A Scheme-to-Java Translator with Soft Typing

Akihito Nagata Eijiro Sumii Akinori Yonezawa

Department of Computer Science,
Graduate School of Information Science and Technology,
University of Tokyo

{ganat,sumii,yonezawa}@yl.is.s.u-tokyo.ac.jp

ABSTRACT

This paper presents a compiler that translates Scheme code into the Java language. The programming language Scheme is known to be highly *expressive* and *safe*. By translating Scheme into Java, one can benefit from such features of Scheme as well as the portability of Java. Our compiler implements dynamic type checking in Scheme via down-casting in Java by mapping each Scheme value to a Java `Object`. However, naively doing so requires dynamic class checking in Java for every primitive operation in Scheme and thereby incurs much overhead. We reduce this overhead of runtime type checking by exploiting partial static type information obtained through *soft typing*. This method achieves about 200–700% speedup in micro benchmarks and 67% for a ray tracing program. Thus, our compiler combines the expressiveness and safety of Scheme with the portability of Java without much loss of efficiency.

1. INTRODUCTION

In today's computing environment, where a large variety of operating systems and processors are available, it is an important but hard task to support as many platforms as possible when developing a compiler of a general-purpose language. For this reason, several compilers such as Kawa [3], SILK [1], and MLj [2] use the Java Virtual Machine (JVM) [7] as their target platform in order to benefit from its wide availability and other selling points such as standard libraries, garbage collection, just-in-time (JIT) compilers, and bytecode verification.

However, these benefits do not come for free: since JVM is primarily designed for compiling Java programs, there are non-trivial obstacles to compiling other languages onto JVM. A major instance of such obstacles is difficulty in implementing proper tail-call optimization [12]. Another is the static type system of JVM, which is based on declarative (as opposed to structural) subtyping as found in many object-oriented languages. Although it plays an essential role in the bytecode verification of JVM, the system is not necessarily expressive enough to efficiently encode other styles of typing such as parametric polymorphism in ML and dynamic typing[1] in Scheme.

For example, consider the translation of the following Scheme code into JVM. For the sake of readability, we use the surface syntax of Java for presentation. Its difference from the bytecode is insignificant in this paper.

```
(+ (+ 1 2) (if some-complex-condition 3 "abc"))
```

If we faithfully (and naively) follow the semantics of Scheme, the translated Java code would require many casts—both upward and downward—as follows.

```
((Integer)(Object)new Integer(1)).intValue() +
((Integer)(Object)new Integer(2)).intValue() +
((Integer)(some_complex_condition ?
           (Object)new Integer(3) :
           (Object)new String("abc"))).intValue();
```

Note that both directions of casts incur various overheads here: up-casts from `int` to `Object` require memory allocation of `Integer` objects, while down-casts from `Object` to `Integer` require dynamic class checking as well as method invocation and pointer dereference in `intValue()`.

However, many of those casts are obviously redundant. Specifically, the first addition `(+ 1 2)` should not require any dynamic type checking at all because the types of the arguments are apparent. Taking this into account, the above translation can be optimized as follows.

```
1 +
2 +
((Integer)(some_complex_condition ?
           (Object)new Integer(3) :
           (Object)new String("abc"))).intValue();
```

Note that the remaining casts are still necessary because the type of this conditional expression as a whole cannot be statically determined. (In this particular case, it is also possible to avoid dynamic checking by transforming the original program like

```
(if some-complex-condition
    (+ (+ 1 2) 3)
    (+ (+ 1 2) "abc"))
```

[1]Some claim that there is actually no such thing as "dynamic typing" [10, page 2], but we follow the standard terminology throughout this paper.

or, with the so-caled *let-insertion* method for avoiding code duplication,

```
(let ([x (+ 1 2)])
  (if some-complex-condition (+ x 3) (+ x "abc")))
```

but doing so in general incurs explosion or even divergence of code size under the presence of recursion.)

How can we obtain such partial static type information in general? The answer is *soft typing* [4, 16]: it tells us the type of each expression as far as it can be determined statically, in which case the tagging (through up-casts) and type checking (through down-casts) can be eliminated at translation time; otherwise, they are left to the runtime.

The purpose of our work in this paper is to investigate how much speedup can be achieved by using soft typing in the translation of Scheme into Java. It may seem obvious from the above discussion that at least *some* speedup will be obtained, but *how much* is not so clear, in particular because of the sophisticated optimizations performed by JIT compilers embedded in today's JVMs. For this purpose, we have implemented a Scheme-to-Java translator with soft typing and carried out experiments about a few tiny programs as well as one more realistic application: ray tracing. The results were encouraging despite the simplicity of our implementation.

The rest of this paper is structured as follows. Section 2 defines the source language of our translation. Section3 introduces the basic way to translate Scheme into the Java language. Section 4 explains the soft typing system for Scheme and how we use it to optimize our translation. Section 5 reports the result of experiments. Section 6 discusses related work and Section 7 concludes with future work.

2. CORE SCHEME: OUR SOURCE LANGUAGE

For simplicity, we focus on a minimal subset of Scheme, which we call *Core Scheme*, as the source language of our translation. This language embodies only the essence of Scheme, but many of the other features can be implemented in much the same way as this essence. (See Section 7 for other mechanisms that require extra treatment.)

The syntax of Core Scheme is defined as follows, where e is the meta-variable for expressions.

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & (\lambda\ (x)\ e) \\
& | & (\mathbf{ap}\ e_1\ \ldots\ e_n) \\
& | & (\mathbf{if}\ e_1\ e_2\ e_3) \\
& | & (\mathbf{let}\ ([x\ e_1])\ e_2) \\
& | & (\mathbf{cons}\ e_1\ e_2) \\
& | & (\mathbf{car}\ e) \\
& | & (\mathbf{cdr}\ e)
\end{array}
$$

x is the meta-variable for identifiers and c for constants, which include primitive values (numbers, $\#t$, $\#f$, $'()$, etc.) as well as their operators (+, null?, etc.). $(\lambda\ (x)\ e)$ is a λ-abstraction and $(\mathbf{ap}\ e_1\ \ldots\ e_n)$ is a procedure application. $(\mathbf{if}\ e_1\ e_2\ e_3)$ is a conditional expression. $(\mathbf{let}\ ([x\ e_1])\ e_2)$ is a local variable definition. $(\mathbf{cons}\ e_1\ e_2)$ is a pair constructor, where $(\mathbf{car}\ e_1)$ and $(\mathbf{cdr}\ e_1)$ are pair projections. As usual, identifiers are bound by λ and **let**-forms.

```
public class ScmObj {
   public boolean evalBool() { return true; }
}
public class ScmNum extends ScmObj {
   double d; (* exactness issues put aside *)
   public ScmNum(double d) { this.d = d; }
   public final double evalNum() { return d; }
}
public class ScmProc extends ScmObj {
   public ScmObj app(ScmObj arg)
     { return null; (* dummy body *) }
}
```

Figure 1: Classes to represent Scheme values as Java objects (excerpt)

3. TRANSLATION WITHOUT SOFT TYPING

In this section, we explain our translation *without* soft typing in order to present the way to implement basic features of Scheme, in particular dynamic typing and first-class procedures, which are anyway necessary.

3.1 Representation of Scheme Values

First, we consider how to represent Scheme values in the Java language.

Unlike statically typed languages, dynamically typed languages accept evidently invalid code such as (+ 1 #f) at compiling time, and signal an error only at execution time when actually executing the code. To support this behavior, each value must carry its own type information. Since Java does not allow tagging of values by direct access to memory, we represent this type information by exploiting the class system of Java.

Specifically, we provide a collection of Java classes for representing Scheme values, as in Figure 1. `ScmObj` is the base class of all classes in this package. All constant values and all the results of evaluation are represented by objects in this class or classes inherited from this class. Thus, every object is regarded as an instance of `ScmObj` at compiling time by the class system of Java. When the actual value of an object is needed, it is down-cast to a subclass according to the required type, and then the corresponding evaluation method (such as `evalNum`) is called. The class checking in Java during this down-casting serves as dynamic type checking in Scheme. If it succeeds, the evaluation method returns the actual value as expected. Otherwise, an exception will be raised (`ClassCastException`, to be specific).

3.2 Implementation of First-Class Procedures

Next, implementation of Scheme procedures has to be considered. They are of course a kind of Scheme values, but

```
(define (g x)
   (define (f y) (+ x y))
   f)
```

Figure 2: Scheme code example

demand more consideration because Scheme procedures are first-class while Java methods are not. To bridge this gap, we represent each procedure as an object of a class with a method `app`, which actually contains the body of the procedure. This suffices because objects are first-class in Java. Each of such classes is defined to be a subclass of `ScmProc`, which is again a subclass of `ScmObj` because procedures are also values in Scheme. Thus, when an instance of `ScmObj` is applied as a procedure, it is down-cast to `ScmProc` and then the method `app` is invoked.

3.3 Allocation of Free Variables in First-Class Procedures

When translating Scheme procedures to Java classes as explained above, where to allocate variables must be considered. That is, whether a Scheme variable should be translated into a method local variable or an instance variable, for instance.

To consider this issue, see Figure 2 for example. The procedure `f` is defined inside of `g`. The procedure `g` must be able to refer to `x` and `f`, and `f` must be able to refer to `y` and `x`. In our approach, `f` and `y` are translated to local variables of the `app` methods of `g` and `f`, respectively. On the other hand, `x` is translated to an instance variable of the class representing `g`, while the class representing `f` is given an instance of that class, so that it is able to refer to `x`. Thus, this example is translated as follows.

```
class G extends ScmProc {
   ScmObj x;
   ScmObj app(ScmObj x) {
      this.x = x;
      F f = new F(this);
      return f;
   }
}
class F extends ScmProc {
   G g;
   F(G g) { this.g = g; }
   ScmObj app(ScmObj y) {
     return new ScmNum(((ScmNum)g.x).evalNum() +
                       ((ScmNum)y).evalNum());
   }
}
```

In general, variables which may be referred-to by other procedures must be allocated as instance variables, while other variables can be defined as local variables in the `app` method. (This way of implementing nested scoping by passing an outer environment to its inner definition is similar to the standard implementation [7] of inner classes in Java.) Furthermore, when a procedure having instance variables is applied, it must be cloned so that the instance variables are allocated in a different space. (This corresponds to allocation of stack frames by procedure calls in a naive implementation of Scheme.)

Readers who are familiar with functional languages may wonder why `f` is given the procedure object `g` itself, rather than the value of `x` only. Passing `g` is necessary, however, for letting `f` share the value of `x` with `g`. This difference can be observed in programs using destructive updates, like:

```
(define (g x)
   (define (f y) (+ x y))
   (set! x 3)
   f)
```

If `f` and `g` does not share the same value of `x`, the evaluation of `((g 1) 2)` gives an incorrect result `3` instead of the correct one `5`.

3.4 Putting These Altogether

On the basis of the above ideas, the overall translation of Core Scheme expressions into Java programs can be given as follows.

- A λ-expression is translated into the definition of a procedure class and its instantiation. As explained above, the constructor of this procedure class takes the representations of outer procedures as its arguments in order to refer to their instance variables. Conversely, formal arguments of the λ-expression are translated either to instance variables of this procedure class if they are referred-to from inner procedures, or else to local variables of its `app` method.
- An **ap**-form is translated into down-casting of the function part to `ScmProc`, followed by an invocation of its `app` method. In addition, if this procedure object may have instance variables, it must be `clone`-d before applied.
- **if**-forms are translated into `if` statements in Java, along with down-casting of the condition part to `ScmBool` and an invocation of its `evalBool` method.
- **let**-forms are translated into assignments in Java. If the variable is referred-to by an inner procedure, it is translated to an instance variable of the current procedure class; otherwise, it is translated as a local variable.
- Finally, the resulting value of each procedure is `return`-ed at the end of the corresponding `app` method.

4. OPTIMIZATION WITH SOFT TYPING

Although the above translation of Core Scheme into Java works fine as far as only correctness is concerned, it suffers from large overhead because of up-casts and down-casts for dynamic type checking as discussed in the Introduction. In this section, we present a soft type system to reduce this overhead.

4.1 Types

The syntax of types and type schemes in our system is given in Figure 3. As for types, we have primitive types for numbers, booleans, and $'()$ in addition to $\rightarrow$ as a standard type constructor for functions and $\times$ for pairs. As for type schemes, we have union types and recursive types for the purpose of soft typing.

$$
\begin{array}{rcll}
\rho & ::= & \mathit{number} & \\
 & | & \mathit{boolean} & \\
 & | & \mathit{null} & \\
 & | & \sigma_1 \to \sigma_2 & \\
 & | & \sigma_1 \times \sigma_2 & \\
\tau & ::= & \emptyset & \\
 & | & \rho \cup \tau & \\
\sigma & ::= & \mu\alpha_1.\ldots.\mu\alpha_n.\tau & (n \geq 0) \\
 & | & \alpha &
\end{array}
$$

Figure 3: Types and type schemes

Consider, for example, a Scheme expression (**if** x 1 #t). Since neither *number* nor *boolean* (any single type, in fact) suffices as the type of this expression, we adopt a union type scheme $\mathit{number} \cup \mathit{boolean}$. To simplify the typing rules and the type inference algorithm in later sections, we impose a restriction that each type ρ_i in a union type scheme $\rho_1 \cup \cdots \cup \rho_n$ consists of a different type constructor. That is, we do not allow such type schemes as $(\sigma_1 \to \sigma_2) \cup (\sigma_3 \to \sigma_4)$ or $(\sigma_1 \times \sigma_2) \cup (\sigma_3 \times \sigma_4)$.

For another example, take (λ (f) (**ap** f f)). Since the argument of the function f is itself, its type α must satisfy an equation of the form $\alpha = \alpha \to \tau$, but again there exists no such type. Thus, we introduce a recursive type scheme $\mu\alpha.\, \alpha \to \tau$ to denote a solution of this equation.

We assume standard equivalence for union types and recursive types and use them implicitly when necessary. For the sake of convenience, we sometimes refer to type schemes just as "types" in the rest of this paper when their distinction is not important.

Note that we do not have polymorphic types. If we did, we could improve the precision of our analysis as in other soft type systems [4, 16], but we cannot do so (in a straightforward manner, at least) because the present type system of Java does not have parametric polymorphism. See also Section 7 for a relevant argument.

4.2 Typing Rules

We define the discipline which our soft type system must follow in the form of inductive rules. They are given in Figure 4. A type judgment $\Gamma \vdash e : \sigma$ claims that the expression e has the type scheme σ under the type environment Γ, where a type environment is a mapping from identifiers to type schemes and assigns type schemes to free variables in an expression. $\Gamma[x \mapsto \sigma]$ denotes the extension of Γ with x being mapped to σ. The meta-level operator *TypeOf* gives the types of constants in Core Scheme. For example, $\mathit{TypeOf}(1) = \mathit{number}$ and $\mathit{TypeOf}(\textbf{positive?}) = \mathit{number} \cup \sigma_1 \to \mathit{boolean} \cup \sigma_2$ for any σ_1 and σ_2.

4.3 Type Inference

Now we explain how to automatically determine the types of expressions in a Scheme program under our type system. Unlike in explicitly (and statically) typed languages, this is essential in Scheme because such type information is hardly given by programmers.

$$\frac{}{\Gamma \vdash c : \mathit{TypeOf}(c) \cup \tau}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma[x \mapsto \sigma_1] \vdash e : \sigma_2}{\Gamma \vdash (\lambda\ (x)\ e) : (\sigma_1 \to \sigma_2) \cup \tau}$$

$$\frac{\Gamma \vdash e_1 : (\sigma_2 \to \sigma_1) \cup \tau \quad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash (\mathbf{ap}\ e_1\ e_2) : \sigma_1}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2 \quad \Gamma \vdash e_3 : \sigma_2}{\Gamma \vdash (\mathbf{if}\ e_1\ e_2\ e_3) : \sigma_2}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma[x \mapsto \sigma_1] \vdash e_2 : \sigma_2}{\Gamma \vdash (\mathbf{let}\ ([x\ e_1])\ e_2) : \sigma_2}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash (\mathbf{cons}\ e_1\ e_2) : (\sigma_1 \times \sigma_2) \cup \tau}$$

$$\frac{\Gamma \vdash e : (\sigma_1 \times \sigma_2) \cup \tau}{\Gamma \vdash (\mathbf{car}\ e) : \sigma_1}$$

$$\frac{\Gamma \vdash e : (\sigma_1 \times \sigma_2) \cup \tau}{\Gamma \vdash (\mathbf{cdr}\ e) : \sigma_2}$$

Figure 4: Typing rules

The algorithm for inferring the type of an expression e can roughly be given as follows.

- If e is a constant c, its type is given by using the *TypeOf* function as $\mathit{TypeOf}(c) \cup \alpha$, where α is a fresh type variable.
- If e is a variable x, its type is looked up in the type environment Γ as $\Gamma(x)$.
- If e is a λ-abstraction (λ (x) e_1), a fresh type variable α is assigned to x and the type of e_1 is inferred under the extended type environment $\Gamma[x \mapsto \alpha]$ to be, say, σ. Then, the type of e is inferred to be $(\alpha \to \sigma) \cup \beta$ for another fresh type variable β.
- If e is a function application (**ap** e_1 e_2), the types of e_1 and e_2 are first inferred to be, say, σ_1 and σ_2. Then, σ_1 and $(\sigma_2 \to \alpha) \cup \beta$ are *unified* where α and β are fresh type variables. The method of unification is explained later. The type of e is given by this α.
- If e is a conditional expression (**if** e_1 e_2 e_3), the types of e_1, e_2 and e_3 are inferred and the types of e_2 and e_3 are unified. Then, the type of e is inferred to be the same type as e_2 and e_3.
- If e is a **let** binding (**let** (x e_1) e_2), the type of e_1 is inferred to be, say, σ_1 and the type of e_2 is inferred under the extended type environment $\Gamma[x \mapsto \sigma_1]$. The type of e is then inferred to be the same as that of e_2.

- The cases for a pair construction and pair projections are defined in the same manner as the above cases.

Our unification algorithm is standard (see [10, pp. 326–329], for example) except for the presence of union type schemes. In standard algorithms, a type variable can be implemented as a reference to a possibly empty type—that is, an element of "`type option ref`" in an ML-like notation—which enables unification via assignments. In our algorithm, by contrast, a union type scheme $\rho_1 \cup \cdots \cup \rho_n$ is represented by a mutable record of type:

```
{ number : bool, boolean : bool, null : bool,
  procedure : (type * type) option,
  pair : (type * type) option }
```

Each field of this record denotes whether the corresponding type is included in the above union type scheme. For example,

```
{ number = true, boolean = false, null = true,
  procedure = None, pair = None }
```

represents *number* $\cup$ *null* and

```
{ number = false, boolean = true, null = false,
  procedure = Some(t1, t2), pair = None }
```

represents *boolean* $\cup$ ($\texttt{t1} \rightarrow \texttt{t2}$). Thus, in order to unify two union type schemes, it suffices to mutate fields of the two records in an increasing way (that is, from `false` to `true` or `None` to `Some`) so that they become equal.

After all of these, the remaining free type variables are assigned the empty type $\emptyset$ in order to eliminate ambiguity.

4.4 Optimization with the Type Information

Now that all (syntactically valid) expressions of Core Scheme are given static types, the translation into Java can be optimized by exploiting the type information. Specifically, if the type of an expression is *not* eqiuvalent to a union type, it is unambiguous and therefore the expression's value need not be up-cast into `ScmObj` or be down-cast when used. Accordingly, the interfaces of Java classes to represent Scheme procedures are no longer uniform, because the types of their arguments and return values may be different from `ScmObj`.

Note that we have carefully designed the typing rules so that (1) up-casts may be necessary only when values are produced (λ-abstractions, pair constructions, etc.) and (2) down-casts may be necessary only when values are consumed (function applications, pair projections, etc.). If we introduced a more general subtyping rule, the corresponding coercions between Java objects would be far more difficult (if possible at all) and incur another overhead itself.

5. RESULT OF EXPERIMENTS

In order to measure the degree of improvement by the soft typing optimization, we compare the execution time of Scheme programs under the following three conditions. First, we execute the original programs in Chez Scheme [5]. Second, we translate the same programs into Java without soft typing and executed them on JVM. Last, we translate them with soft-typing.

	Chez [msec]	Without soft typing [msec]	With soft typing [msec]
QSort	430	6024	1774
Fib30	1820	8600	1093
RayTrace	11510	39728	23731

Table 1: Execution time of benchmark programs

Original Scheme code:

```
(w3 (/ (* (atan w5) 30.0) 3.1415927))
```

Translation into Java code with soft typing:

```
double pv682;
pv682 = Math.atan(this.v105);
pv681 = pv682 * 30;
v107 = pv681 / 3.1415927;
```

Figure 5: Fragment of the ray tracing program

The execution environment is Solaris 2.6, Sun Ultra Enterprise 4000, UltraSPARC 168MHz processors, 1.2GB main memory, Solaris Java VM version 1.2.1, and Chez Scheme version 6.0a. The Scheme programs are Quick Sort, Fibonacci Numbers, and Ray Tracing. For writing these programs, our implementation supports a subset of Scheme which is a little larger than Core Scheme. In particular, it supports procedures with multiple arguments.

Table 1 shows the results of this experiment. Each result reflects characteristics of the corresponding program as follows.

Quick Sort. This program sorts a list of numbers by the quick-sorting algorithm. Its main part consists of number comparisons and pair manipulations. The type of "car" parts of the list is inferred to be *number*, while that of "cdr" parts is inferred to be (*number list*) $\cup$ *null*, which is (strictly speaking) an ambiguous type. However, since we implement $'()$ in Scheme by the `null` reference in Java, this does not require dynamic type checking—instead, JVM detects null references—and is not an actual problem. Thus, the optimization worked well.

Fibonacci Numbers. Computes *fib*(30). In this program, most of the operations are arithmetic ones (addition and subtraction). Our soft type inference algorithm statically detected those *number* types and therefore the optimization performed well.

Ray Tracing. This program also mainly performs numeric operations, but extensively uses lists and vectors like structures in the C language. Consequently, many elements of lists and vectors are treated as `ScmObj` because their types are unified during the typing of functions manipulating those elements. Nevertheless, other parts of the program are given more precise static types and optimized accordingly (as in Figure 5, for example).

Comparison with Chez Scheme is more difficult because we have little access to its internals, but we believe that our results are not too bad considering the disadvantages of JVM and its JIT compilation against offline native-code compilers.

6. RELATED WORK

Bothner [3] provided a set of Java classes, named Kawa, for implementing dynamic languages. On top of it, he also built an implementation of near-R^5RS Scheme. It supports a large variety of Scheme features including number types, multiple values, continuations, and proper tail recursion. Since Kawa aims to be a generic class library for compiling functional languages to Java bytecode, it—unlike our soft typing optimization—adopts a uniform value representation with dynamic type checking in preference of simplicity over efficiency.

Anderson, Hickey and Norvig have developed SILK [1], an implementation of Scheme in Java covering a large subset of R^4RS Scheme. A major technical point of SILK is interoperation between Scheme and Java: in SILK, Scheme code has access to Java classes and methods as well as Java programs can call Scheme procedures. SILK does not have a soft type system, either. Combining soft typing with SILK-like interoperation of Scheme and Java would require non-trivial integration of their type systems.

Oiwa [17] proposed to extend the specification of JVM in order to improve the performance of dynamic type checking by adding a new type called *descriptors*, which is a supertype of both `int` and `Object`. By using descriptors, dynamic typing can be implemented without constructing an alternative type hierarchy besides the native one. This approach achieves higher efficiency but is less portable because it requires modification of JVM.

7. FUTURE WORK

7.1 Polymorphism and Code Duplication

In our soft type system, we did not introduce parametric polymorphism because it does not immediately help optimizing the translation into Java, whose type system does not have parametric polymorphism anyway. However, if we *duplicate* a type-parametric procedure for different type instantiations—as the MLj compiler does [2]—the precision of type inference may possibly improve and so may the efficiency of the resulting translation. It remains to see whether or when doing so pays for its cost.

7.2 Tail-Call Optimization

The Revised[5] Report on the Algotithmic Language Scheme [8] requires proper implementation of tail recursion. However, it usually needs low-level stack manipulations, which JVM does not allow. Nevertheless, there are several approaches to implementing tail recursion or, more generally, tail call optimization within the framework of Java.

Kawa [3] and MLj [2] partially implement proper tail recursion by depending on the fact that many tail calls are actually self calls and replacing such calls with simple `goto`'s. This works as far as only self tail-calls are concerned.

Another approach is to use trampolines [15]. In this approach, procedures are divided into two kinds, tail-call procedures and the others. A tail-call procedure returns the closure of its continuation, which is then invoked by the outer procedure. This approach prevents stack growth but incurs a constant overhead in each tail call. Schinz and Odersky [12] reduces this overhead by counting the number of tail calls and use trampolines only once in a while.

Although our aim in this work has been to investigate the effect of soft typing in the translation from Scheme to Java, it should also be possible to modify our translator and implement some of these approaches.

7.3 First-Class Continuations

The Revised[5] Report also requires implementation of first-class continuations. It is often realized by copying stack frames to a garbage-collected heap, but this is of course not possible on JVM. Transforming the whole program into CPS is a straightforward way to implement first-class continuations, but doing so would incur another overhead in our situation.

Kawa [3] partially implemented first-class continuations by using exception handling. Invocations of `call-with-current-continuations` are translated to `try` statements, while applications of continuations are translated to `throw` statements. This approach does not always work correctly, however—to be specific, it fails when a continuation escapes out of its static scope for the obvious reason.

Another approach is to represent continuations by suspended threads [9, 6]. This method works only when the continuations are *one-shot* (that is, applied only once) because threads are linear (in the sense of linear logic) by nature. To remove this restriction, some primitive for duplicating a thread would be necessary.

To the best of our knowledge, the only complete method of implementing first-class continuations on Java (without modifying JVM or incurring much overhead) is the one based on sophisticated global source-to-source transformation, which makes use of exceptions to save and restore the execution state of a running program [14]. This technique is applicable not only to the surface language of Java [13] but also to its bytecode [11].

Again, although it is has not been the goal of this work to implement all the features of Scheme, it should be possible to implement first-class continuations by adopting some of the above methods in our translation.

8. REFERENCES

[1] K. R. Anderson, T. J. Hickey, and P. Norvig. SILK: a playful blend of Scheme and Java. In *Scheme and Functional Programming 2000*, pages 13–22, September 2000.

[2] N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 129–140, 1999.

[3] P. Bothner. Kawa—compiling dynamic languages to the Java VM. In *Proceedings of the USENIX 1998 Technical Conference, FREENIX Track*, New Orleans, LA, June 1998. USENIX Association.

[4] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*, volume 26, pages 278–292, 1991.

[5] Chez Scheme. http://www.scheme.com/.

[6] M. Feeley. A portable implementation of first-class continuations for unrestricted interoperability with C in a multithreaded Scheme. September 2000.

[7] G. S. James Gosling, Bill Joy and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, 2000.

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

[9] S. Kumar, C. Bruggeman, and R. K. Dybvig. Threads yield continuations. *Lisp and Symbolic Computation*, 10(3):223–236, 1998.

[10] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[11] T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode transformation for portable thread migration in Java. In *Proceedings of Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents (ASA/MA 2000)*, volume 1882 of *Springer LNCS*, pages 16–28, September 2000.

[12] M. Schinz and M. Odersky. Tail call elimination on the Java virtual machine. In *Proc. ACM SIGPLAN BABEL'01 Workshop on Multi-Language Infrastructure and Interoperability*, volume 59 of *Electronic Notes in Theoretical Computer Science*, pages 155–168. Elsevier, 2001. http://www.elsevier.nl/locate/entcs/volume59.html.

[13] T. Sekiguchi, H. Masuhara, and A. Yonezawa. A simple extension of Java language for controllable transparent migration and its portable implementation. In *Coordination Models and Languages*, volume 1594 of *LNCS*, pages 211–226, 1999.

[14] T. Sekiguchi, T. Sakamoto, and A. Yonezawa. Portable implementation of continuation operators in imperative languages by exception handling. In *Advances in Exception Handling Techniques*, volume 2022 of *LNCS*, pages 217–233, 2000.

[15] D. Tarditi, P. Lee, and A. Acharya. No assembly required: Compiling standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, June 1992.

[16] A. K. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems*, 19(1):87–152, January 1997.

[17] O. Yutaka, K. Taura, and A. Yonezawa. Extending Java virtual machine with integer-reference conversion. *Concurrency - Practice and Experience*, 12(6):407–422, 2000.