A Scheme-to-Java Translator with Soft Typing

Soft Typingを利用した Scheme to Java トランスレータ

Akihito Nagata　永田 章人　(学籍番号 01030)

A Senior Thesis　卒業論文

A Scheme-to-Java Translator with Soft Typing
Soft Typing を利用した Scheme to Java トランスレータ
Akihito Nagata
永田 章人 (01030)

A Scheme-to-Java Translator with Soft Typing
Soft Typing を利用した Scheme to Java トランスレータ
Akihito Nagata
永田 章人 (01030)

A SCHEME-TO-JAVA TRANSLATOR WITH SOFT TYPING

Soft Typingを利用した Scheme to Java トランスレータ

by

Akihito Nagata
永田 章人

A Senior Thesis
卒業論文

Submitted to
the Department of Information Science
the Faculty of Science
the University of Tokyo
on February 5, 2002
in Partial Fulfillment of the Requirements
for the Degree of Bachelor of Science

Thesis Supervisor: Akinori Yonezawa　米澤 明憲
Professor of Information Science

ABSTRACT

This thesis 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 local procedure definitions of Scheme by translating Scheme procedures into Java inner classes. It also implements dynamic type checking of Scheme with down-casting of Java by translating each Scheme data into 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 2 times speedup for *Fibonacci numbers* and 1.2 times for *quick sort*. Thus, our compiler combines the expressiveness and safety of Scheme with the portability of Java without much loss of efficiency.

論文要旨

この論文では Scheme で書かれたコードを Java 言語に変換するコンパイラを作る。

プログラミング言語 Scheme は高い表現力と安全性を持っており、Java 言語へと変換することによりこれらの Scheme の性質に加え、Java のもつポータビリティも利用することができる。Scheme の関数を Java の class に変換することにより、Scheme の局所関数定義を実装し、Scheme の各データと Java の Object を対応させ、それをダウンキャストすることにより、動的な型検査を実装した。しかし単純に実装した場合、プリミティブ関数を適用する度に Java の class の検査を行う必要があり、オーバーヘッドがかかる。ここでは soft typing により得られる情報を用いて、静的な型情報を部分的に用いることにより、実行時の型検査を軽減させた。この方法によりフィボナッチ数列の場合は 2 倍、クイックソートの場合は 1.2 倍、計算速度を向上させることができた。このようにして、Scheme の持つ表現力と安全性、Java の持つポータビリティと効率を大きく損なうことなくコンパイルすることができた。

Acknowledgements

We are grateful to our thesis supervisor professor Akinori Yonezawa. We are also grateful to Eijiro Sumii for his many helpful advices and for checking our paper. We would like to thank Yutaka Oiwa for his senior thesis which our thesis is modeled after. We also would like to thank all members of Yonezawa Laboratory for valuable daily discussions.

However, the responsibility for any errors is entirely mine.

Contents

1 Introduction **1**

1.1 Translating Scheme into Java 1

1.2 Optimizing the Translation 2

1.3 Outline of This Paper 4

2 Core Scheme: Our Source Language **5**

3 Translation without Soft Typing Optimization **6**

3.1 Translation of Scheme Value to Java Object 6

3.1.1 Characteristic Features of Scheme Value 6

3.1.2 Java Package for Representing Scheme Objects 7

3.2 Translation of Scheme Procedures to Java Codes 10

3.3 Example 13

4 Optimization with Soft Typing **15**

4.1 Our Static Types Definition 16

4.2 Static Type Discipline 17

4.3 Type Inference Algorithm 18

4.4 Optimization by using Type Informations 19

5 Result of Experiments **23**

6 Evaluation **27**

7 Related Work **29**

8 Future Work **31**

8.1 Duplication of Polymorphic Procedures 31

8.2 Proper Tail Recursion 32

List of Tables

5.1 Measured Execution Time 23

List of Figures

3.1 Package for Representation of Scheme Objects in Java 9
3.2 Source of Sample.scm 13
3.3 Source of Sample.java 14

4.1 Scheme Type Domain 16
4.2 Static Typing Discipline 17
4.3 Redefinition of ScmPair 21

5.1 Measured Execution Time 24

Chapter 1

Introduction

1.1 Translating Scheme into Java

In recent years, along with the popularization of computers, there are several operating systems, such as Windows, Linux, Macintosh, in addition to a large variety of CPU. Since a software which target one platform cannot be used on the other, if one wants to use a software on several platforms, this software has to be rebuilt for each platforms. These are big efforts. To solve this, JVM is developed. JVM allows programs to be distributed easily, since the programs written in Java bytecodes can be executed platform-independently. Thanks to the great success of Java, the bytecodes can run on a wide variety of architectures. These advantages are largely independent of this language, though Java bytecodes are supposed to be compiled from this languages. In addition to the portability, JVM is equipped with garbage-collection and can execute in secure by using class type system. Moreover, this is supported by a capable collection of standard library classes. Thus there are many compilers which compiles other language to Java bytecodes such as MLj[4] which compiles Standard ML to Java bytecodes.

We try to translate Scheme[9] into Java language. Scheme is one of the functional programing language invented by Guy Lewis Steele Jr. and Gerald Jay Sussman. It was designed to have an exceptionally clear and simple semantics and few different ways to form expressions. An important consequence of this simple, uniform representation is the susceptibility of Scheme programs and data to uniform treatment by other Scheme programs. Scheme is designed on the basis of lambda calculus to incorporate first class procedures. Scheme procedures are objects in their own right. They can be created dynamically, stored in data structures, returned as results of procedures, and so on. One distinguishing feature of Scheme is continuations, which has first-class status, too. Continuations are useful for implementing a wide variety of advanced control constructs. Scheme has dynamic typing system associating types with values, not with variables. Types are detected during program execution, not at compiling time. Thanks to this system, there are few restrictions imposed on the language. Thus Scheme can execute the programs safely which will be rejected by the other statically typed language like ML.

Thus Scheme-to-Java is much useful. Using the garbage-collection and leaving the portability and safety of Java, one can use the features of Scheme such as first class procedures. Moreover, if classes of Java can be treated and methods can be called from Scheme, class libraries and softwares written in Java is available in Scheme programs.

Kawa[5] is an implementation of near-R^{5}RS[9] Scheme executed on JVM. SILK[2] is also an implementation of Scheme in Java. It can deal with classes and call methods of Java in Scheme programs.

1.2 Optimizing the Translation

However, the translated programs have defects in efficiency.

Since Java bytecodes are supposed to be translated from Java language, implementations of Scheme features in Java produce much processing overhead. One of the cause of this problem is dynamic typing system of Scheme. As opposed to dynamic typing system of Scheme, type detecting of Java language is almost done at compile time, that is, this language is statically typed. Because Java language has much restrictions for safety as contrasted to machine codes which have no restrictions but have much risk, for implementation of dynamic typing system on Java language, there is no choice but to use the class system at the cost of efficiency. Implementation by this method leads many unnecessary instantiation of class and needless down-cast. We adopt soft typing[6] to eliminate some dynamic type checking operations by detecting the types of program expressions partially at compile time.

Since programmers can write programs with undefined applications such as addition of "1" and "true", a programming language must assign some form of meaning to these undefined applications. However in numerous cases, these expressions will cause risky operations or simply be bugs. To address and eliminate these problem, most programming languages adopt a type discipline. This discipline can either be static or dynamic. A statically typed language prevent the suspicious expressions at compiling time by imposing syntactic restrictions on programs. ML is a prominent example of a statically typed programming language.

In contrast, a dynamically typed language accept all programs. Undefined applications are detected during program executions by tests embedded in the each primitive operations. If arguments are not acceptable, the primitive operation generates a program exception. Scheme is a prominent example of a dynamically typed programming language.

A statically typed language has two important advantages over a dynamically typed.

Error Detection Using static typing, compiler can detect suspicious operations before execution.

Optimization Since compiler can extract the type informations, it can eliminate most runtime type checks and optimize each type representation. Thus it can create efficient code.

As opposed to this, a dynamically typed language also have advantages.

Expressiveness No static type checker can decide whether or not a program operate a invalid code. Therefore it should reject all programs which may execute invalid code. Then some of valid codes beyond the deductive power of static type checker will be rejected. Whereas, dynamically typed languages accepts all programs. Programmers don't need to mind type coherence.

Simplicity In order to extract type informations at compiling time, many restrictions are imposed to syntax and semantics.

Modularity In order to eliminate or optimize the type informations, this must be replaced with soft ware dependent representations. Contrary to this, type informations in the dynamically typed language are represented in the soft ware independent way.

Soft typing system combines the best features of both approaches. It can statically type check all programs written in a dynamically typed language and transform to type correct one by inserting explicit run-time type checker around the arguments of suspicious applications. Soft typing system retains the advantages of static typing because the transformed code conforms to a static type discipline. Therefore, this codes can be optimized by the compiler using type informations, but no programs will be rejected.

This paper reports implementations of soft typing to Scheme-to-Java and it's effect. But, on account of the type restrictions of Java, soft typing cannot be fully implemented, but partially it can be implemented.

The result indicates that all programs are optimized by this method, but it's degree is dependent on the form of programs. The program which take advantage of Scheme functions is not optimized well, because of the ambiguity of type. However, in common programs, programmers try to reduce ambiguous typed expressions for avoiding bugs. Thus in general our method is said to be work well to a certain degree in many cases.

1.3 Outline of This Paper

In chapter2, we define the target language for our translation. In chapter3, we introduce the basic way to translate Scheme into Java language. In chapter4, we explain the soft typing system for Scheme, and how our translation is optimized by using it. In chapter5, we report the result of experiments. In chapter6, we draw a conclusion from our work and evaluate. In chapter7, we introduce the related work and explain the relationship with our work. In the final chapter, we mention our future works.

Chapter 2

Core Scheme: Our Source Language

For simplicity, we confine target language to Core Scheme. This embodies only the essence of Scheme. Because, later implementation of another features is done in much the same way as these essentials.

Core Scheme is defined as follows. $e \in Exp$ is expressions of Scheme and $v \in Val$ is values.

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

Core Scheme has essence of Scheme, but omit the most part for simplicity. **ap** means procedure application. **if** is "if"-syntax and **let** is local variable definition.

Where $x \in Id$ are identifiers, and $c \in Const$ are constants. Const include both basic constants $(numbers, \#t, \#f,'() \in Basic \subset Const\ )$ and primitive operations $(p \in Prim \subset Const)$. Primitive operations are, for example, "+", "car" and so on. Identifiers are bound with λ and **let**-syntaxes.

Chapter 3

Translation without Soft Typing Optimization

This chapter introduces the main framework of translation Scheme to Java. To achieve translation, we must implement dynamic typing system, first-class procedure of Scheme and so on. We are concerned here only with basic translation, it means translation without any optimizations. Optimization using soft typing system is explained in next chapter.

3.1 Translation of Scheme Value to Java Object

3.1.1 Characteristic Features of Scheme Value

First, we consider the representation of Scheme Objects in Java Languages. Because, Scheme has many features which Java language doesn't have. We must take two features into consideration, which are *dynamic typing system* and *first-class procedure*.

Dynamic Typing System Scheme adopts dynamic typing system. Thus the expressions such as (+ 1 #f) raise an exception at execution time, because of type inconsistency. However, Java language adopts static typing system, which means most of type of expressions are decided before execution. For leaving the syntax simplicity and expressiveness of Scheme. The invalid expressions should not be detected at compiling time, but at execution time when the expressions are executed.

For implementation of dynamic typing, We have to add a type information to each object of Scheme. Because Java forbids direct memory access which means to address in memory space and read or write by the byte, the type informations of Scheme objects should be expressed by using the Java class type system which is the only way to access the memory space.

The functions needed for dynamic typing is:

- Primitive operators can deal with only a specific type object. When a primitive is applied at execution time, we have to confirm whether this operator can deal with the operand objects by checking the type of these. If it can, the primitive is applied to the operands But if it cannot, it raise exceptions. However, these type check are not done at compiling time at all, because Scheme can detect undefined application at execution time.
- The object of every type must be passed in the same way. For example, the procedure $(\lambda\ (x)\ x)$ can be applied to either boolean type or number type, even procedure type. In the same way, the procedure $(\lambda\ (x)\ (+\ x\ 1))$ can be applied to either boolean type or number type, but in case that it is applied to boolean type, it will raise exceptions at addition.

First-Class Procedure Scheme procedures are treated as first-class object, which means that they can be created dynamically, stored in data structure, and returned as result values of other procedure. However, methods of Java language cannot be treated like that. Thus, Scheme procedures cannot be simply represented by Java methods.

3.1.2 Java Package for Representing Scheme Objects

If we replace the number type of Scheme with the double type of Java, we cannot substitute the boolean type object there, because Java statically connects a type information with a variable name, not object. Thus double type and boolean type have no compatibility.

To treat these types in the same way, we use Java class typing system. We prepared a package of Java named Scheme package corresponding to the type of Scheme as figure 3.1.

In order to treat every type of objects in the same way, ScmObj is set up as a base class. Every other classes in this package is inheritance of this class. All of object which represent Scheme object including procedures are instances of these class. When programs are executed, all Scheme objects are treated as ScmObj. Thus all objects can be applied and returned in the same way.

In order to treat every objects as ScmObj, firstly, values which appears in Scheme programs have to be converted to ScmObj class objects. These functions are equipped to each Scheme package classes as constructors, because these functions must deffer corresponding to the type of value. Secondly, each value must be extracted, when it applied to primitive operators. Thus each class has at least two method, one for putting values into object, the other for extracting values from object.

However, since all of objects are typed as ScmObj in Java language, we cannot call these extracting methods of each class. Thus, at each application of primitive operator, first, operand objects are down-casted to desired class. This process also plays as dynamic type checking, because if the object is the instance of desired class, it will be down-casted normally, but if not, this process will raise exceptions. Next, the internal values are extracted from these operands. Then, they are applied to the operations actually. Finally, the result value have to be converted to ScmObj, too, for being treated in the same way.

In addition, we mentions the characteristic manipulations on values of Scheme.

Evaluation as Boolean In Scheme, all of object can be evaluated as boolean type. If the type of objects is boolean, the exact value is return. But if the type is another, it is evaluated as "true". In order to cope with this, every class should have the method "evalBool". Therefore it is implemented to ScmObj and returns "true" always. Moreover, it is also implemented to ScmBool, which return the exact value.

Sophisticated Number Type The class ScmNum represents the Number type of Scheme. It has internal data typed as "double". And arithmetic operations are performed by using the floating-point operator embedded in JVM. However, Scheme's model of arithmetic is not as simple as which of Java, yet more complicated.

It designed to remain as independent as possible of the particular ways in which numbers are represented within a computer. According to R^5RS[9], in Scheme, every integer is a rational number, every rational is a real, and every real is a complex number.

But in this paper, numbers are represented by using "double" simply. Perfect implementations of Scheme's arithmetic model is left for future work. Kawa[5] actually implemented this mathematical ideal into JVM.

```java
public class ScmObj {
   public boolean evalBool()
   {
      return true;
   }
}
public  class ScmNum extends ScmObj{
   public ScmNum(double d)
   {
      this.d = d;
   }
   double d;
   public double evalNum()
   {
       return d;
   }
}
public  class ScmBool extends ScmObj{
   public ScmBool(boolean b)
   {
      this.b = b;
   }
   boolean b;
   public boolean evalBool()
   {
       return b;
   }
}
```

```java
public  class ScmPair extends Scmobj{
   ScmObj car ;
   ScmObj cdr ;
   public ScmPair(ScmObj car , ScmObj cdr)
   {
       this.car = car;
       this.cdr = cdr;
   }
   public ScmObj evalCar()
   {
       return car;
   }
   public ScmObj evalCdr()
   {
       return cdr;
   }
}
```

Figure 3.1: Package for Representation of Scheme Objects in Java

3.2 Translation of Scheme Procedures to Java Codes

In this section, we introduce how to translate a procedure of Scheme into class of Java.

Scheme can deal with procedures as first-class values, but Java cannot deal with methods in the same way. Therefore we cannot simply replace procedures of Scheme with methods of Java. Moreover, in Scheme, even if a number of arguments of procedure application is different, it is not detected at the compiling time. And every applications is needed to be tested whether the first element of **ap**-syntax is really a procedure, and the procedure can accept it's arguments at execution time. If it is proper application, procedure is called and executed. If it is invalid application, an error is detected for the first time.

The requirement for implementing procedure is summarized as follows:

- In order to treat procedures of Scheme as object in the same way as others, it must be an object of an class inherited from ScmObj in Java language. Moreover the classes which represent the procedures must have more than one methods. This method behaves as the same way as the procedure.

- Type checking occurs when **ap**-syntax is executed. It must be confirmed that whether the first element of **ap**-syntax is a procedure type object and whether the number of arguments is correct.

In Scheme, procedures are made by λ-syntax. If all λ-expressions which appears in Scheme programs are replaced with the objects of the same class, those behaviors will be all the same. This means that, differently from another objects, λ-expression must be translated to unique classes, according to it's behavior.

Procedures must be treated in the same way as the other objects. This means that the classes which represent λ-expressions must be inherited from ScmObj. Thus procedures are treated as ScmObj as others.

The classes for procedure must have more than one method, which describe the behaviors of the procedures. Though methods in Java can be applied to only one pattern of arguments, in Scheme, there are the procedures which can be applied to many pattern, such as "+", which can be applied to one or more operands. In this case, the procedure class has to have all pattern of method which appear in Scheme programs. When the procedure is called at execution, according to the number of arguments, JVM select the suitable a method.

The body parts of method is decided by the behavior of procedure. Now, the classes which represents procedures are defined. Then, all λ-expressions are replaced with the construction of these class.

In translating Scheme procedures into Java classes, there are two points which must be considered, which are allocating variables in memory and tasks to be done when procedures are applied.

Variable Allocation In scheme programs, there are many variables. We have to consider where the variables are allocated in memory space.

Since Scheme was designed based on lambda calculus, local procedure definition is allowed. This means that procedures can be defined inside of the procedure, which methods of Java cannot. Moreover, for safety and convenience of Java, JVM restricts memory manipulations. When programmers have to reserve in memory space, they needs to make new object. For this restriction, programmers cannot create and operate the stack in memory directly.

The local variables have to be allocated in memory at every procedure call. This functions is implemented in Java, too. In Java, local variables are allocated in operand stack in JVM. However, as long as using this way, local variables are focused only on the method where these variables were defined. But in Scheme, local variables defined in a procedure are refereed or even rewritten by the procedures which are defined as local procedure.

Thus local variables must be allocated in the procedure class as member variables. By using this way, these variables can be referred by the procedure class itself by nature, and referred from the local procedures indirectly by the way of this outer procedure instance which is passed at construction of this inner procedures. That is, local procedures should have the outer procedure instance as member variables as a substitute for the variables defined at the outer procedure.

Thus the member variables which the procedure class should have is summarized as follows:

Instances of outer procedures The instance of the class of outer procedure. These variables are substituted at instantiation of a local procedure class, that is, when a λ-expression is evaluated.

Local Variables The variables defined inside of a procedure.

Arguments The arguments of procedure must be stored as member variables, too, because they can be accessed from local procedures. These variables are substituted when a procedure is called.

Tasks at Procedure Application Before procedure applications are executed at **ap**-syntax, two tasks are required. First, dynamic type checking must be executed to confirm whether the first element of **ap**-syntax is procedure object, and if so, it can be applied to the present arguments. In order to set these informations into a procedure, interfaces are implemented to the procedure class. Each interface has only one method. An interface represents an information how many arguments the procedure can accept. For example, primitive procedure "car" implements an interface which has "`ScmObj app(ScmObj arg1,ScmObj arg2)`", because this procedure can be applied only to two arguments. But, since the procedure "list" can be applied to any numbers of arguments, it must implement all interfaces of patterns which appear in the Scheme program.

When **ap**-syntax is executed, the first element of it is casted to a interface which has the method of present pattern of application. If the first element is procedure and has this interface, procedure is applied to present arguments normally, but if not, an exception is raised. Because all objects but procedures don't implement these interfaces, by casting to a interface, procedures can be distinguished from another objects.

However, it is not all to do before procedure applications. In our implements, local variables are stored in objects as member variables. Continuously, if the procedure is called recursively, it's local variables will be updated and old values will be disposed. When returned from recursive call, the procedure cannot access the old values. These variables should be stored in stack naturally, and when the procedure is called recursively, new memory spaces are allocated on the stack. That is when called recursively, a procedure needs new memory space. Thus, when the procedure is called, it must duplicate itself to acquire new spaces in memory space, which is corresponding to stack growth. This is accomplished by using "clone" function implemented in "Object" class. Therefore, procedure class must implements "Clonable", too. After the dynamic type checking, second, "clone" method must be called. Only after these operations, a procedure actually is executed.

Based on the above, translation, functional language into procedural language, is performed by:

- λ-expressions are replaced with the instantiation of the procedure classes.
- **ap**-syntaxes are translated into the method calls of Java.
- **if**-syntaxes are translated into **if-else** branches of Java.
- **let**-syntaxes is translated into substitutions of Java.

```
(let ((sqr (lambda (x)
               (* x x))))
   (sqr 3))
```

Figure 3.2: Sample.scm

- And final value is returned at the bottom of the method.

3.3 Example

We translate the Scheme program to Java for example.The target program “Sample.scm” is shown in figure 3.2.

The result of translation of “Sample.scm” is shown in figure 3.3.

```
interface Ifoo {
   public ScmObj app(ScmObj a1);
}
public class Sqr extends ScmObj
                 implements Cloneable,Ifo
{
  Sample sample;
  ScmObj x;
  public Sqr(Sample sample)
  {
    this.sample = sample;
  }
  public ScmObj app(ScmObj x)
  {
     ScmObj v;
     v = new ScmNum(((ScmNum)x).evalNum() + ((ScmNum)x).evalNum());
     return v;
  }
  public Object clone() {
     return super.clone();
  }
}
public class Sample extends ScmObj
{
    ScmObj sqr;
    public Sample()
    {
    }
    public ScmObj app()
    {
       sqr = new Sqr(this);
       ScmObj v;
       v = ((Ifo)sqr.clone()).app(new ScmNum(3));
       return v;
    }
}
```

Figure 3.3: Sample.java

Chapter 4

Optimization with Soft Typing

We can translate Scheme to Java by treating all of the object of Scheme as ScmObj in Java. For example, (+ 1 2) is translated to

```
new ScmNum(new ScmNum(1).evalNum() + new ScmNum(2).evalNum())
```

However, in this case, useless instantiations appear. If we can translate that simply to "1 + 2", it will be much more efficient.

If it turns out that the Scheme Object, such as "1", is used only as *number* type, it can be expressed by simply "1", not "ScmNum(1)" in Java. This simplification leads elimination of dynamic type checking and class instantiation. We aim to apply this simplification as many expressions as possible by using soft typing system[6].

A soft typing system can statically, that is compiling time, assign types to all objects in programs written in a dynamically typed language like Scheme. By using this type informations, the estimations of the usage of each objects can be restricted. If the usage is decided uniquely, the object needs no type informations, because it is guaranteed the safety by soft typing system.

In this chapter, we introduce soft type assignment, and optimization by using it. First, we define a static type for Core Scheme. Next, we introduce the type discipline and then the way to assign types. Finally, we explain the optimization by using type informations.

$$
\begin{aligned}
D &= numbers \\
&\cup\ boolean \\
&\cup\ null \\
&\cup\ \{< v_1, v_2 > | v_1 \in D_1 \subseteq D \ and \ v_2 \in D_2 \subseteq D\} \\
&\cup\ \{f | f(v_1, \ldots, v_n) \in D_2 \subseteq D \ for \ all \ v_1 \in D_1 \subseteq D, \ldots, v_n \in D_n \subseteq D\}
\end{aligned}
$$

Figure 4.1: Scheme Type Domain

4.1 Our Static Types Definition

To construct a static type system for Core Scheme, we must define a static types. We partition the data domain of types into disjoint subsets as figure 4.1.

For Core Scheme, all numbers inhabit the same partition, because primitive operations such as addition or division can produce small integer, big integer, rational, real or complex answers. (In other language, like ML, where there are distinct operations for integers and reals, then we would use distinct partitions for integers and reals.) This time, we restricted the use of complex data structures to only "pair" structure for simplicity, because other data structures can be typed in the same way. The partitions containing pairs are further subdivided as the component of each (D_1, D_2) which are partitioned into the same way. Much the same is true on procedures.

Objects in dynamically typed programs belong to many different semantic types, and dynamically typed programs routinely exploit this fact. Thus the objects in dynamically typed programs often cannot be assigned just to one type. For example, (if #t 1 #t) cannot be assigned one type, that is neither *number* type or *boolean* type work well. To accommodate these, a soft type for Core Scheme should include union types.

We define the type of Scheme:τ as follows.

$$
\begin{aligned}
\tau ::= \emptyset \mid \alpha \ \cup \ \ldots \ \cup \ \beta \\
\mid \mu\alpha.\tau \quad (\alpha, \ldots, \beta \in D)
\end{aligned}
$$

τ is a disjoint union of zero or more partitions. The union type describes possibility of types. For example, (if #t 1 #t) is assigned as $number \cup boolean$. As explained

$$\frac{\tau \prec TypeOf(c)}{A \vdash c : \tau}$$

$$A \vdash x : A(x)$$

$$\frac{A \vdash e_1 : (\tau_2 \rightarrow \tau_1) \cup \tau_3 \quad A \vdash e_2 : \tau_2}{A \vdash (\textbf{ap}\ e_1\ e_2) : \tau_1}$$

$$\frac{A[x \mapsto \tau_1] \vdash e : \tau_2}{A \vdash (\lambda\ (x)\ e) : (\tau_1 \rightarrow \tau_2) \cup \tau_3}$$

$$\frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2 \quad A \vdash e_3 : \tau_2}{A \vdash (\textbf{if}\ e_1\ e_2\ e_3) : \tau_2}$$

$$\frac{A \vdash e_1 : \tau_1 \quad A[x \mapsto \tau_1] \vdash e_2 : \tau_2}{A \vdash (\textbf{let}\ ([x\ e_1])\ e_2) : \tau_2}$$

Figure 4.2: Static Type Discipline

above, this expression cannot be assigned to unique type, but this union type indicates that this expression has either *number* or *boolean* type.

In addition to union type, we introduce the recursive types as shown by $\mu\alpha.\tau$. Recursive types represent infinite regular trees. The type $\mu\alpha.\tau$ binds α in τ. We have $\mu\alpha.\tau = \tau[\alpha \mapsto \mu\alpha.\tau]$. Recursive types must be formally contractive, i.e., phrases such as $\mu\alpha.\alpha$ are not types. By using this recursive type, we can express ,for example, β type list as $\mu\alpha.cons(\beta, \alpha) \cup null$.

4.2 Static Type Discipline

Before assignment of types to Core Scheme expressions, we define the type discipline which our soft typing system must follow.

Type discipline is shown in figure4.2. Let e be a program expression and let t be a Scheme type. The formula $e : t$ is called a typing judgment. The intended meaning is that e has type t. Let A be a type environments which are finite maps from identifiers to Scheme types. $A[x \mapsto \Sigma]$ denotes the functional extension or update of A at x to Σ. The function $TypeOf$ maps the constants of Core Scheme to type schemes describing their behavior. For example, number object "1" is typed as $\forall v.num \cup v$ by $TypeOf$ function. v is type variable. $\prec$ means one of the substitution of Scheme types to the variables of type scheme. In one case, such as (if #t 1 2), the empty set: $\emptyset$ will be substituted v. But in another case, such as (if #t 1 #f), *boolean* type will be substituted to v.

Basically, union type system allows only once appearance of each structured data type constructor. That is, for example, the type $< boolean, number > \cup < boolean, boolean >$ is invalid, in order to forbid type ambiguity at "cdr" position. But, in our system, multiple appearance of type constructor is allowed only for procedure type corresponding to the number of arguments, because these are distinguishable in both of Scheme and Java. This allowance weaken the interaction of procedure objects which should be unified, then types can be assigned more freely and more clearly.

In *Soft Typing* written by Robert Cartwrite[6] and Mike Fagan or in *A Practical Soft Type System for Scheme* written by Andrew K. Write and Robert Cartwright[16], type polymorphic functions are introduced. By introducing these functions, static types can be assigned more precisely. However, in this system, it is not introduced, because of Java's type restrictions. In ML, the function "id x = x" is typed as $\forall \alpha . \alpha \rightarrow \alpha$. But in Java, the method which work as same as this function cannot be defined, because Java doesn't allow type polymorphism. Implementation of type polymorphic functions in Java is left for future work, which is explained at chapter8.

4.3 Type Inference Algorithm

We introduce how the types of objects are determined. In principle, the type informations of Scheme doesn't appear in the program source. Therefore, the type must be inferred and the inferred types must follow the static type discipline.

For the optimizations which explained later, types should be as unambiguous as possible. The type of constants is assigned by $TypeOf$ function as type scheme initially. If the empty set: $\emptyset$ is substituted to all of the variables which appears in these type scheme, it is the least ambiguous type assignments. But, this way does not always satisfy the static type discipline. We aim to assign static types to program expressions as unambiguous as possible with keeping the static type discipline.

Here, first, we present how to infer the static types. It is similar to that of ML.

The static type of the expression: e is determined as follows.

1. If e is constant, the type is assigned by the *TypeOf* function. For example, #f is assigned as $boolean \cup \alpha$, where α is a type variable.

2. If e is variable reference, the type is extracted from a type environment A.

3. If e is λ-expression, such as $(\lambda\ (x)\ e_1)$, first, new type variable τ_1 is assigned to x, next, the type of e_1 is inferred under the extended type environment $A[x \mapsto \tau_1]$. If the type of e_1 is assigned as τ_2, The type of e is inferred as $\tau_1 \rightarrow \tau_2$.

4. If e is **if**-syntax, such as (**if** $e_1\ e_2\ e_3$), first, the types of e_1, e_2, e_3 are inferred, next, the types of e_2 and e_3 are unified. The way to unify is explained later. Then the type of e is inferred as same type as e_2 and e_3.

5. If e is **ap**-syntax, such as (**ap** $e_1\ e_2$), and the inferred types of e_1 and e_2 are α and β respectively, α and $\beta \rightarrow \tau$ are unified, where τ is new type variable. Then the type of e is inferred as τ.

6. If e is **let**-syntax, such as (**let** $(x\ e_1)\ e_2$), first, the type of e_1 is inferred, for example, as α, then the type of e_2 is inferred under the extended type environment $A[x \mapsto \alpha]$. The type of e is inferred as same as the type of $e2$.

Next, we explain the function of *unify*. The function of our unifier is basically similar to the unifier which introduced "*Types and Programming Language*" [13],pp.326-329. However, our unifier doesn't check the type consistency, but update type environment to fill the static type discipline. If the expressions, for example, e_1 whose type is $number \cup \alpha$ and e_2 whose type is $boolean \cup \beta$, are unified, first, unifier try to find minimal substitutions which unify these types. As a result, the unifier find the substitutions $[\alpha \rightarrow boolean \cup \gamma]$ and $[\beta \rightarrow number \cup \gamma]$, where γ is new type variable. Then the type environment A is updated to $A[\alpha \rightarrow boolean \cup \gamma][\beta \rightarrow number \cup \gamma]$, and both of e_1 and e_2 are typed as $boolean \cup number \cup \gamma$.

After the type inference, the variables whose type values cannot be decided uniquely are substituted the empty set value: $\emptyset$ for to reduce ambiguity.

4.4 Optimization by using Type Informations

When all of Scheme expressions have been assigned static types, the translated code of Java can be optimized by using these type informations.

If one of the Scheme expression, for example (if #t 1 #f) , is assigned $number \cup boolean$ type , it means that this object may be treated as either *number* or *boolean* type. In order to treat this object as both of types simultaneously, this must be treated as ScmObj, and "1" and "#f" must be translated to ScmNum(1) and ScmBool(false) respectively. As mentioned, this is not efficient.

But if the type is assigned uniquely, for example (if #t 1 2) typed as *number*, it means that this object is treated only as a *number* type. Now this object can be treated as the type “double” in Java, since it is guaranteed that this object is never treated as except for a *number* type. Therefore, this object doesn't have to be instantiated of ScmNum and simultaneously be extracted by evalNum method.

With the optimizations by type inference, we must correct the method of translation. Though all the translated objects were treated as the instance of ScmObj before, now with type optimizations, various types which is not compatible with ScmObj, like “double”,“boolean” and so on, appear.

We must mainly modify the translation of Scheme-to-Java about the following two cases.

Interface Since all types were ScmObj before, the interfaces could be distinct only from the number of arguments. But now these must be distinguished by type informations of arguments in addition to the number of arguments.

For example, in case, a procedure “p” has the type $number \cup boolean \rightarrow number \cup null$, then this procedure must be implemented the interfaces, like

```
interface Ifoo {
   public ScmObj app(ScmObj a1);
}
```

But in another case, a procedure has the type $number \rightarrow number$, then this must be implemented the interfaces, like

```
interface Ifdd {
   public double app(double a1);
}
```

These cases must be distinguishable. These interfaces decided at compile time, and the procedures are casted to an appropriate interface by the compiler.

Complex structures As earlier definition, ScmPair class could store at the “car” or “cdr” positions just to ScmObj class object. But now we must treat other type as well. This means that ScmPair must store these various type object.

We redefine ScmPair as figure 4.3. Now, various constructors and evaluation methods are added, corresponding to the pattern of types which appears in Scheme programs.

```
public  class ScmPair extends Scmobj{
   double dcar ;
   double dcdr ;
   :
   :
   ScmObj ocar ;
   ScmObj ocdr ;

   public ScmPair(double dcar , double dcdr)
       this.dcar = dcar;
       this.dcdr = dcdr;
   }
   public ScmPair(ScmObj car , double dcdr)
       this.ocar = ocar;
       this.dcdr = dcdr;
   }
   :
   :
   public ScmPair(ScmObj car , ScmObj cdr)
       this.ocar = car;
       this.ocdr = cdr;
   }
   public double evalDCar()
   {
       return dcar;
   }
   :
   :
   public ScmObj evalOCdr()
   {
       return ocdr;
   }
}
```

Figure 4.3: Redefinition of ScmPair

However, there is a possibility that this extension leads unsafe errors. If a program sequences of Java, like

```
ScmPair p = new ScmPair(1,false);
return p.evalOCdr();
```

, are executed, a problem occurs. That is, first, when the constructor is called, "1" of *number* type and "false" of boolean type are stored to "dcar" and "bcdr" respectively, then, when the method evalOCdr is called, the value stored to "ocdr" returns. But no value is stored there. Therefore, a meaningless value will return, and then execution of the program is not correct any longer.

But, since soft type system assures the soundness of the program. Since constructors and methods are determined by the result of this system, such unsafe program sequences will not be generated as a result of the translation.

Chapter 5

Result of Experiments

We actually tried executing the translated programs on Solaris 2.6, SUN Ultra Enterprise 4000, CPU: UltraSPARC 168MHz x 14, memory: 1.2GB. And we used Solaris Java VM version 1.2.1 and Chez Scheme version 6.0a.

In order to clarify the degree of improvement as a result of optimization by soft typing, we measured the time which is taken up to execute these. We measured under several conditions. First, we executed the original Scheme programs by chez scheme. Second, we translated the Scheme programs into Java language, compiled them to Java bytecodes and executed them on JVM. But in this case, for comparison we didn't optimized by soft typing. This means that every object appears in programs is treated as ScmObj. Last, we translated, compiled and optimize them by soft typing.

The sample programs are "Quick Sort", "Fibonacci Numbers" and "Ray Tracing". Table 5.1 shows the time taken up to execute those samples. Each results reflect the features of a program respectively .

Quick Sort This program sorts the contents of the number list by "quick sort algorithm". List is made by combining a pair object in Scheme as linked list. The

	Chez Scheme [msec]	Without type inference [msec]	With type inference [msec]
Quick Sort	430	6024	4917
Fibonacci Numbers (30)	1820	17814	9680
Ray Tracing	11510	39728	23731

Table 5.1: Times token up to execute some test programs

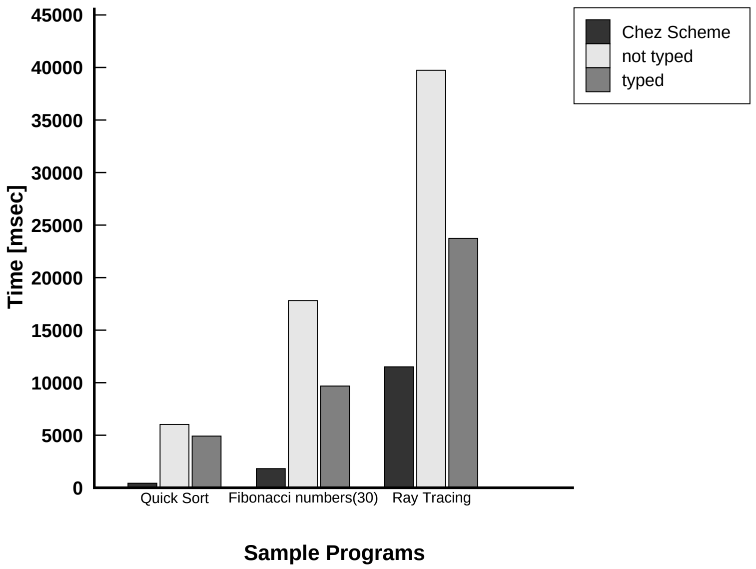

Figure 5.1: Mesured execution time. First, executed on Chez Scheme. Second, translated to Java language without optimization and executed on JVM. Third, translated to Java language with optimization by using Soft Typing and executed on JVM

large portion of quick sort program is consist of the number comparison operations and the pair manipulations. The "car" position of each pair in number list is typed as *number* uniquely. But the "cdr" position of each pair cannot be typed uniquely, which is typed as $numberlist \cup null$. On account of this, the most of the type of objects stored at "cdr" positions are treated as ScmObj and dynamic type checking will occur at each pair handling operations. This lead much lowering of efficiency. Thus the soft typing optimization is not work well and there is not large deferences between optimized and not optimized.

List structure is the most common data type in Scheme programs. But, it is not primitive data type, but the user defined data type consist of pairs and null. Thus, it's type is a little ambiguous. In order to translate the list operations into more effecient code, we must change the representation of pair and null. For example, nill is represented of null in Java and typed as ScmPair, Or ScmList class is set as the parent class of ScmPair and ScmNull.

Fibonacci Numbers We tried to find Fib(30). In this program, most of the operations are consist of the arithmetic operations, in particular addition. By the static type inference, these numbers are typed to the *number* type correctly. Owing to this, most of arithmetic operations doesn't cause the dynamic type checking and executed normally without loss of efficiency. Thus, the optimization work well. As a result, the program with type optimization is 1.8 times faster than the the program without it.

For further optimizations, tail call elimination is required. The implementation of it is explaind in chapter8.

Ray Tracing This program make the 3-dimensional images by calculating the color of each pixel using the informations of each objects, such as shape, color and so on and direction and brightness of light source. For calculation of log and trigonometric functions, we use Java library.

This program also uses arithmetic operations mainly. However, since this uses the list and vector like as a structure in *C language*. Our type inference rule is apt to unify the every elements of vector and some of list, in order to apply these elements to general vector or list handling procedures. Consequently, many elements of vector of list are treated as ScmObj. In order to treat these objects as proper type, the each positions of list and vector must be assigned defferent types. Treating in this way is already applied to list type, because it consist of pair and null. As opposed to this, applying vector is much difficult. The most simple way for this is to replace vector with list.

But, in this program, since most of number objects are typed correctly, this program can be optimized at a certain degree.

Our results shows that the translated programs, whether optimized or not, are far less efficient than the original programs which are executed by Chez Scheme. Chez Scheme compiled scheme programs to machine codes dynamically at execution time. Concurrently with this, it optimize the code. In our translation, because of translating to Java language, many restrictions of Java language, such as static class type system and limited memory access, prevents from optimization. Thus our soft typing systems cannot be implemented perfectly. As opposed to this, no restrictions are imposed when compiling to machine codes. Thus Chez Scheme can optimize as much as possible.

However, by soft typing programs can be translated more effective. Moreover our implements have an advantage over Chez Scheme in portability and compatibility with Java.

Chapter 6

Evaluation

Our result indicates that programs can be improved to closely twice as efficiency occasionally.

Though JVM is useful run-time environment, it is designed to accommodate only to Java language by nature. Thus, JVM has many facilities which are useful for programs compiled from mainly Java language, such as garbage-collections and bytecode level object construction to control memory spaces securely and efficiently, and verifier to detect suspicious codes which is liable to execute undefined operations and invoke invalid stack states. However, in order to implements these facilities, JVM has to be imposed many restrictions. First, for controlling memory spaces securely and efficiently, programmers cannot access these memory spaces directly. Second, for security, bytecodes must follow static class type discipline. Third, for avoiding stack over flow and type inference, the state, which is the number of operands and it's type, of operand stack has to be decided uniquely at every bytecodes. These features are much useful to write programs in Java language and execute it.

However, these features often prevents from implementing compilers for the other languages. As far as compiling Scheme goes, indirect memory access prevents from effective representation of type informations. If there is no restrictions in case compiling to machine codes, we will add these informations paired with objects transforming these informations to simple integer and dynamically type check by comparing these integers, without creating new objects and down casting it. Moreover, our translated programs imposed two type disciplines, which is consist of a static type discipline as a result of soft typing imposed at compiling time and Java's type discipline imposed to bytecodes. These restrictions lower the bytecode level expressiveness, then it repress the potential of soft typing. In this paper we cannot implement type polymorphic procedures for this reason. This subject is argued in next chapter.

We consider the portability is one of the important elements. For leaving the portability of Java, the translated programs must follow Java's security rules, that is to pass the Java verifier, because the program rejected by verifier is regarded as dangerous one. In order to translate Scheme into Java holding it's security, we must implement the many features of Scheme by replacing with functions of Java. There are many implementations of Scheme-to-Java, which satisfy these conditions. However, almost all of these implement dynamic type system by simply replacing with Java class type system being less concerned with efficiency. In this paper, we introduced soft typing system for improving in efficiency. However, there is some approaches for this purpose apart from ours. Oiwa[12] achieved this purpose by extending JVM adding new type called descriptor. Such approaches extending JVM can implement dynamic typing system effectively, because no restrictions are imposed to extending JVM and some restrictions imposed on JVM can be removed. Thus, the programs translated and executed in this way will be more effective than ours. However, extending JVM indicates loss of portability, because the translated programs for extended JVM cannot be executed on a normal JVM. On the other hand, Our approach can be applied to normal JVM and optimize all programs. Most of expressions of programs should be written to be as strictly typed as possible, because ambiguous typed expressions are liable to be bugs. In such cases, programs can be typed uniquely and be highly optimized. But the programs which take full advantages of Scheme, it, beyond our power, cannot be typed uniquely and it's expressions are treated as ScmObj, so cannot be optimized well.

Soft typing system have another advantage. By a static type inference, our translator can point out the some expressions which may or must be an error because their types are ambiguous or invalid. Notifying these to the programmer as warnings of bugs and sopposing to rewrite these, the programmer can find the bugs and correct these easily. Moreover, since the improved codes will be more strictly typed, the codes can be translated to be more effective.

Chapter 7

Related Work

Per Bothner [5] provided a set of Java classes for implementing dynamic languages named Kawa. Kawa is also an implementation of near R^5RS Scheme using these classes. It implemented the full of Scheme number types, multiple values, continuations, proper tail recursion and so on. Kawa also compiles Scheme to the bytecode instructions of the Java Virtual Machine and by using Java class-loader can execute as interpreter immediately after compilation. However, since Kawa aims also to be the class library for compiling general functional languages to Java bytecodes, the representation of object which are treated in this library must be general-purpose, that is, be implemented simply. Thus the representations cannot be optimized by using type information, which weaken the generality and Kawa takes up for dynamically type check at execution.

Kenneth R. Anderson, Timothy J. Hickey and Peter Norvig made SILK[2], which is a compact Scheme implemented in Java. It is an implementation of a large subset of $R4^4RS$ Scheme. SILK can treat Java classes and call methods of class in Scheme and methods in Java can call all Scheme procedures. SILK does not introduce soft type system, too. Because it extends Scheme to be able to treat Java classes and methods, for the implementation, soft typing system have to be extended to deal with types of Java. Type system of Java is much different from our soft type system, because it requires subtyping system corresponding to class hierarchy. Even if all of objects are assigned as ScmObj which are used in both of Java and Scheme, because at application Java class method in Scheme, typing system must investigate whethe the class has such method or not. For this type system, type domain must be extendable.

Yutaka Oiwa [12] extended JVM to improve performance of dynamically type checking. JVM is extended by adding the new type called “descriptor”, which is the super-type of integer and reference. Owing to this, dynamic type system can be implemented without constructing type alternative classes. However, this implementation is achieved at the cost of portability, because JVM must be extended. This approach have higher quality in efficiency than ours, but our approach retains the portability of Java.

Chapter 8

Future Work

8.1 Duplication of Polymorphic Procedures

In this implementation, we could not implement the type polymorphic procedures. For example, "(lambda (x) x)" will be typed as $\forall\alpha.\alpha \to \alpha$ with type scheme in certain language like ML. Consequently, "((lambda (x) x) 1)" and "((lambda (x) x) #f)" typed as *number* and *boolean* respectively. But in Java, the method which work in the same way as this procedure cannot be defined. Because, in Java, types are not inferred automatically, but written in source code explicitly. Thus, by using the method

```
ScmObj app(ScmObj a)
{
   return a;
}
```

the expression "app(new ScmNum(1))" will be typed as "ScmObj" in Java, not "ScmNum".

Mlj[4] implemented the type polymorphic procedures on JVM. If we implement in a similar way, "(lambda (x) x)" will be translated into

```
double app(double x)
{
   return x;
}
boolean app(boolean x)
{
```

```
   return x;
}
  .
  .
  .
```

Corresponding to all patterns which apears in source code, one or more methods are defined. Inferring the types by using soft typing, our translater can decide which of these methods should be called at compiling time. Thus, "((lambda (x) x) 1)" and "((lambda (x) x) #f)" are typed as *number* and *boolean* respectively. By using type polymorphic procedure system, types will be decided more precisely, and then the result code will be more optimized.

However, this method has negative aspects. This method leads increments of code size. Thus, if the code size is emphasized, it is better not to use this method, but if the efficiency is emphasized, this method should be applied.

8.2 Proper Tail Recursion

According to Revised^5 Report on the Algotithmic Language Scheme [9], properly tail recursive is requiered, which means the procedure-call without stack growth and direct return to caller waiting at the bottom of stack at tail call. For the practical translator, we should follow this.

However, implemeting proper tail-recursive demands low-level stack manipulations, which JVM doesn't allow. Thus full implementation of proper tail recursion is impossible. But this can be implemented partially. There are some approaches to implements tail call elimination on JVM.

Kawa [5] and MLj [4] partially implemented the proper tail recursion. Using the fact that tail calls are actually self-tail-call in many cases, they replace the tail calls with simple loop codes using *goto*. This, as far as the self-tail-call goes, implemented proper tail recursion correctly. If we implement as this, the programs, like fibonacci number, will be more optimized. And this approach we can implement this without a drastic revision.

However this method often inadequate for JVM, because it leads growth of codes in one method and JVM does not allow methods bodies of more than 64 kilobytes, and obviously this cannot handle the procedure which jumps over the methods. The other approach is to use a trampoline system[15]. In this system, procedures devided into two kind, tail-call procedure and not. A tail-call procedure, when it called, return the closure which is a continuation of this function, and then this closure is also called by the outer procedure which call the tail-call procedure. This approach prevent the stack growth, but the efficiency is not so optimized. Another approach is devised by Henry G.Baker[3] and it is modified for JVM by Michel Schinz and Martin Odersky[14]. In this system, programs are converted to continuation passing style(CPS), therefore turning all calls into tail calls, and compiled as usual. At run-time, when the depth of recursion reaches a certain point, the current continuation is built and returned to the trampoline waiting at the bottom of the call stack.

In order to implement this into our system, we must rebuild our translation system, representation of Scheme objects and soft type system from scratch. In addition to the implementation of CPS, our representation of Scheme in Java and type system have to be repaierd to handle these. But translating intermediate language: CPS is much useful to implement *call-with-current-continuation* procedure, even if partially.

References

[1] Kenneth R. Anderson and Timothy J. Hickey. Reflecting java into scheme. In *Reflection*, pages 154–174, 1999.

[2] Kenneth R. Anderson, Timothy J. Hickey, and Peter Norvig. Silk - a playful blend of scheme and java.

[3] Henry Baker. Cons should not cons its arguments, part ii: Cheney on the m.t.a.

[4] Nick Benton, Andrew Kennedy, and George 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.

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

[6] 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.

[7] Mike Fagan. *Soft Typing: An Approach to Type Checking for Dynamically Typed Languages*. PhD thesis, 1992.

[8] Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture (FPCA), La Jolla, California*. ACM Press, 1995.

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

[10] Assaf J. Kfoury and Santiago M. Pericas-Geertsen. Type inference for recursive definitions. pages 119–128, 1999.

[11] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.

[12] Yutaka Oiwa. Extending java virtual machine to improve performance of dynamically-typed languages. Senior thesis, University of Tokyo, Department of Information Science, February 1999.

[13] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[14] Michel Schinz and Martin 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.

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

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