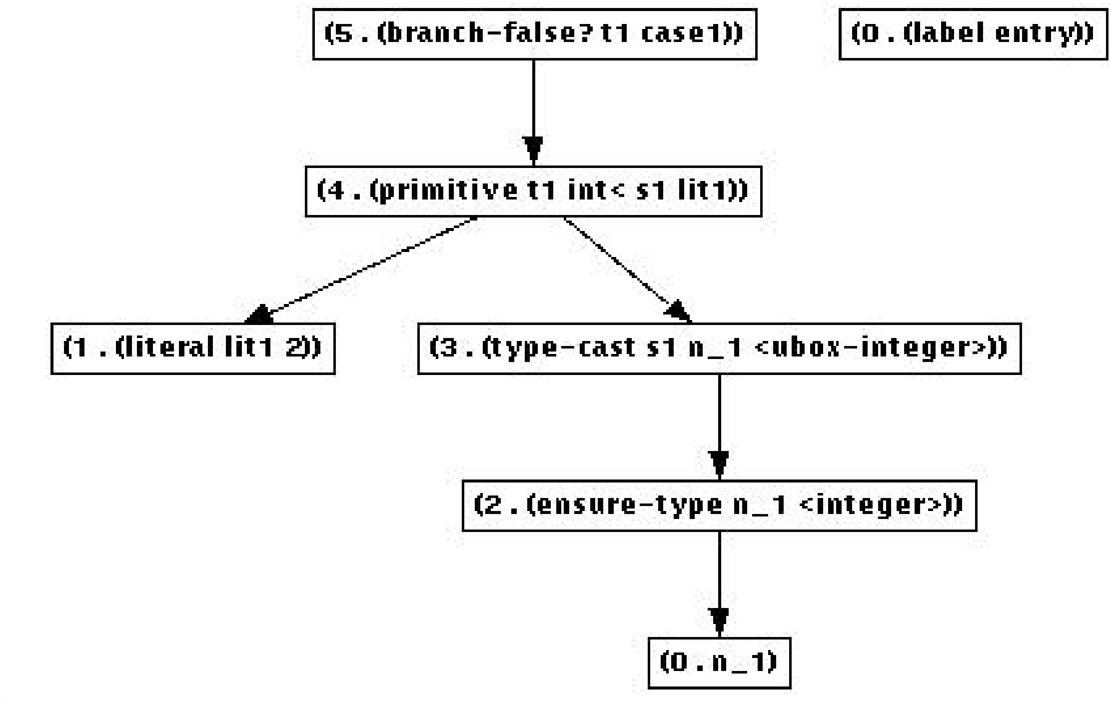

Figure 1: DAG for Entry block

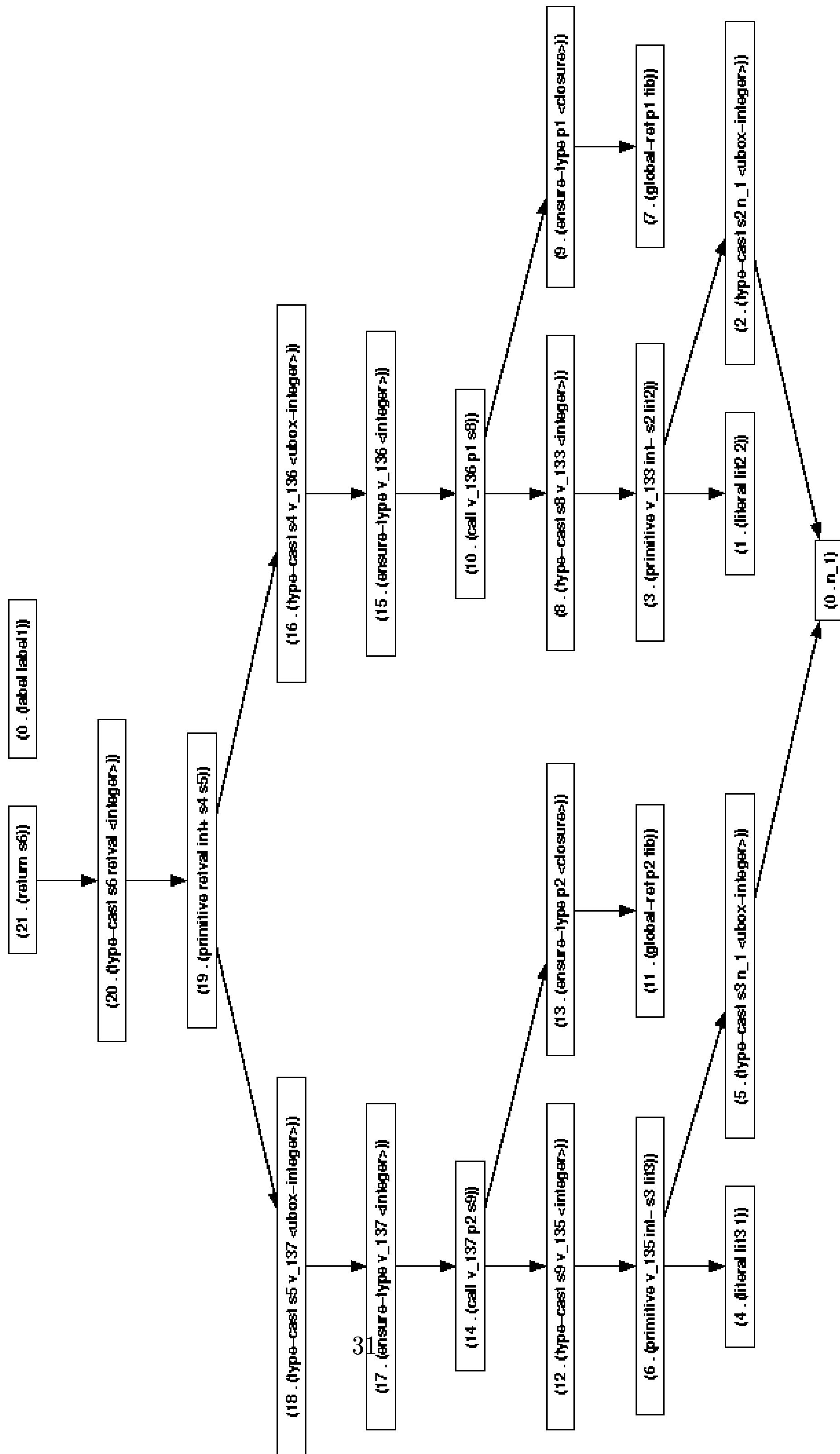

Figure 2: DAG for LABEL1 block

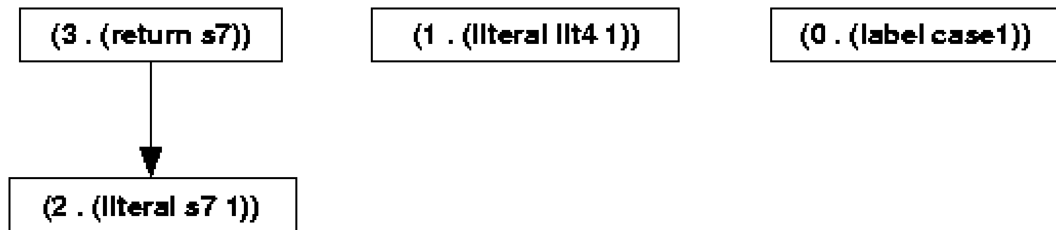

Figure 3: DAG for CASE1 block

```
entry:
    aload_1
    checkcast java/lang/Integer
    invokevirtual java/lang/Integer/intValue()I
    iconst_2
    if_icmpge lbl626
SLABEL59:
    getstatic tmplt_fib620/lit0 Ljava/lang/Integer;
    areturn
lbl626:
    getstatic tmplt_fib620/fib Lscheme/Location;
    getfield scheme/Location/val Ljava/lang/Object;
    checkcast scheme/Closure
    aload_1
    invokevirtual java/lang/Integer/intValue()I
    iconst_1
    isub
    invokestatic scheme/System/box_integer(I)Ljava/lang/Integer;
    invokevirtual scheme/Closure/apply(Ljava/lang/Object;)Ljava/lang/Object;
    checkcast java/lang/Integer
    invokevirtual java/lang/Integer/intValue()I
    getstatic tmplt_fib620/fib Lscheme/Location;
    getfield scheme/Location/val Ljava/lang/Object;
    checkcast scheme/Closure
    aload_1
    invokevirtual java/lang/Integer/intValue()I
    iconst_2
    isub
```

```
    invokestatic scheme/System/box_integer(I)Ljava/lang/Integer;
    invokevirtual scheme/Closure/apply(Ljava/lang/Object;)Ljava/lang/Object;
    checkcast java/lang/Integer
    invokevirtual java/lang/Integer/intValue()I
    iadd
    invokestatic scheme/System/box_integer(I)Ljava/lang/Integer;
    areturn
.end method
```

where the static class variable `lit0`contains a static class field holding the boxed object representation of the literal 1.

7 Performance Results for JVM-SINTL

We benchmarked the PM compiler with the SINTL JVM backend against several scheme compilers. All of the benchmarks presented are from the Gabriel scheme benchmarks, a port of the classic Gabriel [6] benchmarks for LISP compilers by William Clinger. The current compiler is capable of running most, but not all, of the Gabriel benchmarks. The following benchmarks, including a short description of each, follows:

tak A vanilla implementation of the triply recursive Takeuchi function. The tak benchmark measures the performance of arithmetic operations on integers and recursive and tail-recursive calls.

takl The Takeuchi function above, using lists as counters. In this variant, a list of length n is created for each argument to the Takeuchi function. The tak function implements $<$ as a tail-recursive (`shorter? l1 l2`) predicate on lists. Subtraction is implemented by taking the `cdr` of the counter list. Since the counter lists are preallocated, takl measures the performance of list processing operations without consing.

deriv A simple symbolic derivative function using lists and symbols to represent arithmetic expressions. Deriv uses a **cond** on the `car` of the expression list to compute the derivative on the expression. The deriv function cons up the answer as a list; hence the deriv benchmark primarily measures the performance of branches, `cons` and `list`.

destruct A benchmark measuring the performance of destructive list operations, using a simple large function and many internally defined loops.

cps-tak A continuation passing style variant on the tak benchmark. Measures the performance of firse class procedures and tail-recursion.

dderiv A variant of the deriv benchmark, using LISP style property lists to perform expression dispatch. The property lists are implemented using a doubly nest global association list. Dderiv measures the performance of `assq`, `cons` and `list`.

div-iter An iterative division benchmark using lists as counters. div-iter does not cons, and hence measures the performance of list operations. The iteration is accomplished using an internally defined procedure.

div-rec A recursive variant of the div-iter benchmark.

We choose several compilers to use to compare the performance of the JVM backend. Obviously, we must compare ourselves to other Scheme compilers using the JVM. The only publicly available Scheme system using the JVM is Kawa [3], [2]. Since the JVM backend is running on a virtual machine (although with a JIT compiler), it is important to consider the performance of the backend relative to other Scheme virtual machines. For this reason we selected Scheme48 [13], a byte-coded virtual machine specific to the Scheme language. Our next comparison compiler is the DrScheme system [5] from the PLT group at Rice. We consider both their byte-compiled and native compiled (via C) performance. Finally, to estimate our performance relative to an optimizing, native code compiler, we selected the Bigloo [14] compiler from INRIA.

More detailed information on each compiler is provided below:

JVM-SINTL Benchmark results were producing using Sun Microsystems Java 1.2.2 distribution for Linux with native green threads using the Inprise just-in-time compiler from Borland. The Java heap was set to an initial 5,000,000 words, where each benchmark was run using the same heap. The benchmark numbers are using a non-verifiable version of the JVM SINTL backend. The front end compiler to SINTL inlined standard bindings for many Scheme procedures. [4]

Kawa Using the current stable version of Kawa, version 1.6.1. The benchmarks were compiled using the Kawa compiler into .class files with standard bindings for primitives. The compiled files were loaded into the Kawa REPL, run and timed. [5]

Scheme48 Using a vanilla Scheme48 0.53 distribution build with egcs-2.91. The benchmarks were compiled in Scheme48 using ,bench mode, which allows the Scheme48 compiler to inline primitive operations.

[4] The front end compiler currently does not transform internal, non-escaping, tail-recursive procedures into loops. This optimization, which is performed by Bigloo, Kawa, and MZC, would vastly improve the performance of the destruct and div-iter benchmarks. A version of this analysis is currently in development.

[5] It is not clear why the Kawa compiler is this slow or cannot compile many of the benchmarks. I am current contacting the Kawa developers to uncover the reasons. More recent (and less stable) versions of the compiler are significantly faster.

	tak	takl	deriv	destruct	cps-tak	dderiv	div-iter	div-rec	Total
PM	1.46	2.62	7.13	17.02	9.34	10.9	4.67	4	59.83
Kawa 1.6.1	14.88	119.37	36.56	94.02	-	58.77	-	-	440.28
Scheme48	1.56	15.06	6.82	8.42	2.87	7.93	4.67	5.23	67.78
MzScheme	3.2	33.72	8.89	20.5	4.99	10.61	7.97	7.8	131.87
Mzc	1.5	14.68	4.15	3.92	1.9	5.42	1.64	2.98	50.89
Bigloo	0.15	0.43	1.09	1.19	1.38	1.24	0.62	0.62	7.19
C-SINTL	0.28	0.45	1.91	3.58	3.71	3.11	1.1	0.96	15.1
Kawa 1.6.66	1.63	4.25	13.29	23.33	6.13	11.17	5.96	6.08	71.83

Table 3: Scheme systems performance on the Gabriel Scheme benchmarks. The numbers are running time in seconds. A dash indicates that the benchmark did not compile or ran incorrectly.

MzScheme MzScheme was benchmarked using both their byte-code and native code compiler. The byte-code results were collected by loading the benchmark code into the interpreter (which performs byte-code compilation) and running each benchmark. We used MzScheme version 1.01.

mzc The mzc benchmark results were collected using the mzc native code compiler version 1.01 with –prim and –lite options to perform primitive inlining and lightweight closure conversion.

Bigloo The Bigloo compiler is an optimizing Scheme compiler producing native code executables via ANSI standard C. Bigloo benchmarks were collected by compiling the benchmarks into a single stand-alone application using version Bigloo version 2.1. We used optimization level 06, the highest available, which includes heap→stack optimizations, method inlining, loop unrolling and inlining, and macro optimizations.

All performance measurements we conducted on a 233 MHz Intel Pentium II processing running RedHat Linux 6.1. Each benchmark result represents the sum cost of running the benchmark 10 times. Benchmark numbers are presented in table 7.

Finally, figure 7 depicts the total runtime of the entire Gabriel benchmark suite for each compiler tested. This numbers must be taken with a grain of salt, as the much of the details of each compiler have been concealed in the summation. To further caution conclusions from this data, it is important to note that each benchmark in the Gabriel suite is not calibrated to require a fixed amount of time. A summation such as presented in figure 7 consequently weights performance on each benchmark differently,

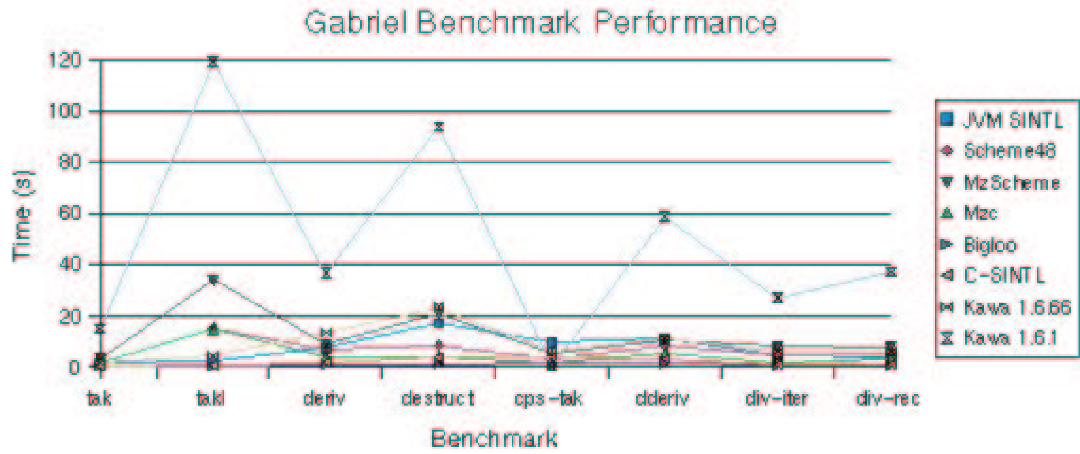

Figure 4: Performance results for all examined Scheme compilers on the Gabriel benchmark suite.

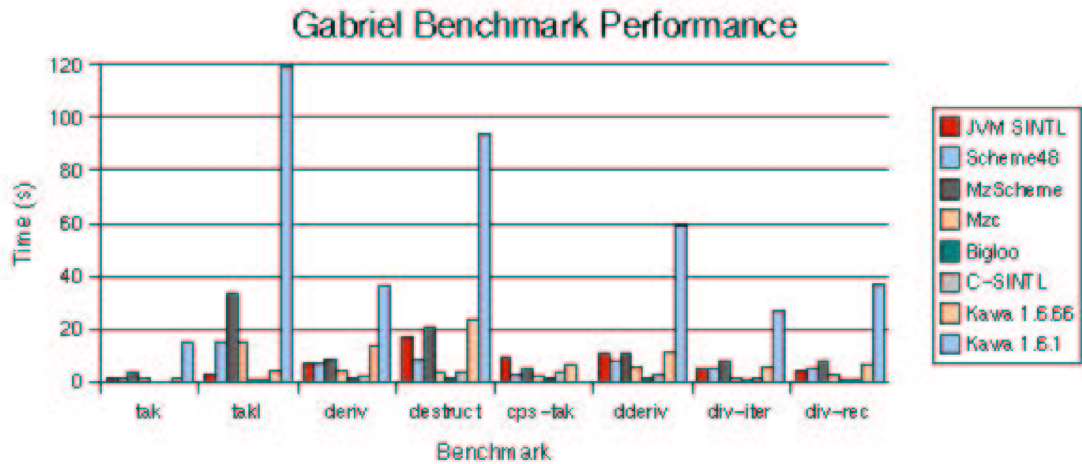

Figure 5: Performance results for all examined Scheme compilers on the Gabriel benchmark suite, in bar graph format.

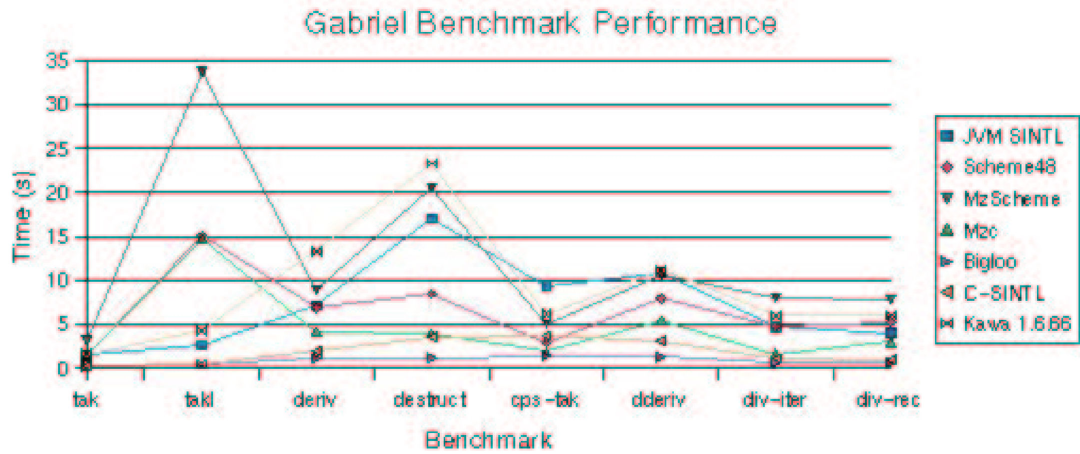

Figure 6: Performance results for all examined Scheme compilers on the Gabriel benchmark suite, minus Kawa 1.6.1.

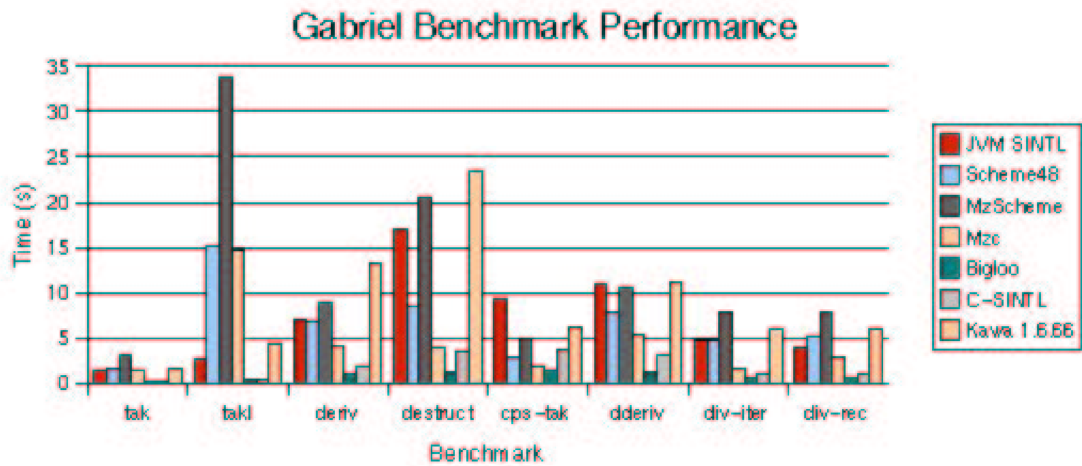

Figure 7: Performance results for all examined Scheme compilers on the Gabriel benchmark suite, minus Kawa 1.6.1, in bar graph format.

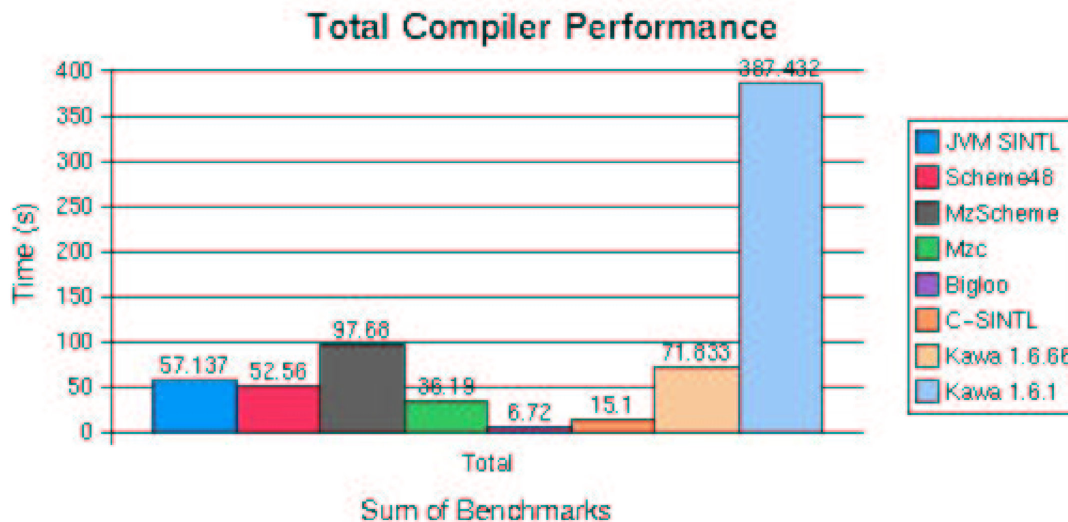

Figure 8: The total performance results for all examined Scheme compilers on the Gabriel benchmark suite. Each number is the summed total of the seconds required to run all of the benchmarks in the Gabriel benchmark suite.

and consequently poor performance on a single benchmark may dispropotionately influence the total time consumed by the compiler. That said, large differences in the total performance are still interesting and should be presented.

Several comparisons are worth considering in more detail. The performance of JVM-SINTL relative to Kawa 1.6.1 and Kawa 1.6.66 is extremely useful to judge to quality of the PM compiler. As a minimum requirement, JVM-SINTL should produce better (more efficient) code than its closest competitor compilers. Figures 7 and 7 depict the performance of JVM-SINTL relative to Kawa 1.6.1 and 1.6.66, respectively.

Another useful comparison is between JVM-SINTL and Scheme48. This comparison gives some indication of the performance benefits of compiling to a Scheme-specific virtual machine, relative to a well-supported but inhospital machine like the JVM. Figure 7 shows that the performance of JVM-SINTL and Scheme48 differ radically on many of the benchmark. However, *in toto*, JVM-SINTL and Scheme48 consume about the same amount of time on all of the benchmarks.

By the same logic, the C-SINTL backend should be considered relative to the two other Scheme-¿C compilers, Bigloo and mzc. Such a direct comparison is displayed in figure 7. Notice that, while C-SINTL handily beats mzc, the Bigloo compiler remains a little over twice as fast as C-SINTL.

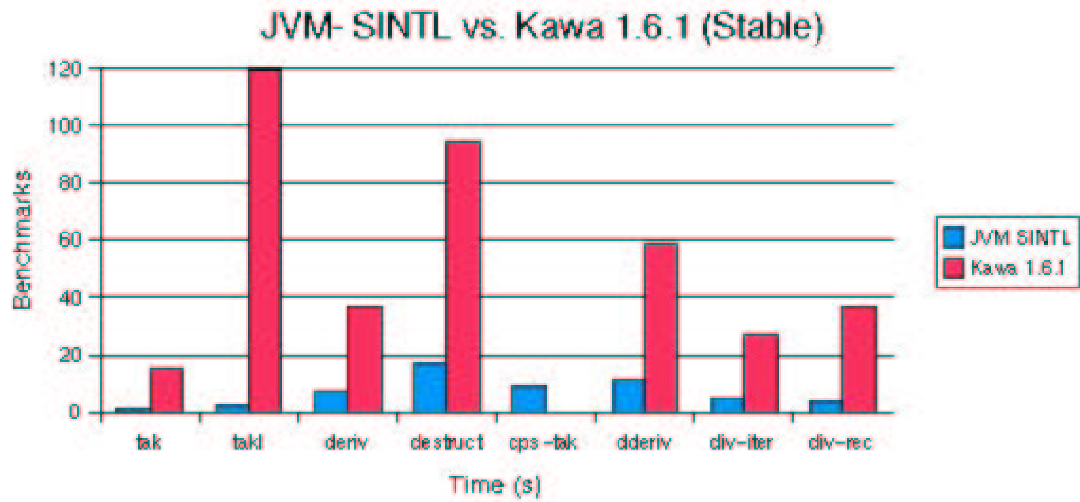

Figure 9: Performance of JVM-SINTL vs. Kawa 1.6.1 on the Gabriel benchmark suite.

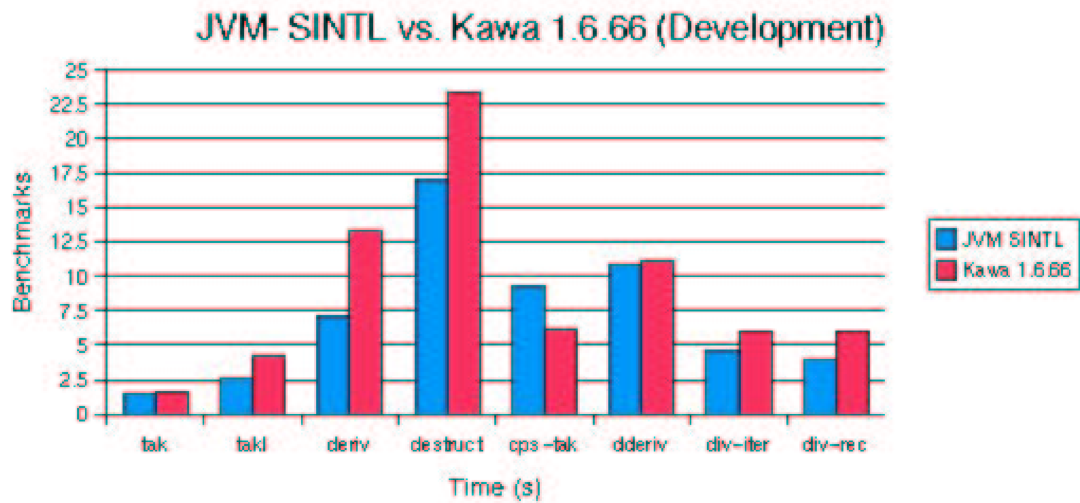

Figure 10: Performance of JVM-SINTL vs. Kawa 1.6.66 on the Gabriel benchmark suite.

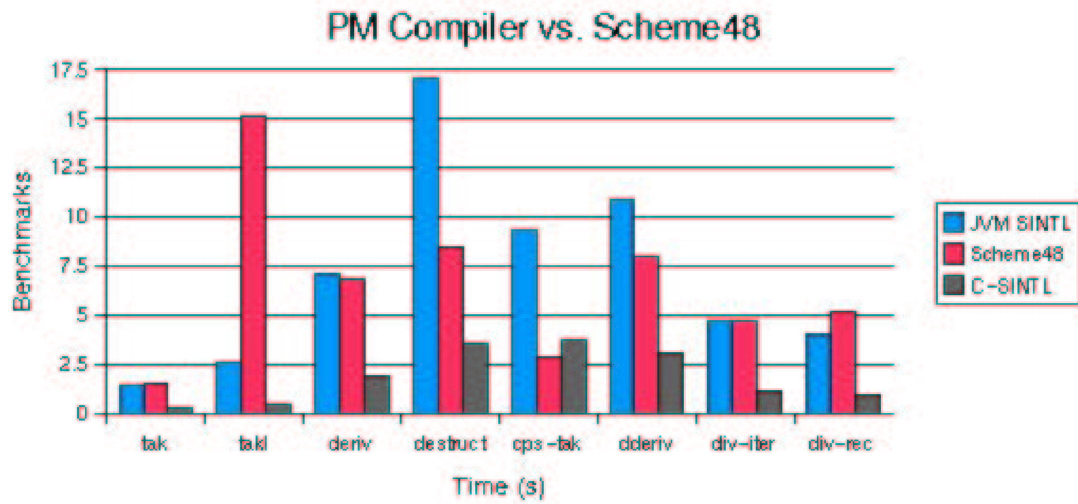

Figure 11: Performance of JVM-SINTL vs. Scheme48 0.53 on the Gabriel benchmark suite.

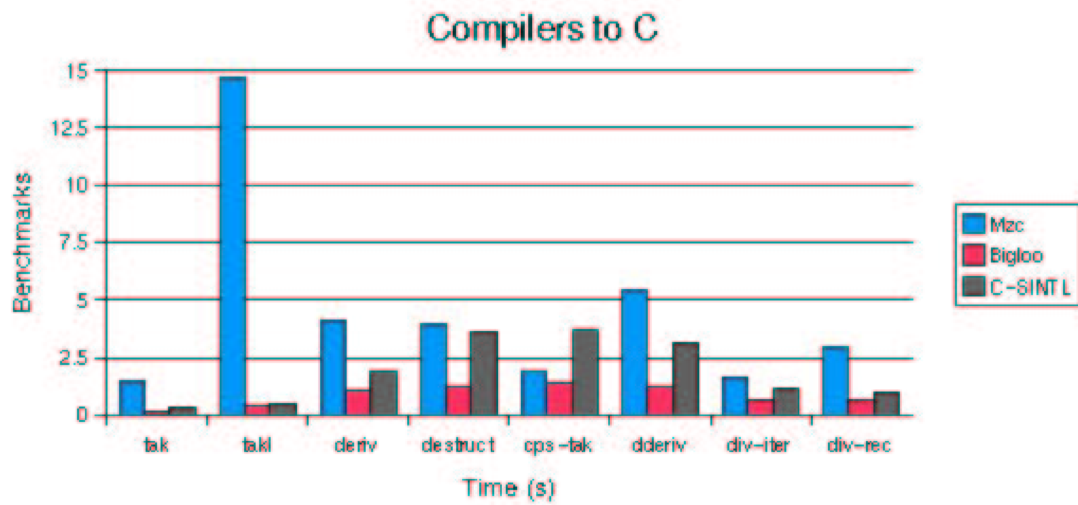

Figure 12: Performance of C-SINTL, mzc, and Bigloo on the Gabriel benchmark suite.

	tak	takl	deriv	destruct	cps-tak	dderiv	div-iter	div-rec	Total
Verifiable	2.2	10.3	8.9	21.1	9.4	11.1	7.3	6.6	87.3
Unverifiable	1.7	10.5	8.9	18.5	8.7	12.1	7.5	6.6	85.0
Verifiable, inline	1.9	2.6	7.3	19.5	8.9	11.4	4.9	4.2	63.5
Unverifiable, inline	1.5	2.6	7.1	17.0	9.3	10.9	4.7	4.0	59.8

Table 4: SINTL JVM backend performance among different backend configurations.

	tak	takl	deriv	destruct	cps-tak	dderiv	div-iter	div-rec	Total
Verifiable	100	100	100	100	100	100	100	100	100
Unverifiable	76.9	102.2	100	87.6	93.3	109.1	102.8	100.9	97.4
Verifiable,inline	87.5	25.2	82.0	92.2	95.0	102.1	67.1	62.9	72.7
Unverifiable,inline	67.6	25.4	80.1	80.5	99.7	98.0	64.3	60.5	68.6

Table 5: Relative SINTL JVM backend performance among different backend configurations. Values are in percent relative to baseline code with no inlining and verifiable output.

Another interesting aspect of the SINTL JVM backend is the ability to produce both verifiable and unverifiable JVM output. Due to limitations of the JVM verifier discussed above, the verifier rejects the compiled code when type inference proves that type checks are unnecessary. Tables 7 and 7 shows the runtimes of the Gabriel benchmarks with different compiler and verification options.

Graphical representations of this information are available at the end of this document.

8 Conclusion

SINTL and the PM compiler acheive an order of magnitude performance improvement over existing Scheme compilers to the JVM through intelligent and powerful program analyses. By separating the front end compiler from the SINTL assembler, the PM compiler can more easily support a diverse set of backend targets. With a declarative type system and strongly-typed instructions and primitives, SINTL can both ensure the type safety and satify the instruction type constraints imposed by the SINTL language and primitives. Using the reg→stack algorithm to effectively convert the SINTL register transfer language into a stack machine language, we implemented the JVM-SINTL backend to the Java Virtual Machine. This backend uses natural JVM representations for most Scheme data types, and natively supports the **case-lambda** syntax through clever

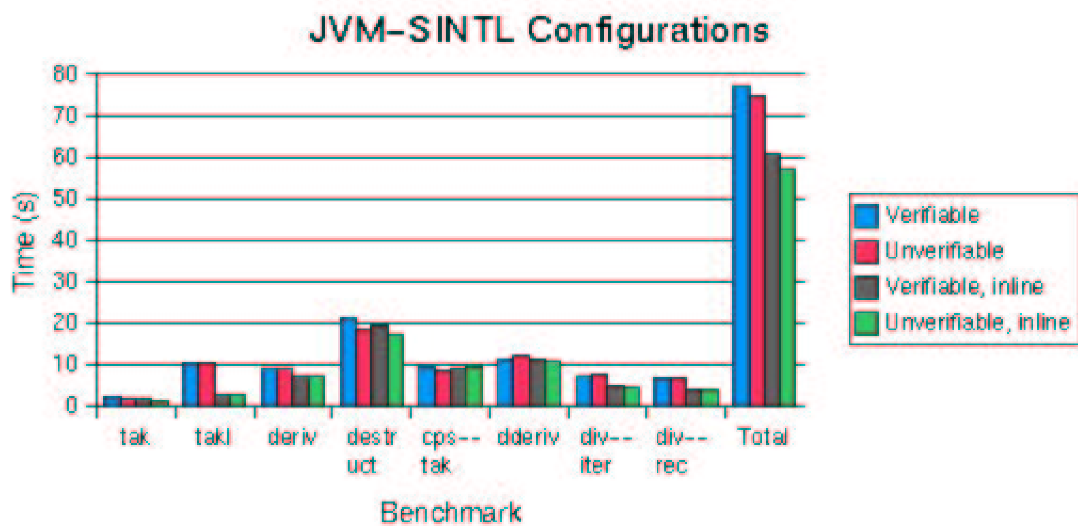

Figure 13: Performance results for different PM compiler configurations for JVM-SINTL backend.

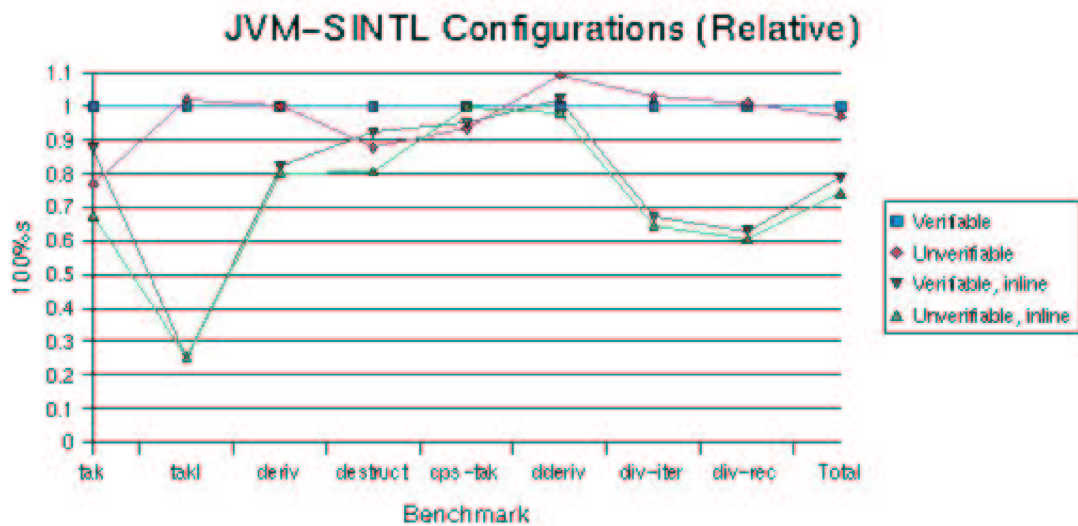

Figure 14: Relative performance results for different PM compiler configurations for JVM-SINTL backend.

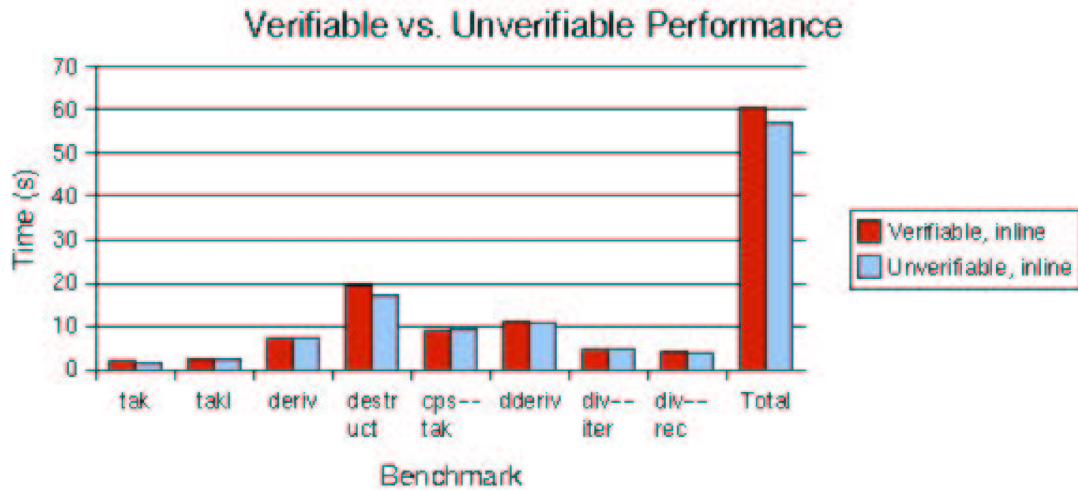

Figure 15: Performance results for verifiable and unverifiable JVM byte-codes for JVM-SINTL backend.

use of the JVM virtual table dispatch. With a runtime system written almost entirely in Scheme (using less than 1000 lines of handwritten Scheme specific Java code), the JVM backend acheives an order of magnitude performance improvement over existing Scheme to Java compilers and comparable performance to Scheme specific virtual machines.

References

[1] J.D. Ullman A.V. Aho, R. Sethi. *Compilers, Principles, Techniques, and Tools.* Addison Wesley, 1988.

[2] Per Bothner. Kawa - compiling dynamic languages to the java vm. In *Usenix Conference*, New Orleans, June 1998.

[3] Per Bothner. Kawa: Compiling scheme to java. In *Lisp Users Conference*, Berkeley, CA, November 1998.

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

[5] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: A pedagogic programming environment for scheme. In Hugh Glaser, Pieter Hartel, and Herbert Kuchen, editors, *Programming*

Languages: Implementations, Logics, and Programs, volume 1292 of *LNCS*, pages 369–388, Southampton, UK, September 1997. Springer.

[6] Richard Gabriel. *Performance Evaluation of Lisp Systems.* ???, 1000.

[7] W. Internet, I. pages, Association, t Advancement, C. in, and E. George. Mlrisc: Customizable and reusable code generators, 1996.

[8] S. Jagannathan, S. Weeks, and A. Wright. Type-directed flow analysis for typed intermediate languages, 1997.

[9] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C–: a portable assembly language that supports garbage collection.

[10] David Walsh-Kemmis Kevin Hammond. Juaskell: Implementing evaluation strategies in java.

[11] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachussetts, 1990.

[12] G. Russell N. Benton, A. Kennedy. Compiling standard ml to java bytecodes. In *ICFP '98*, Baltimore, MD, September 1998.

[13] J. Rees and R. Kelsey. A tractable scheme implementation, 1995.

[14] M. Serrano and P. Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In *SAS 95*, pages 366–381, 1995.

[15] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. Technical Report 93-218, Rice University, December 1993. Revised version in: *Proc. 1994 ACM Conference on Lisp and Functional Programming.*