A RULE-BASED MODEL OF A RUN-TIME PROGRAM GENERATION SYSTEM

BY

TANKUT BARIS AKTEMUR

B.S., Bilkent University, 2003

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

Abstract

In this work we present Mumbo, a simplified version of Jumbo. Jumbo is a staged Java compiler supporting run-time program generation (RTPG). To incorporate RTPG, Jumbo relies on the *compositionality* principle, by which the compiler written for static compilation can also be used for RTPG. This property, however, makes run-time compilation inefficient. Fortunately, it is possible to optimize the compiler by partially evaluating it for each code piece, but because of Java's syntactical details and the complexity of Jumbo's implementation, this transformation becomes difficult and time-consuming to implement. Therefore we develop a simplified version of Jumbo, called Mumbo. Mumbo implements a subset of Jumbo but keeps the crucial properties.

Mumbo is a typed, object-oriented language, supporting RTPG. Its implementation is rule-based and done in Maude, a tool that provides executable rule-based programming. In this thesis we explain details of the language and the compiler. We provide optimizations to partially evaluate the Mumbo compiler to decrease run-time generation cost. With Mumbo, we have been able to make rapid progress in implementing the optimizations and we obtained substantial speedup of run-time compilation. We propose Mumbo as a suitable "testbed" of Jumbo. Mumbo can be used to try new ideas and experiments first, to save time. We also discuss the lessons we have learned from Mumbo.

Acknowledgments

I am thankful to my advisor, Prof. Samuel Kamin. This work wouldn't have existed without his supervision. He also provided my financial support.

Prof. Grigore Roşu's classnotes of the Programming Language Design course were very useful when implementing the syntax and semantics of Mumbo and `LowLevel`.

Paul Adamczyk made a detailed review of the thesis and provided very useful feedback. He also has been kind to listen to the progress of the work over countless meetings for lunch and fish sandwiches on Fridays. I have had very helpful discussions on Jumbo and Mumbo, with Philip Morton. He also suggested the name "Mumbo". Thank you guys!

Lastly, I would like to thank my family, for their love and support.

Table of Contents

Chapter

List of Figures

List of Tables

Chapter 1

Introduction

In this work we present our efforts to model a run-time program generation (RTPG) system, using rule-based programming. The RTPG system we are modeling is Jumbo [17, 6, 13, 14]. It is developed in our research group.

Jumbo provides a high degree of programmer control, source-level specification, and bytecode generation (see Section 1.2). It can compile all of Java 1.4. These properties make Jumbo suitable to use in "real" applications. However, there is a trade-off. These advantages come with the price of complexity in the implementation of Jumbo. In Java, there is more than one way of expressing a computation. Jumbo has to handle all of the different possibilities. This makes the implementation of Jumbo and program transformation/analysis complicated and time-consuming. We experienced this fact while working on optimization of the program generators written in Jumbo.

In RTPG, the user has to pay the initial cost of run-time generation. A generated program is presumably faster than its static version, but as a general rule, it has to be executed many times in order to compensate its run-time generation cost. It is desirable that this cost be as low as possible, so that a generated program starts paying off as early as possible. There are different approaches to decrease the cost of run-time generation. Limiting the flexibility in declaring code fragments is one of them [22, 9]; we do not wish to do this. In Jumbo, we partially evaluate the back-end of the compiler to specialize it for each code piece declared. This method was developed by Lars Clausen as part of

his PhD thesis [6]. However, as we mentioned, because of the complexity of Java and Jumbo's implementation, experimenting with new ideas and writing new optimizations are difficult and time-consuming. These reasons drove us to develop a simplified version of Jumbo, which would become a model of it. In this thesis, we explain how we developed this model, called Mumbo ("mini-Jumbo"). In Mumbo, we can successfully optimize program generators and thus decrease run-time generation cost substantially. We have learned several lessons from Mumbo, which we are planning to apply on Jumbo. Mumbo, in the future, can be used as the first place to experiment with new ideas.

Mumbo's implementation is rule-based and it is done in Maude [7]. Maude is a powerful tool supporting rewriting logic. It provides two types of transitions: equations and rewrite rules. Rewrite rules are used for modeling concurrency. In Mumbo we do not have concurrency. *All the rules we present in Mumbo are equational.* Maude makes these equations executable. This gives us the opportunity to actually run our specifications. Furthermore, equational logic is, in terms of representation, closer to how the optimizations are defined. This makes Maude very suitable to define transformations.

This thesis is structured as follows: In Section 1.1 we give an overview of run-time program generation. In Section 1.2 we explain Jumbo. Chapter 2 gives details of the syntax, preprocessing and semantics of Mumbo. We compile Mumbo programs to a lower level language, called `LowLevel`, which is explained in Chapter 3. In Chapter 4 we discuss the structure of the Mumbo compiler. This is followed by the optimizations, in Chapter 5. In Chapter 6 we give examples to illustrate how run-time generation can be optimized. In the following chapter, we discuss the limits of our set of optimizations. Finally, in Chapter 8, we discuss the lessons we have learned, and conclude.

All the files related to Mumbo are available at Mumbo's webpage: `http://pinatubo.cs.uiuc.edu/~aktemur/mumbo`. We also provide the source code of the compiler in the appendix.

1.1 Run-time Program Generation

In this section we give an overview of run-time program generation (RTPG). We do not attempt to present a complete survey here, but we outline the main ideas.

Run-time program generation, briefly, is the ability to define and combine program pieces at run-time. The pieces, after forming a compilable unit[1], can be compiled and brought into execution. The basic idea is to compose these pieces in such a way that the resulting program will be more efficient than the version that would be implemented statically. The efficiency can be in terms of both time and space. At implementation time, a program has to be written to handle every kind of input. The input that will be received is not known at this time. However, at run-time, when the input becomes available, a new program specialized for that input can be generated. This is the main motivation behind RTPG.

RTPG is similar to partial evaluation [8], but there is an important diference. In partial evaluation the specialization is done automatically or with little user support. On the other hand, RTPG gives the programmer the power to specialize. This is mainly done by extending a language with special quotation mechanism. There are various RTPG systems existing in the literature, including Jumbo [17, 6, 13, 14], quasi-quotations in Lisp [4], DynJava [22], 'C [9, 26, 25], MetaML [30], Metaphor [21] and CodeBricks [3].

As we mentioned, lowering the run-time generation cost is desirable and one approach is to limit the variability in the code pieces. Consider this code piece in Jumbo:

```
$< x + 1 >$
```

This can either be an addition or string concatenation, depending on the type of `x`. If it is an addition, again depending on the type of `x`, it may be an integer, float or double operation. A system like Jumbo should consider all these possibilities. However, DynJava, for example, requires that the type of each free variable be given as part of the defined code piece. The above fragment would then look similar to this, if `x` is an integer:

```
${x : int}< x + 1 >$
```

[1] In Jumbo, this unit is a list of classes, or interfaces.

This specification improves the speed of run-time generation by eliminating many cases which the RTPG system has to handle. But it results in less flexibility. For instance, consider a `Point` object with two fields, `x` and `y`, representing its location. In order to write a piece of code which displays this point's location, we would have to know the type of `x` and `y`. Assume the type is `int`:

```
${x : int, y : int}< System.out.println("X:" + x + " Y:" + y); >$
```

This limits the usability of the fragment — we cannot use it if the type of the fields is `double` or `String`.

In Jumbo, to increase the run-time generation performance while keeping generality, we partially evaluate the back-end of the compiler, which produces the bytecode. The compiler is generic and handles every type of program fragment without assuming any specific property. However, it can execute faster if we make it aware of the properties of the code fragments. For instance, from the compiler's point of view, the code piece `$< x + 1 >$` is just a binary operation between some two operands. On the other hand, by partial evaluation, the compiler can be specialized for this piece so that it knows that the operation is actually a plus sign, the left operand is a variable named `x` and that the right operand is an integer literal 1. This is the main motivation behind optimization of program generators in Jumbo and Mumbo. We will return to this point in Chapter 6, and discuss it in more detail with examples.

1.2 Jumbo

Jumbo [17, 6, 13, 14] is a staged compiler for Java. It allows run-time program generation. In Jumbo, similar to other RTPG systems, program pieces can be defined using a quotation mechanism. The brackets Jumbo uses are `$<` and `>$`. Between these brackets, almost any parsable code fragment can be defined. Those pieces are then put together at run-time to form a complete program to generate. This way, Jumbo provides the programmer with a high degree of control and source level specification. Jumbo does not require the existence of a compiler at run-time. Having the Jumbo API is enough to be

able to generate programs at run-time. All these properties make Jumbo an easy-to-use and powerful RTPG system.

From the point of view of a programmer, the code pieces declared between brackets `$<` and `>$` behave like strings. This makes it easy to reason about the programs that will be generated. However, the code pieces cannot be arbitrary. They should be parsable units and the values that the fragments represent are of type `Code`, not string. Regular string operations cannot be applied on them.

One can also define *holes* in code pieces. A hole in a code fragment can be filled in with other code fragments. The syntax used for a hole is back-quotation (‘). For parsing concerns, in Jumbo, a back-quote should be followed by a syntactic category. Consider the following code:

```
public Code loopGen(Code condition, Code body){
  return $< while(‘Expr(condition)){
              ‘Stmt(param);
              x++;
            } >$ ;
}
```

In this method, the condition of the while-loop and part of the body are holes. The code pieces which fill in these holes are received as parameters of the method. Finally, the resulting code object is returned. An invocation of `loopGen` might be as follows:

```
loopGen($< x <= 5 >$,  $< obj.foo(x); >$);
```

In this case, the returned `Code` value would represent the following code:

```
while(x <= 5){
  obj.foo(x);
  x++;
}
```

This code is now ready to fill in a hole in an enclosing code piece which defines `x` and `obj`.

For expressions of primitive type, there is a second type of anti-quotation in addition to ‘`Expr`. It evaluates the expression at program-generation time and then inserts the

value into the generated code as a constant. For example, `‘Int(x)` means that `x` is an `int` variable and its *current* value is to be inserted into the enclosing code.

Jumbo, as we mentioned at the beginning of this section, is a staged compiler. It is a traditional compiler implemented *compositionally*, meaning that the compilation of a code fragment is a function of the compilation of its subfragments. In Jumbo, every code piece is of type `Code`. A `Code` value is the *partially* compiled version of the code that it is representing. It is a function which produces the byte-level code corresponding to the program fragment it represents. This function accepts an *environment* as its parameter, which includes all the information necessary to calculate the byte-level code. The low level code calculated is returned as the result of the function. In Java, functions are not first class citizens, but objects are. Therefore, instead of a function, a `Code` value in Jumbo is represented as a “function object”, which defines the function as its method. Thus, the application of that function simply becomes the invocation of the object’s method.

The compiler, because of its compositionality, does not have to know anything about quotations or holes. In Jumbo, all the quotations and back-quotations are removed during parsing. The quoted-codes are converted to their equivalent `Code` values representing them. This conversion removes the necessity to define semantics for quotation/back-quotation. Mumbo, as a model of Jumbo, also has a preprocessing phase to convert code fragments to `Code` values, and the Mumbo compiler is written compositionally, like the Jumbo compiler. The preprocessing phase is explained in Section 2.2, and the compiler in Section 4.1.

More information on Jumbo can be found in [17, 6, 13, 14]. The foundational ideas behind Jumbo-style RTPG are discussed in [15, 16]. Jumbo is downloadable at `http://loome.cs.uiuc.edu/Jumbo`.

Chapter 2

Mumbo

Mumbo is a simplified version of Jumbo. It is typed, object-oriented, and supports run-time program generation. Mumbo consists of three main parts: Syntax, Preprocessing and Semantics. Every part is written in Maude, using (conditional) equations. The syntax consists of approximately 200 lines of code. The preprocessing part is in slightly more than 300 lines, and the semantics is in 420 lines.

There are two ways of executing a Mumbo program. (1) Through the executable semantics of Mumbo. (2) First compiling Mumbo to `LowLevel` and then executing the resulting `LowLevel` program through the executable semantics of `LowLevel`. The syntax and semantics of `LowLevel` are given in Chapter 3.

The first way, that is, the executable semantics of Mumbo, may be used to run a regular Mumbo program. However, if a program is generating a program and bringing it into execution at run-time, it does not make sense to use executable semantics. This is because the code generated at run-time is in `LowLevel`, and the semantics of Mumbo knows nothing about `LowLevel` code. We, in fact, do not use this execution method even to run regular programs. We choose to run them after compiling them to `LowLevel`. This is because only then does Mumbo become a true model of Jumbo, which compiles programs to JVM bytecode. However, it should be noted that the Mumbo compiler is implemented in Mumbo (see Chapter 4). In other words, the Mumbo compiler *is* a Mumbo program. Therefore we need a mechanism to run the compiler to be able to

compile programs. The answer is the executable semantics. Since the Mumbo compiler is a Mumbo program, this is the natural and easy way of compilation. We explain how to pass Mumbo programs to the Mumbo compiler in Section 4.4.

In this chapter we first give the grammar of Mumbo, followed by the preprocessing phase. Finally we give the semantics of the language. For defining the syntax and the semantics in Maude, we follow the conventions of [27] and [19].

2.1 Syntax

Mumbo uses `$<` and `>$` brackets, as in Jumbo, to define program fragments. Caret (`^(...)`) is the anti-quotation character. `^I(...)`, `^B(...)` and `^S(...)` can be used to lift integers, booleans and strings, respectively. `^F(...)` and `^M(...)` are the syntactic categories to anti-quote a field and a method, respectively. The corresponding anti-quotations in Jumbo are `‘Int`, `‘Bool`, `‘String`, `‘Method` and `‘Field`. In Mumbo, there is no distinction between expressions and statements. Therefore `‘Expr` and `‘Stmt` anti-quotations of Jumbo are handled by the same anti-quotation in Mumbo, namely (`^(...)`).

In Mumbo, strings are defined by placing built-in Strings of Maude or quoted identifiers (`Qid`s) between square brackets, such as `["abc"]` and `['xyz]`. `Qid`s are used as names (e.g. `'aName`). `self` is a special name, and it refers to the current object. `NIL` is the null pointer. `#` is the string concatenation operator. There are four arithmetic operations: `++, --, **, -:-` for addition, subtraction, multiplication and division, respectively. We do not use the usual operators like `+` and `-` because they are built-in operators defined as commutative. This does not hold in Mumbo, because of side effects. Consider the following code.

```
{ set 'a = 1 ;  'a ++ {set 'a = 10 ; 2} }
```

The result of the addition is 3. On the other hand, it would be 12 if the code was:

```
{ set 'a = 1 ;  {set 'a = 10 ; 2} ++ 'a }
```

A field of an object can be accessed with `->`, as in `'obj -> 'f`. The local variables used in the body of a method should be declared as part of the method header. If there are no local variables, the user is free to omit the declaration, or simply enter `[noVars]`. A variable declaration list has the form `['a`$_1$ `: 't`$_1$`, ..., 'a`$_n$ `: 't`$_n$`]`, where `'a`$_k$ is the name and `'t`$_k$ is the type of a variable. Following the variable declaration list comes the return type of the method. An example method is shown below.

```
method 'foo('x : 'string) ['a : 'int] 'int:
{  set 'a = send self 'bar('x) ;
   'a -:- 10
}
```

A program is an executable unit consisting of a list of classes and a `main` method. `main` is similar to a method, but it does not have a parameter list and a return type.

We do not explain every detail of the syntax and believe that the remaining syntactic issues will be clear from the grammar given in Figure 2.1 and from the examples throughout the thesis.

The grammar is given in extended Backus Naur Form (EBNF). The reader will note that it is ambigious. Ambiguities can easily be resolved by introducing new token types. We do not do that here to keep it simple. In its implementation in Maude, the ambiguities are solved by using precedence values. We also note that parameters in a parameter list, and expressions in an argument list are separated by commas. This is not shown in the figure to keep it shorter.

2.2 Preprocessing

In Mumbo, quotations and anti-quotations are not part of the kernel language. They are used as syntactic sugar. A program is preprocessed to remove quotations. We define two mutually recursive functions for this purpose: `preprocess` and `code`.

`preprocess` is the function we call first. It traverses a given program fragment and converts quoted parts into code objects by calling `code`. The `preprocess` function does

$PROGRAM$	::=	$CLASS^*$ `main [` $VARDECLS$ `]` EXP
$CLASS$	::=	$FINAL$ `class` $NAME$ `extends` $NAME$ $FIELD^*$ $METHOD^*$
$FINAL$	::=	| `final`
$FIELD$	::=	$FINAL$ `field` $NAME$ `:` $NAME$ | `^F(` EXP`)`
$METHOD$	::=	$FINAL$ `method` $NAME$ $PARAMLIST$ `[` $VARDECLS$ `]` $NAME$ `:` EXP
	|	$FINAL$ `method` $NAME$ $PARAMLIST$ $NAME$ `:` EXP
	|	`^M(` EXP`)`
$PARAMLIST$	::=	`()` | $(NAME$ `:` $NAME)^+$
$VARDECLS$	::=	`noVars` | $(NAME$ `:` $NAME)^+$
EXP	::=	$NAME$ | anInteger | $STRING$ | $BOOLEXP$ | $BINEXP$
	|	`self` | `NIL` | `if` EXP `then` EXP `else` EXP
	|	EXP `->` $NAME$ | `set` LHS `=` EXP
	|	`while` EXP EXP | `do` EXP `while` EXP
	|	`new` $NAME$ $EXPLIST$ | `send` EXP $EXPLIST$
	|	`cast` EXP `to` $NAME$ | $BLOCK$ | $QUOTE$ | $LIFT$
$STRING$	::=	`[` aString `]` | `[` aQuotedIdentifier`]`
$NAME$	::=	aQuotedIdentifier
$EXPLIST$	::=	`()` | EXP^+
$BINEXP$	::=	EXP `#` EXP | EXP `++` EXP | EXP `--` EXP
	|	EXP `**` EXP | EXP `-:-` EXP
$BOOLEXP$	::=	EXP `equals` EXP | `not` EXP
	|	EXP `and` EXP | `True` | `False`
LHS	::=	$NAME$ | EXP `->` $NAME$ | `^` EXP
$BLOCK$	::=	`{` $BLOCKEXPS$`}`
$BLOCKEXPS$	::=	EXP | EXP `;` $BLOCKEXPS$
$LIFT$	::=	`^I(` EXP`)` |`^B(` EXP`)` |`^S(` EXP`)`
$QUOTE$	::=	`$<` EXP `>$` \| `$<` $FIELD$ `>$` \| `$<` $METHOD$ `>$` \| `$<` $CLASS^+$ `>$`

Figure 2.1: Mumbo's Grammar

not change pieces occurring out of quotations. Below is its definition for addition and if expressions.

```
eq preprocess(E1 ++ E2) = preprocess(E1) ++ preprocess(E2) .

eq preprocess(if E then E1 else E2) =
     if preprocess(E) then preprocess(E1) else preprocess(E2) .
```

`preprocess` behaves like the identity function except for quoted parts. It just applies itself recursively on the subfragments of the code it is applied on, as seen above. When it detects a quotation, it invokes the `code` function as shown below.

```
eq preprocess($< E >$) = code(E) .
```

We do not define `preprocess` on anti-quotations. This ensures that backquotes occuring out of quotations will raise an error.

`code` replaces fragments with code objects which represent those fragments. When it detects an anti-quotation, it invokes the `preprocess` function because anti-quoted parts belong to the outer scope. To put it differently, `preprocess` operates on the first stage, and `code` operates on the second stage. In the equations below, `code` replaces a summation with a code object representing the binary operation `"add"`, and replaces the if-expression with an instance of `'ifThenElseCode`. It calls `preprocess` on anti-quoted code.

```
eq code(E1 ++ E2) = new 'binOpCode(["add"], code(E1), code(E2)) .
eq code(if E then E1 else E2) =
        new 'ifThenElseCode(code(E), code(E1), code(E2)) .
eq code(^(E)) = preprocess(E) .
```

Mumbo generates code which is again in Mumbo. Therefore, it allows generation of program generators, i.e. programs having more than two stages can be written in Mumbo. We simply define the following equation to convert quotations occurring inside quotations. This makes it possible to preprocess programs which generate program generators.

```
eq code($< E >$) = code(code(E)) .
```

In RTPG terminology, anti-quoting immutable types is called *lifting*. In Mumbo, as in Jumbo, primitive types and strings can be lifted. This is provided by the following equations defined for integer, boolean and string lifting, respectively.

```
eq code(^I(E)) = new 'integerConstantCode(preprocess(E)) .
eq code(^B(E)) = new 'booleanConstantCode(preprocess(E)) .
eq code(^S(E)) = new 'stringConstantCode(preprocess(E)) .
```

Finally, we give some examples of the result of preprocessing. In these examples the code piece to the left of ⇒ is the original code, and the code to the right is its preprocessed version.

- `$< 1 >$` ⇒ `new 'integerConstantCode(1)`
- `$< 'x >$` ⇒ `new 'getNameCode(['x])`

- `$< set 'x = ^('c) >$` ⇒ `new 'assignmentCode(new 'getNameCode(['x]), 'c)`
- ```
  { set 'x = 2 ;
    $< 'x ** ^I('x) >$
  }⇒
  { set 'x = 2 ;
    new 'binOpCode(["mul"],
                   new 'getNameCode(['x]),          ---left operand
                   new 'integerConstantCode('x))    ---right operand
  }
  ```

If a program fragment has a variable number of elements (e.g. an argument list), it is converted to a linked list. For instance:

```
$< send 'obj 'foo(^(c)) >$
```

becomes

```
new 'invocationCode(new 'getNameCode(['obj]), ---target of invocation
                    ["foo"],                  ---name of the method
                    send (new 'LinkedList(NIL, NIL)) 'add('c)) ---arguments
```

Here, the argument list consists only of `'c`. We first create an instance of a linked-list, which is initially empty, and then add `'c` to the list. The list is then passed to the constructor of `'invocationCode`.

2.3 Semantics

In this section we give the semantics of Mumbo. We give the rules in Structural Operational Semantics (SOS) [24], in order to make it more readable for those who are not familiar with Maude syntax and to abstract out details. The Maude implementation of the semantics, however, closely follows the SOS style. For details, the source code is provided on Mumbo's webpage.

The semantics we give here mostly follows Java. Before presenting it in SOS, we explain some important points informally. Mumbo applies dynamic dispatching when calling a method of an object. That is, the search for the method starts from the run-time class of the object. When an object is created with the `new` command, its `'initialize`
```

method is called automatically. Explicit invocation of the `'initialize` method is not allowed. (This method is similar to the constructors in Java.) Subclasses can override methods of their superclasses. Method overloading is not supported. The interpretation of the `final` flag for classes, methods and fields follows that of Java: `final` classes cannot be subclassed, `final` methods cannot be overridden and `final` fields' values can be set only in the `'initialize` method. Every variable should have a declaration, either as a field, a parameter or a local variable. Parameters and local variables can shadow fields. Parameters and local variables of a method should have distinct names.

In Mumbo there is no distinction between statements and expressions. A block's value is the value of its last expression. Variable assignments and loops evaluate to 0. In method invocations, the target of the invocation is evaluated first, followed by evaluation of the arguments. The arguments are evaluated in left-to-right order. A method returns what its body evaluates to. The `new` command returns a reference to the object it instantiated.

Generation of new numbers could be easily done in Mumbo if we had static (i.e. global) variables or methods. But to keep the language simple, we do not support static members in Mumbo. This creates difficulty in generating unique names/numbers in a program. We overcome this problem by introducing special expressions into the language: `GENSYM` returns a new name in the form (`"% 'symX"`) [1], where `X` is a unique number. Similarly, `GENLABEL` returns a new label in the form (`"'labX"`), where `X` is a unique number. These two special expression are handled as part of the semantics.

As noted in Section 2.2, the preprocessing stage removes all the quotations. Hence, quotation and anti-quotation are not part of the kernel language, and we do not define their semantics.

Below we give the semantics of Mumbo in the SOS style. In the semantics, a state σ keeps the information necessary to evaluate a program. It includes an *evironment*, a *store*, and the current object. The environment maps names to locations, that is, it is

[1] In the target low-level code, quoted identifiers are used as labels, and quoted identifiers prefixed with a percent sign are used as names. This is explained in Chapter 3.

a set of pairs of name and location. Similarly, the store maps locations to values. An object is a tuple of its run-time type (i.e. the name of its class) and its own environment (i.e. the fields).

$$State = (Environment \times Store \times Object)$$

$$Environment = \{(n,\ l) \mid n \in Name,\ l \in Location\}$$

$$Store = \{(l,\ v) \mid l \in Location,\ v \in Value\}$$

$$Object = (Name \times Environment)$$

$$Value = Object \mid String \mid Integer \mid Boolean \mid \emptyset$$

We use $env(\sigma)$, $store(\sigma)$ and $obj(\sigma)$ to refer to the elements of σ. $env(o)$ refers to the environment kept in object o.

The variables used are listed below:

$\sigma, \sigma', \sigma'', \sigma'''$	$\in$ State
st	$\in$ String. e.g: `"abc"`
qid	$\in$ Quoted identifier. e.g: `'abc`
e, e_k, be	$\in$ Expression
el	$\in$ List of expressions (i.e. an argument list)
eb	$\in$ List of block-expressions (i.e. a single expression or a semi-colon separated list of expressions.)
x, c, rt	$\in$ Name
i, i_k	$\in$ Integer
v, v_k	$\in$ Value
vl	$\in$ List of values
o	$\in$ Object
pl	$\in$ List of parameters
vd	$\in$ List of variable declarations

The operations used have the following interpretations:

$string(v)$: Converts the value v to a string. e.g: $string$(`'a`)=`"'a"`, $string$(1)=`"1"`.

$\sigma \lhd \mathcal{O}(o)$: Replaces the current object of σ with o, where $o \in Object$.

$\sigma \lhd \mathcal{E}(\epsilon)$: Replaces the environment of σ with ϵ, where $\epsilon \in Environment$.

$\sigma \lhd \mathcal{S}(\psi)$: Replaces the store of σ with ψ, where $\psi \in Store$.

$\sigma[x]$: Returns the value of x. It first checks $env(\sigma)$ to find x. This lookup succeeds if x is a variable. If it fails, that is, if x is a field, this function looks up x in $env(obj(\sigma))$. Recall that x maps to a value via a location.

$\sigma[x \leftarrow v]$: Sets the value of x to v.

$create(c)$: Creates an object of class c.

$meth(m, \sigma)$: Returns the syntactic representation of the method with name m, defined in $obj(\sigma)$.

$st_1 || st_2$: Concatenates st_2 to st_1.

Below we give the full list of semantic rules of Mumbo. A rule like

$$\frac{\mathcal{C}}{\langle e, \sigma \rangle \rightarrow \langle v, \sigma' \rangle}$$

should read as: "The expression e, when evaluated in state σ, gives the value v and changes the state to σ', if $\mathcal{C}$ holds."

Str : $\dfrac{\cdot}{\langle [st], \sigma \rangle \rightarrow \langle st, \sigma \rangle}$ **Self** : $\dfrac{\cdot}{\langle \texttt{self}, \sigma \rangle \rightarrow \langle obj(\sigma), \sigma \rangle}$

Qid : $\dfrac{\cdot}{\langle [qid], \sigma \rangle \rightarrow \langle string(qid), \sigma \rangle}$ **Nil** : $\dfrac{\cdot}{\langle \texttt{NIL}, \sigma \rangle \rightarrow \langle nil, \sigma \rangle}$

Name : $\dfrac{\cdot}{\langle x, \sigma \rangle \rightarrow \langle \sigma[x], \sigma \rangle}$ **Int** : $\dfrac{\cdot}{\langle i, \sigma \rangle \rightarrow \langle i, \sigma \rangle}$

Void : $\dfrac{\cdot}{\langle (), \sigma \rangle \rightarrow \langle void, \sigma \rangle}$ **EList** : $\dfrac{\langle e_1, \sigma \rangle \rightarrow \langle v_1, \sigma'' \rangle \quad \langle (e_2, el), \sigma'' \rangle \rightarrow \langle vl, \sigma' \rangle}{\langle (e_1, el), \sigma \rangle \rightarrow \langle (v_1, vl), \sigma' \rangle}$

Cons : $\dfrac{\langle e_1, \sigma \rangle \rightarrow \langle v_1, \sigma'' \rangle \quad \langle e_2, \sigma'' \rangle \rightarrow \langle v_2, \sigma' \rangle}{\langle (e_1 \ \# \ e_2), \sigma \rangle \rightarrow \langle st, \sigma' \rangle}$, where st is $string(v_1) || string(v_2)$.

$$\textbf{Sum}: \frac{\langle e_1, \sigma\rangle \to \langle i_1, \sigma''\rangle \quad \langle e_2, \sigma''\rangle \to \langle i_2, \sigma'\rangle}{\langle (e_1 \mathbin{++} e_2), \sigma\rangle \to \langle i, \sigma'\rangle}, \text{ where } i \text{ is the sum of } i_1 \text{ and } i_2.$$

$$\textbf{Sub}: \frac{\langle e_1, \sigma\rangle \to \langle i_1, \sigma''\rangle \quad \langle e_2, \sigma''\rangle \to \langle i_2, \sigma'\rangle}{\langle (e_1 \mathbin{--} e_2), \sigma\rangle \to \langle i, \sigma'\rangle}, \text{ where } i \text{ is the difference of } i_1 \text{ from } i_2.$$

$$\textbf{Mul}: \frac{\langle e_1, \sigma\rangle \to \langle i_1, \sigma''\rangle \quad \langle e_2, \sigma''\rangle \to \langle i_2, \sigma'\rangle}{\langle (e_1 \mathbin{**} e_2), \sigma\rangle \to \langle i, \sigma'\rangle}, \text{ where } i \text{ is the multiplication of } i_1 \text{ and } i_2.$$

$$\textbf{Div}: \frac{\langle e_1, \sigma\rangle \to \langle i_1, \sigma''\rangle \quad \langle e_2, \sigma''\rangle \to \langle i_2, \sigma'\rangle}{\langle (e_1 \mathbin{-:-} e_2), \sigma\rangle \to \langle i, \sigma'\rangle}, \text{ where } i \text{ is } i_1 \text{ divided by } i_2.$$

$$\textbf{Eq}_\textbf{t}: \frac{\langle e_1, \sigma\rangle \to \langle v, \sigma''\rangle \quad \langle e_2, \sigma''\rangle \to \langle v, \sigma'\rangle}{\langle (e_1 \texttt{ equals } e_2), \sigma\rangle \to \langle true, \sigma'\rangle}$$

$$\textbf{Eq}_\textbf{f}: \frac{\langle e_1, \sigma\rangle \to \langle v_1, \sigma''\rangle \quad \langle e_2, \sigma''\rangle \to \langle v_2, \sigma'\rangle}{\langle (e_1 \texttt{ equals } e_2), \sigma\rangle \to \langle false, \sigma'\rangle}, \text{ where } v_1 \text{ is not equal to } v_2.$$

$$\textbf{And}_\textbf{t}: \frac{\langle e_1, \sigma\rangle \to \langle true, \sigma''\rangle \quad \langle e_2, \sigma''\rangle \to \langle true, \sigma'\rangle}{\langle (e_1 \texttt{ and } e_2), \sigma\rangle \to \langle true, \sigma'\rangle}$$

$$\textbf{And}_\textbf{f}: \frac{\langle e_1, \sigma\rangle \to \langle t_1, \sigma''\rangle \quad \langle e_2, \sigma''\rangle \to \langle t_2, \sigma'\rangle}{\langle (e_1 \texttt{ and } e_2), \sigma\rangle \to \langle false, \sigma'\rangle}, \text{ where } t_1 \text{ and/or } t_2 \text{ is } false.$$

$$\textbf{Not}_\textbf{f}: \frac{\langle e, \sigma\rangle \to \langle true, \sigma'\rangle}{\langle \texttt{not}(e), \sigma\rangle \to \langle false, \sigma'\rangle} \qquad \textbf{Not}_\textbf{t}: \frac{\langle e, \sigma\rangle \to \langle false, \sigma'\rangle}{\langle \texttt{not}(e), \sigma\rangle \to \langle true, \sigma'\rangle}$$

$$\textbf{Tru}: \frac{\cdot}{\langle \texttt{True}, \sigma\rangle \to \langle true, \sigma\rangle} \qquad \textbf{Fal}: \frac{\cdot}{\langle \texttt{False}, \sigma\rangle \to \langle false, \sigma\rangle}$$

$$\textbf{If}_\textbf{t}: \frac{\langle e, \sigma\rangle \to \langle true, \sigma''\rangle \quad \langle e_1, \sigma''\rangle \to \langle v_1, \sigma'\rangle}{\langle \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2, \sigma\rangle \to \langle v_1, \sigma'\rangle}$$

$$\textbf{If}_\textbf{f}: \frac{\langle e, \sigma\rangle \to \langle false, \sigma''\rangle \quad \langle e_2, \sigma''\rangle \to \langle v_2, \sigma'\rangle}{\langle \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2, \sigma\rangle \to \langle v_2, \sigma'\rangle}$$

$$\textbf{FA}: \frac{\langle e, \sigma\rangle \to \langle o, \sigma'\rangle}{\langle e \mathbin{->} x, \sigma\rangle \to \langle (\sigma' \lhd \mathcal{E}(\emptyset) \lhd \mathcal{O}(o))[x], \sigma'\rangle}$$

$$\textbf{Set}_\textbf{x}: \frac{\langle e, \sigma\rangle \to \langle v, \sigma'\rangle}{\langle \texttt{set } x = e, \sigma\rangle \to \langle 0, \sigma'[x \leftarrow v]\rangle}$$

$$\textbf{Set}_\textbf{f}: \frac{\langle e_2, \sigma\rangle \to \langle v, \sigma''\rangle \quad \langle e_1, \sigma''\rangle \to \langle o, \sigma'\rangle}{\langle \texttt{set } e_1\texttt{-> } x = e_2, \sigma\rangle \to \langle 0, \sigma' \lhd \mathcal{S}(store((\sigma' \lhd \mathcal{E}(\emptyset) \lhd \mathcal{O}(o))[x \leftarrow v]))\rangle}$$

$$\textbf{Blk}_\textbf{s}: \frac{\langle e, \sigma\rangle \to \langle v, \sigma'\rangle}{\langle \{\ e\ \}, \sigma\rangle \to \langle v, \sigma'\rangle} \qquad \textbf{Blk}_\textbf{l}: \frac{\langle e, \sigma\rangle \to \langle v_1, \sigma''\rangle \quad \langle \{\ eb\ \}, \sigma''\rangle \to \langle v, \sigma'\rangle}{\langle \{\ e\ ;\ eb\ \}, \sigma\rangle \to \langle v, \sigma'\rangle}$$

$$\textbf{Loop}_\textbf{t}: \frac{\langle e, \sigma\rangle \to \langle v, \sigma''\rangle \quad \langle be, \sigma''\rangle \to \langle true, \sigma'''\rangle \quad \langle \texttt{do } e \texttt{ while } be, \sigma'''\rangle \to \langle 0, \sigma'\rangle}{\langle \texttt{do } e \texttt{ while } be, \sigma\rangle \to \langle 0, \sigma'\rangle}$$

$$\textbf{Loop}_\textbf{f}: \frac{\langle e, \sigma\rangle \to \langle v, \sigma''\rangle \quad \langle be, \sigma''\rangle \to \langle false, \sigma'\rangle}{\langle \texttt{do } e \texttt{ while } be, \sigma\rangle \to \langle 0, \sigma'\rangle}$$

$$\textbf{New}: \frac{\langle el, \sigma\rangle \to \langle vl, \sigma''\rangle \quad \langle invoke(\text{`}\texttt{initialize}, vl), \sigma'' \lhd \mathcal{O}(o)\rangle \to \langle v, \sigma'\rangle}{\langle \texttt{new } c\ el, \sigma\rangle \to \langle o, \sigma' \lhd \mathcal{O}(obj(\sigma))\rangle}, \text{ where } o \text{ is } create(c).$$

$$\textbf{Sym}: \frac{\cdot}{\langle \texttt{GENSYM}, \sigma\rangle \to \langle st, \sigma\rangle}, \text{ where } st \text{ is unique.}$$

$$\textbf{Lab}: \frac{\cdot}{\langle \texttt{GENLABEL}, \sigma\rangle \to \langle st, \sigma\rangle}, \text{ where } st \text{ is unique.}$$

$$\textbf{Send}: \frac{\langle e, \sigma\rangle \to \langle o, \sigma''\rangle \quad \langle el, \sigma''\rangle \to \langle vl, \sigma'''\rangle \quad \langle invoke(x, vl), \sigma''' \lhd \mathcal{O}(o)\rangle \to \langle v, \sigma'\rangle}{\langle \texttt{send } e\ x\ el, \sigma\rangle \to \langle v, \sigma' \lhd \mathcal{O}(obj(\sigma))\rangle}$$

$$\textbf{Inv}: \frac{meth(x, \sigma) \to (\texttt{method } x(pl)[vd]\ rt:\ e) \quad \langle e, (\sigma \lhd \mathcal{E}(\emptyset))[pl \leftarrow vl][vd \leftarrow \emptyset]\rangle \to \langle v, \sigma'\rangle}{\langle invoke(x, vl), \sigma\rangle \to \langle v, \sigma' \lhd \mathcal{E}(env(\sigma))\rangle}$$

$$\textbf{Cast}: \frac{\langle e, \sigma\rangle \to \langle v, \sigma'\rangle}{\langle \texttt{cast } e \texttt{ to } x, \sigma\rangle \to \langle v, \sigma'\rangle}$$

Although Mumbo is a typed language, we do not implement a type-checker, for the reason that it is not the focus of this thesis. We assume that all programs we receive

are well-formed. For this reason, the **Cast** rule in the semantics simply evaluates the expression-to-be-casted. Casting is especially useful in the context of program analysis. It allows us to propagate type information through the expressions. Analysis of Mumbo programs is explained in Chapter 5.

Chapter 3

LowLevel Language

We compile Mumbo to a toy virtual machine language, called `LowLevel`. The compilation is explained in Chapter 4; in this chapter we present `LowLevel`.

`LowLevel` is again implemented in Maude by us, in approximately 400 lines of code. It is a 3-address code language inspired by LLVM (Low Level Virtual Machine) [18]. In `LowLevel`, one can define functions and structs. It allows allocating new structs, accessing struct elements, and dynamic decision of the name of the function-to-be-called. These features make it a suitable target language for an object oriented language like Mumbo. `LowLevel` is a toy language providing basic features, and it shouldn't be compared to a real target language. In this chapter we explain the syntax and semantics of `LowLevel`. We then give a simple example.

3.1 Syntax

We give the syntax of `LowLevel` in extended BNF, in Figure 3.1.

We would like to note that operands in an operand list are comma-separated. We do not give this in the grammar to keep it shorter.

$PROGRAM$	::=	`DEFS` $VTABLE^+$ `IMPL` FUN^+
$VTABLE$	::=	`{` $NAME$ `:` `{` $NAME$ `,` $NAME$ `}`$^+$ `}`
FUN	::=	$NAME$ $VARLIST$ `{` $BBLOCK^+$ `}`
$BBLOCK$	::=	$NAME$ `:` $STMT^+$
$STMT$	::=	VAR `=` EXP `;`
	|	`br` $NAME$ `;`
	|	`br` $OPRND$ `?` $NAME$ `:` $NAME$ `;`
	|	`return` $OPRND$ `;`
	|	`store(` $OPRND$ `,` $OPRND$ `)` `;`
EXP	::=	$BINOP$ `(` $OPRND$ `,` $OPRND$ `)` | $UNOP$ `(` $OPRND$ `)`
	|	`malloc` | `NULL` | `NEWSYM` | `NEWLABEL` | `S(` aString `)`
	|	`malloc(` $STRUCT$ `)` | `call(` $OPRND^+$ `)`
$BINOP$	::=	`add` | `sub` | `mul` | `div` | `and` | `concat` | `seteq` | `setlt`
	|	`setle` | `getelementptr` | `getfunctionname`.
$UNOP$	::=	`not` | `load` | `copy`
$STRUCT$	::=	`{` `{` $NAME$ `,` VAL `}`$^+$ `}`
VAR	::=	`%` $NAME$
$VARLIST$	::=	`()` | VAR^+
$OPRND$	::=	$VARLIST$ | `true` | `false` | $NAME$ | anInteger
$NAME$	::=	aQuotedIdentifier | $NAME$ `$` $NAME$
VAL	::=	$STRUCT$ | $NAME$ | `loc(` anInteger`)` | `int(` anInteger`)`
	|	`str(` aString`)` | `bool(` aBool`)` | `nil`

Figure 3.1: The grammar of `LowLevel`

3.2 Semantics

We give the semantics of `LowLevel` informally.

The executable unit in `LowLevel` is a program. A program consists of a list of struct definitions and a list of functions. Every `LowLevel` program has to have a `main` function, which takes no parameters. The execution of the program starts from the `main` function, and the execution of a function starts from the `'entry` basic-block. We return from a function after executing the basic-block named `'end`. This basic-block should have only one statement: `return`. The operand to `return` is then evaluated and its value is returned. Other basic-blocks should end with a branch-statement. `store` and `load` are used to mimic writing to/reading from the memory.

Below we give the meaning of the expressions:

- The arithmetic operators are `add, sub, mul` and `div`. They take two operands and stand for addition, subtraction, multiplication and division, respectively.
- The logic operators `and` and `not` work as their names imply.
- `concat` takes two operands, which should result in string values, and returns the concatenation of these two strings.
- `seteq, setlt, setle` are comparison operations, which stand for "set if equal to", "set if less than" and "set if less than or equal to", respectively. They accept two parameters, and return boolean values.
- `load` expects one parameter, a pointer to a location. It returns the value of that location.
- `call` is used for function invocation. It takes variable number of parameters. The first parameter should be the name of the function to be invoked, and the rest are the arguments to the function.
- `copy` returns the value its argument evaluates to.
- `getelementptr` accepts two arguments: (1) A pointer to a struct. (2) A name. It returns a pointer to the element of the struct, whose name is the second argument.
- `getfunctionname` accepts two arguments. It expects the first one to be a struct, say s, and the second one to be a name, say n. The struct s keeps a name of a vtable, v. This name is used to retrieve v and find the function name, kept in v, that corresponds to n. The function name is returned. In other words, it finds the name of the function in order for dynamic dispatch to work correctly.
- `malloc`, when given a struct as the argument, puts the struct into the memory, and returns a pointer to it. If not given an argument, it allocates a space whose value is undefined, and returns the pointer to that space.

- `store` takes two arguments. The first is a value, and the second is a pointer. It stores the value in the location pointed by the pointer.

- (`br` l) branches unconditionally to the basic block named l. (`br` c `?` l_1 `:` l_2) first evaluates c. If it is true, then it jumps to the basic-block named l_1, otherwise it jumps to the basic-block named l_2.

3.3 A Simple Example

In this section we give a simple example to illustrate how a Mumbo program would be written in `LowLevel`. Below is a Mumbo program.

```
class 'A extends 'object
  field 'x : 'int
  method 'initialize() 'int :  --- this is the constructor
    set 'x = 1
  method 'getX() 'int :
    'x
main [noVars]
  send (new 'A()) 'getX()
```

In the main method of this program, which is the starting point for execution, first a new object of type `'A` is created. This object has a field named `'x`. In the constructor, `'x` is set to 1. Then the `'getX()` method of the object is invoked. This method returns the value of `'x`. Below is the corresponding `LowLevel` program.

```
DEFS --- Definition of 'object is put automatically by the compiler
     {'object : {'initialize, 'object}}
     --- Definition for class 'A
     {'A : --- Method 'getX is defined in class 'A
           {'getX,'A}
           --- Method 'initialize is defined in class 'A
           {'initialize,'A} }
IMPL
--- 'initialize method of 'object is automatically put by the compiler
'object $ 'initialize (% 'self )
{ 'entry :  br 'end ;
  'end :    return 0 ;
}
```

```
--- 'getX method of class 'A
'A $ 'getX (% 'self )
{ 'entry :  % 's2 = getelementptr(% 'self, 'x) ;
            % 's1 = load(% 's2) ;
            br 'end ;
  'end :    return % 's1 ;
}

--- 'initialize method of class 'A
'A $ 'initialize (% 'self )
{ 'entry :  % 's4 = add(0,1) ;
            % 's5 = getelementptr(% 'self, 'x) ;
            store(% 's4, % 's5) ;
            % 's3 = add(1,0) ;
            br 'end ;
  'end :    return % 's3 ;
}

--- the main method
'main()
{ 'entry :  --- first allocate an object, whose vtable points to
            --- 'A (i.e. the object is of type class 'A)
            --- and it has a field named 'x
            % 's10 = malloc( {{'VTABLE, name('A)}{'x, nil}}) ;

            --- Get the pointer to the 'initialize method and invoke it
            % 's12 = getelementptr(% 's10, 'VTABLE) ;
            % 's13 = load(% 's12) ;
            % 's14 = getfunctionname(% 's13, 'initialize) ;
            % 's11 = call (% 's14, % 's10) ;

            --- Get the pointer to the 'getX method and invoke it
            % 's7 = getelementptr(% 's10, 'VTABLE) ;
            % 's8 = load(% 's7) ;
            % 's9 = getfunctionname(% 's8, 'getX) ;
            % 's6 = call (% 's9, % 's10) ;
            br 'end ;

  'end :    --- return the result of the call to 'getX
            return % 's6 ;
}
```

Chapter 4

The Mumbo Compiler

In order to compile Mumbo programs to `LowLevel` code, we implemented a compiler. In this chapter we explain this compiler in detail.

The compiler is implemented in Mumbo and we use the executable semantics of Mumbo to run it. This way, the compiler can compile Mumbo programs. This method was introduced in Chapter 2. The compiler doesn't have a separate scanner or parser. Since we defined the syntax of the language in Maude, we can pass Mumbo programs to the compiler very easily, without the need to write a parser or an explicit AST. This is explained in Section 4.4.

The Mumbo compiler is a traditional compiler. It does not know anything about run-time generation. (Recall that the quoted expressions were converted to expressions in the kernel language in the preprocessing stage; see section 2.2). However, the Mumbo compiler, like the Jumbo compiler, has a very important property: *compositionality*. Compositionality states that the compilation of a program fragment is a function of the compilation of its sub-fragments. By means of compositionality the capability of run-time program generation comes for free. In other words, the static compiler can be used for runtime compilation.

Another property of the compiler — which, we think, is crucial for better optimizability — is *side-effect-freeness*. The compiler is written in functional style, which makes

it side-effect-free. We claim that this property makes it more suitable for optimizations. We discuss the optimizations in Chapter 5.

The compiler has one-and-a-half passes over a program. At the program level, we have a shallow pass over the classes, and at the class level, we have another shallow pass over the members of the class to collect information about them. Other than these, the compiler has only one pass over the program that is being compiled. This can be seen in the source code pieces we provide in this section and in the whole source code of the compiler given in the appendix.

In this chapter we first explain compositionality and then the side-effect-free structure of the compiler by analyzing the source code. This gives a detailed presentation of the internals of the compiler. In Section 4.3, we explain how to do runtime generation. Lastly, Section 4.4 shows how to pass Mumbo programs to the Mumbo compiler.

4.1 Compositionality

In Jumbo-style run-time program generation, a quoted program fragment is represented as a function, which represents the partially compiled version of the quoted-code. However, functions in Mumbo and Jumbo are not first-class citizens. Therefore, we use "function objects", which provide the method representing the quoted code. Then the application of that function simply becomes the invocation of the method. This method is named `eval`, and we refer to the objects generally as "code objects". All these objects have a base type: `Code`. In Mumbo, names are quoted identifiers. For this reason, from now on, we'll use `'eval` and `'Code` when we are referring to the eval method and the base code type in Mumbo.

We have stated that compositionality means the compilation of a code fragment is a function of the compilation of its sub-fragments. The compositionality occurs in the `'eval` method. The basic behavior is as follows: When a `'Code` object receives the `'eval` message, it invokes the `'eval` method of its sub-fragments. These invocations return the result of compilation of the sub-fragments — which is a `LowLevel` code in

Mumbo. Then the object combines these results properly and returns the combined result. The `'eval` method receives an environment as a parameter. The environment includes all the information necessary for the code fragment to be compiled down to low-level code.

In Mumbo, the environment that is passed to the `'eval` method is an instance of the `'Env` class and contains the following information: (1) The list of local variables and parameters of the enclosing method. (2) The list of fields of the enclosing class (including the fields of its super classes). (3) The name of that class. (4) The name of that class's superclass. (5) The list of all classes in the current program. The return value of the `'eval` method is a `'ClosedCode` object, which is a tuple consisting of two elements: `'lowlevelCode`, which is the virtual machine code produced, and `'lowlevelName`, which is the name of the register that keeps the result `'lowlevelCode` evaluates[1]. There are two methods in `'ClosedCode`. `'getCode()` returns `'lowlevelCode`, and `'getName()` returns `'lowlevelName`.

The compiler consists of a `'Code` class for each syntactic element. In addition to these, there is `'ClosedCode`, `'Env` and `'LinkedList`, and several auxiliary classes. Below is the `'Code` class representing an integer constant:

```
final class 'integerConstantCode extends 'Code

  final field 'value : 'int

  method 'initialize('v : 'int) 'int :
    set 'value = 'v

  method 'eval('env : 'Env)
  ['sym : 'string, 'lowlevel : 'string] 'ClosedCode :
  {
    set 'sym = GENSYM ;
    set 'lowlevel = 'sym # [" = add(0,"] # 'value # [")"] # [" ; "] ;
    new 'ClosedCode('lowlevel, 'sym)
  }
```

[1]Recall that `LowLevel` is a 3-address code. We would not need `'lowlevelName` if it were stack code.

In the `'eval` method of the class above, we first get a new `LowLevel` name and then assign the value represented by this code object, to this new name. `'integerConstantCode` is one of the base cases of compositionality — it does not have sub-fragments.

Another base case is a variable expression, that is, an occurrence of a name. Compared to an integer constant, it is more complicated to compile a name, because the name may be a local variable or a field — or an error if it is free. We need to use the information stored in the environment to find out where the name is defined. We produce different `LowLevel` code depending on whether the variable is a local variable or a field. The `'eval` method of `'getNameCode` is given below.

```
method 'eval('env : 'Env)
  ['sym : 'string, 'lowlevel : 'string, 'temp : 'ClosedCode] 'ClosedCode :
{
  if (send 'env 'isLocal('name)) then
  { --- either a local var or a method parameter
    set 'sym = ["% "] # 'name ;
    set 'lowlevel = [""]
  }
  else if (send 'env 'isField('name)) then
       { set 'sym = GENSYM ;
         set 'temp = send self 'getPtr() ;
         set 'lowlevel = send 'temp 'getCode()
                       # 'sym # [" = load("] # (send 'temp 'getName())
                       # [")"] # [" ; "]
       }
       else
       { set 'sym = [" CANNOT RESOLVE: "] # 'name ;
         set 'lowlevel = [""]
       } ;
  new 'ClosedCode('lowlevel, 'sym)
}
```

The `'getPtr` method which is called if the name is a field, has the following implementation:

```
method 'getPtr() ['temp : 'string, 'tempSym : 'string] 'ClosedCode :
{
  set 'tempSym = GENSYM ;
  set 'temp = 'tempSym # [" = getelementptr(% 'self, "]
            # 'name # [")"] # [" ; "] ;
  new 'ClosedCode('temp, 'tempSym)
```

```
}
```

We now show the code of the class which represents a binary operation. It sends the `'eval` message to its operands, puts their results together and uses them to compute its own value.

```
final class 'binOpCode extends 'Code
  final field 'op : 'string
  final field 'left : 'Code
  final field 'right : 'Code
  method 'initialize('o : 'string, 'l : 'Code, 'r : 'Code) 'int :
  {
    set 'op = 'o ;
    set 'left = 'l ;
    set 'right = 'r
  }
  method 'eval('env : 'Env)
    ['sym : 'string, 'lowlevel : 'string,
     'lval : 'ClosedCode, 'rval : 'ClosedCode] 'ClosedCode :
  {
    set 'sym = GENSYM ;
    set 'lval = send 'left 'eval('env) ;
    set 'rval = send 'right 'eval('env) ;
    set 'lowlevel = (send 'lval 'getCode())
                  # (send 'rval 'getCode())
                  # 'sym # [" = "] # 'op # ["("]
                  # (send 'lval 'getName()) # [", "]
                  # (send 'rval 'getName()) # [")"] # [" ; "] ;
    new 'ClosedCode('lowlevel, 'sym)
  }
```

We finally show the `'eval` method of the `'Code` class representing a class. In this method, the environment is defined by setting the class name and the superclass name kept in it. The method then adds all the fields defined in the class and its superclasses to the environment. The environment is then ready to be passed to the `'eval` of the sub-fragments, which are the code objects representing the methods of the class. Lastly, the method list is iterated and the `'eval` method of each method is called. The `Low-Level` code obtained from these calls are put together and the result is returned.

```
method 'eval('env : 'Env)
  ['lowlevel : 'string, 'meth : 'List, 'methVal : 'ClosedCode,
```

```
   'sname : 'string, 'spr : 'defineClassCode] 'ClosedCode :
{
  --- defining the environment
  set 'env = send (send 'env 'resetForNewClass('name, 'superName))
            'addList('fieldList) ;
  set 'sname = 'superName ;
  while not('sname equals ['object])
  {
    set 'spr = send 'env 'getClassInfo('sname) ;
    set 'env = send 'env 'addList('spr -> 'fieldList) ;
    set 'sname = send 'spr 'getSuperName()
  } ;
  --- defined the environment

  set 'lowlevel = [""] ;
  set 'meth = 'methodList ;
  --- send 'eval message to all the methods defined in the class
  while (send 'meth 'hasNext()) {
    set 'methVal = cast send 'meth 'value() to 'Code ;
    set 'lowlevel = 'lowlevel
                 # (send (send 'methVal 'eval('env)) 'getCode()) ;
    set 'meth = send 'meth 'next()
  } ;
  new 'ClosedCode('lowlevel, [" $NONAME-FOR-CLASSES$ "])
}
```

With the source codes given above, we show the compositionality of the compiler. The classes representing other syntactical units are written in the same style. We do not show them here. Their code can be found at Mumbo's webpage: `http://pinatubo.cs.uiuc.edu/~aktemur/mumbo` and in the appendix.

4.2 Side-Effect-Freeness

The Mumbo compiler is written in functional style. We see this as a crucial property because it makes the compiler *side-effect-free*. This property makes the compiler more amenable to optimizations. We discuss the optimizations in Chapter 5. In this section we explain how to write the compiler in functional style.

We achieve side-effect-freeness by defining every field in the compiler as `final`. We are able to do this mainly because the `'Code` objects do not change their state after they

are created. `'ClosedCode` is simply a tuple, so its fields can easily be declared final. For the other classes, such as `'Env` and `'LinkedList`, we do the following: When there is need to change the state of an object, we return a new object with the new state. For instance an `'Env` object returns a new `'Env` when a new field is added to the field-list.

```
--- 'varList, 'fieldList, 'currClass, 'superClass, 'classInfos
--- are data members of Env
method 'addField('name : 'string) 'Env :
  new 'Env('varList,
           send 'fieldList 'add('name),
           'currClass,
           'superClass,
           'classInfos)
```

Similarly, a `'LinkedList` object returns a new `'LinkedList` object when we add a new element to the list. This style potentially has negative effect in the performance compared to non-functional version, but as we show in this thesis, it brings advantages in optimization. It also causes creation of many intermediate objects with short life-time. These objects can be garbage collected. However, we do not have a garbage collector in Mumbo, because it is not in the scope of this work.

4.3 How to use the compiler

In the previous sections we explained the internals of the compiler. In this section we look at it from the user's point of view and explain how to use it to compile Mumbo programs.

We distinguish two types of compilations: static time and run-time. Static time compilation happens when we write a program and compile it to get `LowLevel` code. Run-time compilation happens when a compiled program generates code during its execution. In the latter case, we send the `'generate` message to the `'Code` object which is representing a full program or class. This returns the corresponding `LowLevel` code.

The source code of `'generate` is given below. This method is defined in the `'Code` class, which is the base of all `'Code` objects.

```
final method 'generate() 'string :
  send (send self 'eval(new 'Env(new 'LinkedList(NIL, NIL),  ---var. list
                        new 'LinkedList(NIL, NIL),           ---field list
                        [""],                       ---enclosing class name
                        [""],                       ---superclass name
                        new 'LinkedList(NIL, NIL)) ---list of classes
                       )) 'getCode()
```

Below is a simple example showing how to generate a class.

```
class 'Gen extends 'object
  method 'getCode() 'Code :
    $< class 'Temp extends 'object
         method 'getId() 'int :
           123
    >$

main [noVars]
  send (send (new 'Gen()) 'getCode()) 'generate()
```

The compilation starts with an empty environment. Mumbo does not have a statement like `import` to include other classes. When it is desired to start code generation with some classes in the initial environment, `'generateWithClasses` should be used instead of `'generate`. This method accepts a list of classes. Each element of the list is expected to be a `'Code` object representing a class, that is, a `'defineClassCode` object (`'defineClassCode` is a subclass of `'Code`).

```
final method 'generateWithClasses('lib : 'List) ['env : 'Env] 'string :
{ set 'env = new 'Env(new 'LinkedList(NIL, NIL),
                      new 'LinkedList(NIL, NIL),
                      [""],
                      [""],
                      new 'LinkedList(NIL, NIL)) ;
  set 'env = send 'env 'addList('lib) ;
  send (send self 'eval('env)) 'getCode()
}
```

For static time compilation, one can use the `compile` operation. When given a program, this operation can be reduced to give `LowLevel` code. It is explained in the next section.

4.4 The `compile` operation

We define an operation, called `compile`, to get the `LowLevel` code for programs. The equation is defined as follows and illustrated in Figure 4.1.

```
eq compile(P) = run((MCompiler
                     main [noVars]
                       send ($< P >$) 'generateWithClasses(
                                        preprocess($< MCompiler >$)))) .
```

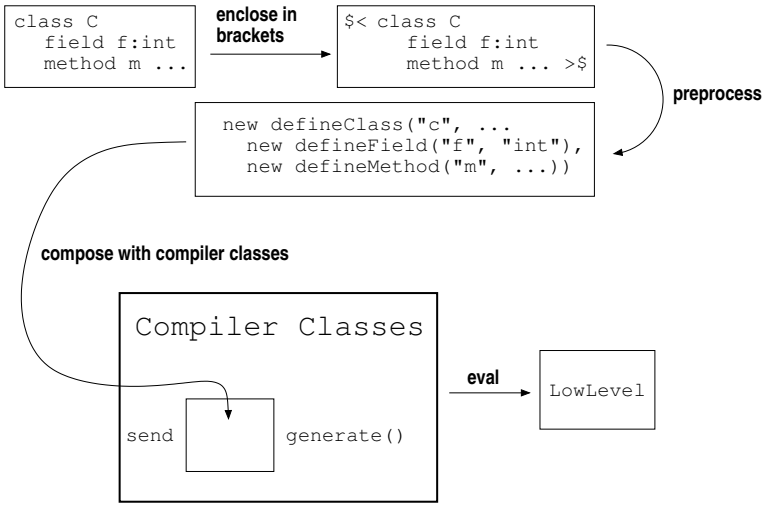

Figure 4.1: Passing Mumbo programs to the Mumbo compiler.

In this equation, `P` is a Mumbo program, `MCompiler` is the constant which holds all the classes of the compiler, and `run` is an operation which first preprocesses its input program, and then evaluates it using the executable semantics. The `compile` operation first encloses the given program between brackets. This makes it a code object. We then generate the `LowLevel` code of this program by sending to it the `'generateWithClasses` message. (We do not use the `'generate` method because the program may be program-generating. In that case we need to have the compiler classes present in the environment.) As mentioned above, the list of classes that is passed to the `'generateWithClasses` method should be a `'LinkedList` object consisting of code objects representing the classes. This is trivially achievable by enclosing the classes inside brackets and then

preprocessing the resulting expression. Finally, this expression becomes the body of a `main` method and we `run` it as a program. Remember that the compiler is itself a Mumbo program. Therefore, we can compile the compiler by just passing it to the `compile` operation. We define the following equation to make it easy. (We add the `"main [noVars] 1"` part to make it a full program.)

```
eq compileLibrary = compile((MCompiler main [noVars] 1)) .
```

We use the compiled compiler when executing program generating programs, which are compiled to `LowLevel`.

Chapter 5

Optimizations

A specialized program normally runs faster than a generic program. However, run-time generation has a considerable cost. Because of this cost, the generated program does not pay off immediately; it first amortizes its generation cost. The moment when it catches up with the non-generative version of the program is called the *crossover point*. We, naturally, would like to have a low crossover point so that run-time program generation compansates its cost earlier.

In this chapter we define optimizations to rewrite program generators at source level, so that they will generate code faster at run-time. In this chapter we first explain how to write traversers easily. We then explain the program analyses. These analyses will be used by optimizations. Afterwards we give the notation we use to explain the optimizations. We choose a different and more abstract notation than Maude source code, in order to make it more readable for those who are not very familiar with Maude syntax. This will be followed by the description of the auxiliary functions and the optimizations we define. We finally discuss the adequacy of these optimizations. Our claim is that the side-effect-freeness of the compiler makes our set of optimizations adequate for substantial efficiency gain. However, there are cases where the code could be optimized further, but the set of optimizations is not capable of doing that. We discuss these cases and the limits of the optimizations in Chapter 7.

5.1 Code Visitors

To do source-level optimizations, one should traverse over the program, first collecting some information (i.e. analyzing the code), and then transforming the code so that it is (or at least expected to be) faster than the original code. In this sense, we distinguish two types of code traversers: analyzers and transformers.[1] In this section we explain how to write code traversers (i.e. visitors [10]) for Mumbo. As always, the implementation is done in Maude.

By means of equational logic and matching, Maude makes it easy to traverse the syntax tree of a program. To define these traversals easily, we first define an operation called `visit`:

```
op visit : MRewRule MExp MRewData -> MExpListDataPair .
```

Given an expression, which is actually a tree, we define this operation to traverse over every node of the expression. The first argument to this operation is the name of the traversal. e.g: `inline, unroll, genKill`, etc. The second argument is the expression we want to traverse. In other words, it is the root of the (sub)tree. The last argument is data. We use this argument to pass any information between nodes.

The `visit` operation returns a pair of an expression and data as its result. In the case of transforming visitors, the returned expression can be used to replace the original expression. The analyzers use the data part to report the information collected over the code. For this reason, the `visit` operation is generic enough to be used both to transform the tree, and to analyze it. It works as the skeleton for analyzers and transformers.

We define the default behaviour of traversal in the `IDENTITY` module. For each syntactic component we define `visit` to apply recursively on the sub-components of the node. By default it traverses the tree in a top-down, left-to-right order without changing anything in the tree (hence the name `IDENTITY`). For instance, below is the equational definition for addition:

[1] In [31], three types are identified: accumulators, transformers and accumulating transformers. We view accumulating transformers as transformers.

```
var R : MRewRule . vars E E' E1 E2 : MExp . vars D D1 D2 : MRewData .
ceq visit(R, E ++ E', D) = {E1 ++ E2, D2}
  if {E1, D1} := visit(R, E, D)
  /\ {E2, D2} := visit(R, E', D1) [otherwise] .
```

In this conditional equation, `R` is a variable of sort `MRewRule`; `E, E', E1, E2` are of sort `MExp`, and `D, D1, D2` are of sort `MRewData`. We would like to note that they are variables in the *mathematical sense*, not like the variables in an imperative programming language. The conditional equation states that for any visitor `R`, traversing a summation, we first visit the left operand with the same visitor, `R`. We pass this traversal the incoming data. We then traverse the right operand, but this time we pass the information we obtained from the visit to the left operand. This data may be different from the original. Visiting the right operand returns us a possibly different expression and data. We return the final data obtained from this traversal and return the summation of two new operands, as the result of the traversal. The default behaviour of `visit` is defined in the same manner for other syntactic units.

Since we implemented the default traversal behaviour on nodes, when we want to have a visitor doing some particular work on the tree, we just need to define its behaviour for the nodes we are interested in. This is similar to the traversal functions of [31] and the Visitor pattern of [10]. The default behaviour has the attribute `[otherwise]`, which ensures that it will be applied only if no other equation matches.

We now give a simple example illustrating how to define a visitor. Consider a traverser whose task is to replace a specified expression with another expression. Let's call this traverser as `replace`. We also define `replaceData`, which contains two expressions. The first is the expression that is to be replaced, and the second is the expression that will replace the original one. We only need to define the equation below, for this traverser to work.

```
eq visit(replace, E, replaceData(E, E')) = {E', replaceData(E, E')} .
```

So when we're traversing the expression we would like to replace, Maude matches it and applies the equation above. For other expressions, the identity function is applied. Consider this expression e:

```
if 'x equals 0 then set 'x = 10 else send self 'foo('x)
```

We may call the `replace` traverser on this expression as follows:

```
visit(replace,
      if 'x equals 0 then set 'x = 10 else send self 'foo('x),
      replaceData('x, 'y))
```

It reduces to the following tuple, replacing all the occurrences of `'x` with `'y`.

```
{if 'y equals 0 then set 'y = 10 else send self 'foo('y),
 replaceData('x, 'y)}
```

The transformed expression returned in this tuple can then be used instead of the old expression.

Storing information about nodes

The optimizers require information about the program to be able to transform it. For example, use-def analysis is required to propagate constants. This makes it necessary to store information on nodes. If we were doing the implementation in an object-oriented language, we could easily add new fields to the classes representing nodes, and those fields could store the information for us. But in Maude, we cannot do that. For this reason we define a new syntax tree node:

```
op info : JExp JInfoData -> JExp .
```

`info` can be seen as a closure, which contains an expression and some information about the expression. It is used to replace the expression it contains. This way, we are able store information along with the expressions. Various types of information, such as use-def, gen-kill and type information, can be stored in `info`. An analyzer, when traversing the tree, simply updates the information on the visited node, and continues its traversal. Another visitor, later on, can use or change this information. For instance, the `tag` analyzer assigns unique numbers to the loops in the program. We then use these tags to distinguish loops when unrolling. We define the following conditional equation in the `TAG` module:

```
var InfD : JInfoData . var N : Nat .
vars E BE E' BE' : JExp . vars D1 D2 : JRewData .
ceq visit(tag, info(do E while BE, InfD), tagData(N)) =
              {info(do E' while BE', InfD tag(N)), D2}
 if {E', D1} := visit(tag, E, tagData(N + 1))
 /\ {BE', D2} := visit(tag, BE, D1) .
```

This equation matches a loop, takes the current tag from the incoming data, increments the tag and recursively traverses the body and the condition, and finally leaves the tag as part of the stored information.

5.2 Analyses

We implement several analyzers which are necessary to do optimizations. These analyzers are defined using the `visit` operation as explained in Section 5.1. These are well-known and well-documented source-level analyses ([1, 20]). For this reason, we simply give the list of analyses without going into details.

- **INFO:** Encapsulates expressions in `info` packages, as explained in Section 5.1.
- **TAG:** Assigns unique numbers to names, loops and method calls. The tags are used for identification when unrolling loops, inlining methods and in use-def analysis.
- **UNTAG:** Removes tags.
- **TYPE:** Propagates type information over expressions. Narrows types when possible.
- **ISLOCAL:** Determines if a variable is local to the enclosing method, or a field.
- **ISFINAL:** Determines if the accessed field is final.
- **GEN-KILL:** Computes gen-kill sets of expressions. See [1] for a detailed explanation of gen-kill sets, why they are useful, and for the algorithm to compute them.
- **UD:** Computes use-def chains. It can do *may* or *must* analysis. See [1] for a more detailed discussion of use-def chains.

5.3 Auxiliary Functions

In this section we give the list of auxiliary functions, which are used in the optimizations described in Sections 5.5 and 5.6.

- $getMustDef(v)$: Finds the dominating definition of the variable v.
- $definable(e)$: An expression e is said to be *definable* if all the subexpressions it contains are definable. A variable is definable if its definition can be found in the method we are working on and if the right-hand-side of this definition is also definable. An instance creation, a literal, `self`, `NIL`, `GENSYM` and `GENLABEL` are always said to be definable.

  Example: Think of a variable, which is a parameter of the current method. Unless this variable has a definable definition in the method, it is non-definable because its definition is not accessible from inside the method.
- $mayMove(v, e)$: Returns true if the variable v can safely be moved to the place of the expression e.
- $mayHaveUse(v)$: Determines if the variable v may have a use in the program.
- $isLiteral(e)$: Returns true if the expression e is literal.
- $findFieldValue(x, c, el)$: Given a field name, x, a class name, c, and a list of arguments, el, which is passed to the instantiation of an object of c, this function looks into the `'initialize` method of the class and tries to find the value assigned to the field x. This value is expected to be a parameter of the method, which corresponds to an expression e in el. This function then assigns e to a fresh variable s, passes it to the new instance creation instead of e, and returns it so that accesses to the field x can now be replaced by s.
- $replace(e_1, e_2)$: Replaces the expression e_1 with the expression e_2. This is useful especially in α-conversion.

- $freeVars(e)$: Finds the free variables in the method body e. These free variables are the fields of the class the method is defined in.

- $vars(e)$: Finds all the variables occurring in the expression e.

- $collectDefs(e)$: Collects all the definitions (i.e. variable assignments) found in e.

- $sideEffectFree(e)$: Returns true if the expression e is known to have no side-effects. Literals, names, `self`, `NIL`, arithmetic, boolean operations and concatenation of side-effect-free operands are all side-effect-free. (`cast` f `to` x) and (f`->`x) are side-effect free if f is side-effect free. $sideEffectFree$ is conservative, because, without analyzing the method, every method invocation is considered side-effectful.

- $findFieldType(x, c)$: Returns the declared type of the field x of class c.

- $isFinal(m)$: Returns true if the method m is final, false otherwise.

- $recursive(m)$: Returns true if the method m is directly recursive, otherwise returns false.

- $getMethod(m, c)$: Given a method name, m, and a class name, c, returns the code of the method.

- $isSafe(m, c)$: Returns true if the method m of the class c does not change the state of its parameters. The method is said to be unsafe if it invokes a method or accesses a field of a parameter.

- $flatten(el)$: Given el, a comma-separated list of expressions (i.e. an argument list), converts it to a semi-colon-separated list of block-expressions; e.g. `(1, 'x)` becomes `{1 ; 'x}`.

- $typeof(e)$: Returns the static type of the expression e. This type is found as a result of TYPE analysis, and it may be narrower than the declared type.

5.4 Notation

All the optimization-related transformations we give in this chapter are implemented in Maude. This may make them hard to read if one is not very familiar with Maude syntax. We also want to abstract out some details. For these reasons, we use another notation to explain them. A transformation like

$[\![x]\!] \Rightarrow [\![y]\!]$

if $\tau =$ copyAssignment

$\bigwedge isLocal(x)$

$\bigwedge y \leftarrow getMustDef(x)$

$\bigwedge isLocal(y)$

$\bigwedge mayMove(x, y)$

should read:

> A variable x is replaced by a variable y, if the visitor τ is copyAssignment and x is local and the dominating definition of x is the variable y and y is local and y may be moved to the place of x.

We use $\leftarrow$ as the matching operator in the sense that $lhs \leftarrow rhs$ requires rhs to be matched to lhs. The variables we use are listed below:

$x,\ y,\ target,\ rt,\ f,\ c \in$ Name

$i,\ i_k \in$ Integer

$e,\ e',\ e_k,\ be,\ body,\ e_r,\ e_\alpha,\ e_f \in$ Expression

$e_b,\ e'_b,\ e''_b \in$ Block-expression (i.e. a single expression or a semicolon-separated list of expressions)

$el \in$ Expression list (comma-separated)

$lhs \in$ Left-hand-side (i.e. a name or a field access)

$m \in$ Syntactic representation of a method

$pl,\ pl' \in$ Parameter list

$vd,\ vd' \in$ Variable declaration list

5.5 Transformations not requiring analyses

In this section we define optimizations which do not need analysis results. These optimizations can take place without any out-of-context information. Because of its built-in matching features, it is trivial to define these in Maude.

- Remove useless braces:

  $[\![\{e\}]\!] \Rightarrow [\![e]\!]$

  $[\![\{e_b \ ; \ \{e'_b\}\}]\!] \Rightarrow [\![\{e_b \ ; \ e'_b\}]\!]$

  $[\![\{\{e_b\} \ ; \ e'_b\}]\!] \Rightarrow [\![\{e_b \ ; \ e'_b\}]\!]$

  $[\![\{e_b \ ; \ \{e'_b\} \ ; \ e''_b\}]\!] \Rightarrow [\![\{e_b \ ; \ e'_b \ ; \ e''_b\}]\!]$

- Reduce useless statements:

  $[\![\texttt{if } be \texttt{ then } e \texttt{ else } e' \ ; \ e_b]\!] \Rightarrow [\![be \ ; \ e_b]\!]$
  if $sideEffectFree(e) \bigwedge sideEffectFree(e')$

  $[\![e \ ; \ e_b]\!] \Rightarrow [\![e_b]\!]$
  if $sideEffectFree(e)$

  $[\![\texttt{cast } e \texttt{ to } x]\!] \Rightarrow [\![e]\!]$ [2]
  if $typeof(e) = x$

  $[\![\texttt{if True then } e_1 \texttt{ else } e_2]\!] \Rightarrow [\![e_1]\!]$

  $[\![\texttt{if False then } e_1 \texttt{ else } e_2]\!] \Rightarrow [\![e_2]\!]$

  $[\![\texttt{if } be \texttt{ then } e \texttt{ else } e]\!] \Rightarrow [\![\{be \ ; \ e\}]\!]$

[2] Note that this reduction requires type information. Since TYPE analysis computes this information and stores it as part of the `info` expression, it can be directly accessed. Hence, we give this reduction among the transformations which do not require out-of-context information.

$[\![\texttt{do}\ e\ \texttt{while False}]\!] \Rightarrow [\![e]\!]$

- Constant reduction:

  $[\![i_1 {+}{+} i_2]\!] \Rightarrow [\![i]\!]$, where i is the sum of i_1 and i_2

  $[\![\texttt{not(True)}]\!] \Rightarrow [\![\texttt{False}]\!]$

  etc.

- Simplify expressions with blocks:

  $[\![\texttt{set}\ lhs\ =\ \{e_b\ ; e\}]\!] \Rightarrow [\![\ \{e_b\ ;\ \texttt{set}\ lhs = e\}\ ]\!]$

  $[\![\texttt{send}\ \{e_b\ ; e\}\ x\ el]\!] \Rightarrow [\![\ \{e_b\ ;\ \texttt{send}\ e\ x\ el\}\ ]\!]$

  $[\![\texttt{send}\ (\texttt{if}\ be\ \texttt{then}\ e\ \texttt{else}\ e')\ x\ el]\!]$
  $\Rightarrow [\![\texttt{if}\ be\ \texttt{then}\ (\texttt{send}\ e\ x\ el)\ \texttt{else}\ (\texttt{send}\ e'\ x\ el)]\!]$

5.6 Transformations requiring analyses

In this section we give more complicated optimizations. These transformations need analysis results, such as use-def information, in order to function. Therefore we first pass analysis traversers over the code, accumulate information, and then run these optimizations.

- **Inline:** It inlines a method if it is guaranteed that it is the particular method being invoked. We do not inline non-final methods because polymorphism may change the exact method being called. We also do not inline a method if it is recursive and the target or at least one of the arguments is not *definable* (as defined in Section 5.3). This heuristic prevents us from falling into infinite loops when inlining methods automatically as part of the **Auto** optimizer (discussed in Section 5.7).

  In order to avoid name capture, we rename local variables and parameters of the method we are inlining.

$[\![\texttt{send}\ e\ x\ el]\!] \ \Rightarrow \ [\![e']\!]$

if $\tau = \mathsf{Inline}$

$\bigwedge m \leftarrow getMethod(x, typeof(e))$

$\bigwedge isFinal(m)$

$\bigwedge(definable(e, el) \bigvee \neg recursive(m))$

$\bigwedge e' \leftarrow performInlining(e, el, m)$

where $performInlining$ is defined as follows:

$performInlining(e, el, m) \ \Rightarrow \ \{\texttt{set}\ target = e\ ;\ e'\ \}$

if $target \leftarrow freshSym()$

$\bigwedge m = \texttt{method}\ x(pl)[vd]\ rt : body$

$\bigwedge e_r \leftarrow replace(\texttt{self}, target)$ in $body$

$\bigwedge\langle e_\alpha : pl'\rangle \leftarrow$ do α conversion for pl and vd in e_r

$\bigwedge e_f \leftarrow$ for each variable f in $freeVars(e_\alpha),\ replace(f, target\text{->}f)$

$\bigwedge e' \leftarrow \{\ argAssignments(pl', el)\ ;\ e_f\ \}$

where $argAssigments$ is the following recursive function:

$argAssignments((p, pl), (e, el)) \ \Rightarrow \ \{\ \texttt{set}\ p = e\ ; argAssignments(pl, el)\ \}$

$argAssignments((), ()) \ \Rightarrow \ \{\}$

- **Unroll:** It unfolds a do-while loop once. The loop becomes nested in an if-statement.

  $[\![\texttt{do}\ e\ \texttt{while}\ be]\!] \ \Rightarrow \ [\![\{e\ ;\ \texttt{if}\ be\ \texttt{then do}\ e\ \texttt{while}\ be\ \texttt{else}\ 0\}]\!]$

  if $\tau = \mathsf{Unroll}$

- **Copy Assignment:** If a variable is assigned to another variable, as in $\texttt{set}\ x\ \texttt{=}\ y$, this optimization replaces uses of x with y, if they are dominated by this definition and if it is safe to move y.

  $[\![x]\!] \ \Rightarrow \ [\![y]\!]$

  if $\tau = \mathsf{CopyAssignment}$

$\bigwedge isLocal(x)$

$\bigwedge y \leftarrow getMustDef(x)$

$\bigwedge isLocal(y)$

$\bigwedge mayMove(x, y)$

- **Constant Propagation:** If a variable is assigned a literal, this optimization replaces the uses of that variable with the assigned literal.

  $[\![x]\!] \Rightarrow [\![e]\!]$

  if $\tau =$ ConstantPropagation

  $\bigwedge isLocal(x)$

  $\bigwedge e \leftarrow getMustDef(x)$

  $\bigwedge isLiteral(e)$

- **Nil-Check:** It reduces expressions of the form (e `equals NIL`) to (`False`), if it can prove that e is not `NIL`.

  $[\![e$ `equals NIL`$]\!] \Rightarrow [\![$`False`$]\!]$

  if $\tau =$ NilCheck

  $\bigwedge$ `new` $x\ el = getMustDef(e)$

- **Useless Definition:** It removes a definition if it has no use. The assigned expression is not removed because it may have side-effects. If not, it will be removed by other optimizations.

  $[\![$`set` $lhs\ =\ e]\!] \Rightarrow [\![e]\!]$

  if $\tau =$ UselessDef

  $\bigwedge \neg(mayHaveUse(lhs))$

- **Useless Declaration:** It removes a variable declaration from the method header if that variable is never used in the method body.

  $[\![$`method` $x(pl)[vd]\ rt\ :\ e]\!] \Rightarrow [\![$`method` $x(pl)[vd']\ rt\ :\ e]\!]$

  if $\tau =$ UselessDecl

  $\bigwedge vd' \leftarrow (vd\ \cap\ vars(e))$

- **Useless New:** It removes an instance creation if it is useless and side-effect-free. It leaves the arguments of the instantiation because they may have side-effects. If not, they will be removed by other optimizations.

  $[\![\texttt{new}\ x\ el\ ;\ e_b]\!] \ \Rightarrow\ [\![flatten(el)\ ;\ e_b]\!]$

  if $\tau =$ UselessNew

  $\bigwedge isSafe(\text{`}\texttt{initialize}, x)$

  Note that we remove an instance creation only if there exist other statements after that. Otherwise we could be, for instance, changing the return value of a method, which would invalidate our program.

- **Field-value:** It extracts the value of a final field from an object and replaces accesses to that field by that value. For instance, suppose we have a piece of code like

  ```
  set 'a = new 'A(3) ; 'a -> 'f
  ```

  where class `'A` is defined as follows:

  ```
  class 'A extends 'object
     final field 'f : 'int
     method 'initialize('i : 'int) 'int : set 'f = 'i
  ```

  This optimization then transforms the code piece above to the following:

  ```
  set 'a = new 'A({ set 'sym = 3 ; 'sym}) ; 'sym
  ```

  If there are no more accesses to the field, then CopyAssignment and UselessDef will further optimize this code, giving

  ```
  set 'a = new 'A(3) ; 3
  ```

  This transformation is defined as follows:

  $[\![x\texttt{->}f]\!] \ \Rightarrow\ [\![e']\!]$

  if $\tau =$ FieldValue

$\bigwedge isFinal(f)$

$\bigwedge \texttt{new}\ c\ el = getMustDef(x)$

$\bigwedge e' \leftarrow findFieldValue(f, c, el)$

5.7 Putting the optimizers together

We define operations, which put the optimizations given above together, in order to make it easier and faster for the user to apply them on the program. In other words, these operations automate the optimization process. There are three of them:

Cleanup

This is a fixed-point iteration. It applies constantPropagation, copyAssignment, nilCheck, fieldValue, uselessDef, uselessNew and uselessDecl until the code does not change anymore. Since none of these optimizers is code-expanding, the cleanup loop is guaranteed to terminate.

Auto

This is another fixed-point iteration. It applies Inline, Unroll and **Cleanup**, until the code doesn't change. Since inlining and unrolling are code-expanding optimizations, there is the danger of non-termination. This is avoided by two heuristics: (1) Once a do-while loop is unfolded, it becomes nested in an if-statement. We do not further unroll loops which are nested in if-statements. (2) We do not inline a method if it is recursive and its target or at least one of the arguments is not definable. (This heuristic was explained in Section 5.3 and used in the Inline optimizer.) This heuristic may still cause infinite inlining in the general case, but it helps us avoid it because of the structure of the compiler.

Specialize Class

This is the most complicated optimizer. The need for **SpecializeClass** comes from the fact that Mumbo does not have anonymous classes. We first explain with an example the problem introduced by the lack of anonymous classes, and then give the solution. Consider the single code expression

```
$< 1 >$
```

The preprocessing phase converts the above code to the following expression:

```
new 'integerConstantCode(1)
```

If we had anonymous classes, we might have the following expression instead.

```
{ set 'value = 1 ; --- assuming that 'value is a fresh variable
  new 'Code(){     --- instantiating an anonymous class
    method 'eval('env : 'Env) ['sym : 'string, 'll : 'string] 'ClosedCode :
    {  set 'sym = GENSYM ;
       set 'll = 'sym # [" = add(0,"] # 'value # [") ; "] ;
       new 'ClosedCode('ll, 'sym)
    }
  }
}
```

In the body of the `'eval` method, we are referring to the variable `'value`, which keeps the value of the integer the code object is representing. We can propagate this value for optimization, having the following expression as the result. This new instance now specifically represents the integer `1`. Note that `'value` is removed as it becomes useless.

```
new 'Code(){
  method 'eval('env : 'Env) ['sym : 'string, 'll : 'string] 'ClosedCode :
  {  set 'sym = GENSYM ;
     set 'll = 'sym # [" = add(0,1) ; "] ;
     new 'ClosedCode('ll, 'sym)
  }
}
```

This way we can optimize the `'eval` method of every code object.

On the other hand, we cannot do the same with named classes because they are generic and it is not possible to define a named class at the place where a code object is

created. The obvious solution is to duplicate the class, specialize it and then create an instance of that new class instead of the original generic one.

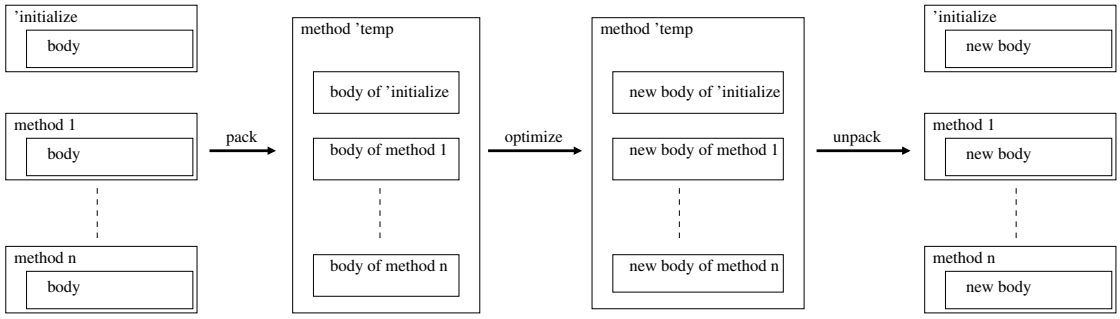

Figure 5.1: Specializing a duplicated class.

In Figure 5.1 we illustrate how to specialize a duplicated class. First, a temporary method is created. Then the bodies of the methods of the duplicated class are extracted, and these bodies — each of which is an expression — are composed such that the body of the `'initialize` method will precede the others. This is required because it will be invoked before all the other methods. The order in which the bodies of the other methods is composed does not matter, because the compiler is written in functional-style — all the fields of an object are final. This means that the methods cannot change the state of the object.

While putting the method bodies together, we perform α-conversion on them to avoid incorrect name capture. We use markers to distinguish method bodies from each other. At this stage, the method is ready to optimize. The **Auto** optimizer is run for optimization. The optimized temporary method is then decomposed to get the new method bodies. The new body of the `'initialize` method is examined to see if there is any need to add new fields or remove old fields that became useless. Finally, a new name for the duplicated class is generated and the instantiation of the code object is changed so that it will be an object of the new specialized class. Examples of the result of this process are given in Chapter 6.

5.8 Adequacy of Optimizations

The set of optimizations we presented in this chapter are adequate to achieve substantial optimization. The adequacy is due to the fact that the compiler is *side-effect-free*. Consider this: Inside the `'eval` method of a `'Code` class, there is a call to the `'eval` method of a subfragment (compositionality!). The subfragment's exact type cannot be known at this time, but when we are given the source code of a program generator, we can reason about the exact type of a fragment, unless it is a hole. Using this information, we inline the invoked `'eval`. In the inlined code, there will possibly be references to the fields of the subfragment. At this point we take advantage of the *side-effect-freeness* of the compiler: The accessed fields are all final. We can easily reason about their values, using the field-value transformer, because the instantiation of the subfragment occurs within the code. We use FieldValue optimization to extract values of the fields. Once this is done, the subfragment becomes useless — because all the information it would carry is inlined — and we can safely remove its creation via UselessNew transformation. If we didn't have the side-effect-free property, we would need more complicated analyses to reason about the values of fields, such as escape analysis or alias-analysis.

In Chapter 6, we give examples illustrating the level of optimization we can achieve. These examples support our claim of adequacy. However, in some cases, we cannot obtain the optimal code. We discuss the limitations of our transformations and analyses in Chapter 7.

Chapter 6

Performance

One of the strongest ideas behind using run-time program generation is that more efficient programs can be generated according to the information obtained at run-time. In other words, we use RTPG to specialize a program depending on the run-time properties. The efficiency of the overall program depends on the efficiency of the generated program compared to the static program, and the time spent during generation. The latter constraint, that is, the run-time generation cost, is paid off as the generated program, which is presumably faster, is used. If the generated program will be used just once, it is very likely that it will not pay off. We typically need to use the generated program several times in order for it to compensate its run-time generation cost, and to start bringing advantage in the overall execution time. The time where the generating program starts to be faster than the original, non-program generating version, is called the *crossover point*.

We would like to have a low crossover point to get more advantage of RTPG. There are mainly two ways to do that. First is to increase the efficiency of the generated program. This mostly depends on the character of the program being specialized and the experience of the programmer. We will not discuss this issue in the thesis. The second way is to generate the program more efficiently, in order to decrease the run-time generation cost. This is the focus of this chapter.

We have explained the Mumbo compiler in Chapter 4. Recall that it is a compositional compiler written in a traditional way. It does not know anything about RTPG. It is generic, in the sense that, given the environment, it compiles any program fragment without making any assumptions on the specifics of the fragment. To make it more concrete, think of the following code piece:

```
'x ++ 2
```

From the compiler's point of view, this is a binary operation of two operands. It does not know that the left operand is a variable expression, and that the right operand is an integer constant. On the other hand, by using the information that this code fragment gives us, we could directly produce `LowLevel` code like (in pseudo code):

```
if('x is local)
  'left = emit code to get local variable 'x
else if('x is a field)
  'left = emit code to access field 'x
else
  error

'right = emit code for constant 2
'result = emit code to add 'left to 'right
```

It is possible to produce this code for the program fragment given, by *partially evaluating* the back-end of the Mumbo compiler. Our ability to obtain this kind of a code depends on how much information we have on the piece of code we are working on. We would like the fragment to be as *complete* as possible. Free-variables and holes occurring in a fragment increase its incompleteness, and make it more difficult to reason about the fragment. For instance, consider the example above again. If we knew that `'x` was a local variable, we could get rid of the (`if...else if...else...`) part, and directly emit code to access local variable `'x`. So, having more information gives more opportunity for optimization. However, for a quoted-code like `$<^(c)>$`, we cannot do anything, because no information is available at this level.

To partially evaluate the compiler for certain code fragments, we apply the optimizations from Chapter 5. We use **SpecializeClass** to produce special versions of code

classes for those fragments. In the examples we provide, there are pieces of quoted code. For the optimized versions, we have optimized all these pieces separately, while for the unoptimized version, we have optimized none of them. We then compiled these versions using the Mumbo compiler. This gives us two programs in `LowLevel` code. We then executed the two versions, using the executable semantics of the `LowLevel` language. Maude automatically reports the number of rewrites it applied. We report this number as the performance of a program. That is, the comparison of the efficiency of two programs is based on the number of rewrites applied to execute the programs.

In this chapter we first give some simple examples to see if we can get code that looks like the one we discussed above. We then give two classic run-time program generation examples: loop unrolling and exponentiation. There also are some cases in which we cannot obtain the ideal result. We discuss them in Chapter 7.

6.1 The First Examples

In this section we provide some simple examples to understand what the result of optimization looks like, and what we can expect to see. The examples include optimization of a complete class definition, classes with holes, methods with holes and simple expressions.

6.1.1 A Complete Class

As the first example, we optimize the following Mumbo program. It has the definition of a complete class between brackets. This can be seen as the ideal case, because we have the whole information for the class. This is probably a non-realistic example —one would not compile a class at run-time if all the information of the class exists before execution — but it is a good point to start with. Since the class is complete, we would expect to optimize the `'eval` method of the object representing the class code, so that it would directly return the `LowLevel` code.

```
final class 'ClassGen extends 'object
  method 'completeClass() 'Code :
```

```
    $< class 'Gen extends 'object
         field 'x : 'int
         method 'bar() 'int :
         {  set 'x = 12 ;
            'x ** 2
         }
    >$
  method 'test() 'string :
    send (send self 'completeClass()) 'generate()
main [noVars]
  send (new 'ClassGen()) 'test()
```

After applying the **SpecializeClass** optimization, we get the following program:

```
final class 'ClassGen extends 'object
  method 'completeClass() [noVars] 'Code :
    send new 'LinkedList(NIL,NIL)  'add(new 'defineClassCode-15() )
  method 'test() [noVars] 'string :
    send send self 'completeClass()  'generate()
main [noVars]
  send (new 'ClassGen()) 'test()
```

The `'eval` method of `'defineClassCode-15`, which is a new, specialized class, is below:

```
method 'eval('env_18 : 'Env)
['sym_6661 : 'string, 'sym_12911 : 'string, 'sym_14091 : 'string,
 'lowlevel_14090 : 'string, 'tempSym_15757 : 'string, 'temp_15756 : 'string,
 'lowlevel_14095 : 'string, 'lowlevel_12908 : 'string, 'sym_15765 : 'string,
 'sym_16955 : 'string, 'tempSym_18812 : 'string, 'temp_18811 : 'string,
 'lowlevel_16954 : 'string, 'sym_16960 : 'string, 'lowlevel_16959 : 'string,
 'lowlevel_15762 : 'string, 'lowlevel_6659 : 'string,
 'lowlevel_4739 : 'string
] 'ClosedCode :
{
  set 'sym_6661 = GENSYM ;
  set 'sym_12911 = GENSYM ;
  set 'sym_14091 = GENSYM ;
  set 'lowlevel_14090 = 'sym_14091 # [" = add(0,12) ; "] ;
  set 'tempSym_15757 = GENSYM ;
  set 'temp_15756 = 'tempSym_15757 # [" = getelementptr(% 'self, 'x) ; "] ;
  set 'lowlevel_14095 = 'temp_15756 # ["store("] # 'sym_14091 # [", "]
                      # 'tempSym_15757 # [") ; "] ;
  set 'lowlevel_12908 = 'lowlevel_14090 # 'lowlevel_14095 ;
  set 'lowlevel_12908 = 'lowlevel_12908 # 'sym_12911 # [" = add(1,0) ; "] ;
  set 'sym_15765 = GENSYM ;
```

```
  set 'sym_16955 = GENSYM ;
  set 'tempSym_18812 = GENSYM ;
  set 'temp_18811 = 'tempSym_18812 # [" = getelementptr(% 'self, 'x) ; "] ;
  set 'lowlevel_16954 = 'temp_18811 # 'sym_16955
                      # [" = load("] # 'tempSym_18812 # [") ; "] ;
  set 'sym_16960 = GENSYM ;
  set 'lowlevel_16959 = 'sym_16960 # [" = add(0,2) ; "] ;
  set 'lowlevel_15762 = 'lowlevel_16954 # 'lowlevel_16959
                      # 'sym_15765 # [" = mul("] # 'sym_16955
                      # [", "] # 'sym_16960 # [") ; "] ;
  set 'lowlevel_6659 = 'lowlevel_12908 # 'lowlevel_15762 ;
  set 'lowlevel_6659 = 'lowlevel_6659 # 'sym_6661
                     # [" = copy("] # 'sym_15765 # [") ; "] ;
  set 'lowlevel_4739 = ["'Gen $ 'bar (% 'self ) { 'entry : "]
                     # 'lowlevel_6659 # [" br 'end ; 'end : return "]
                     # 'sym_6661 # [" ; } "] ;
  new 'ClosedCode('lowlevel_4739, [" $NONAME-FOR-CLASSES$ "] )
}
```

This method directly produces the `LowLevel` code. This is exactly what we were expecting. (We cannot do anything about `GENSYM` expressions at the source-level, because they return a unique string at each evaluation.)

The original program is executed in 26351 rewrites, and the optimized version in 7737. This is a 70% speedup in run-time generation.

6.1.2 A Class with Holes

In the second test we have a class with two holes. The first hole is in place of a field, and the second is in place of a method body. The code is given below.

```
final class 'ClassGen extends 'object
  method 'incompleteClass('body : 'Code, 'f : 'Code) 'Code :
    $< class 'Gen extends 'object
         ^F('f)
         method 'bar() 'int :
           ^('body)
    >$
  method 'field() 'Code :
    $< field 'x : 'int >$
  method 'methodBody() 'Code :
    $< { set 'x = 3 ; 'x ** 2 } >$
```

```
  method 'test() 'string :
    send (send self 'incompleteClass(send self 'methodBody(),
                                     send self 'field())) 'generate()
main [noVars]
  send (new 'ClassGen()) 'test()
```

As mentioned earlier, the code objects are optimized separately. This gives us the following version of the program:

```
final class 'ClassGen extends 'object
  method 'field() [noVars] 'Code :
    new 'defineFieldCode-10280()
  method 'incompleteClass('body : 'Code,'f : 'Code) [noVars] 'Code :
    send new 'LinkedList(NIL,NIL)  'add(new 'defineClassCode-15('f,'body) )
  method 'methodBody() [noVars] 'Code :
    new 'statementsCode-10523()
  method 'test() [noVars] 'string :
    send send self 'incompleteClass(send self 'methodBody(),
                                    send self 'field() )  'generate()
```

The `'eval` method of `defineClassCode-15` now looks like

```
method 'eval('env_20 : 'Env)
  ['bval_4600 : 'ClosedCode, 'lowlevel_4602 : 'string, 'sym6443 : 'Env,
   'sym853 : 'Env, 'sym858 : 'Env, 'sym8968 : 'string, 'tempEnv_856 : 'Env]
  'ClosedCode :
{ set 'sym858 = 'env_20 ;
  set 'sym853 = new 'Env(new 'LinkedList(NIL,NIL),
                         new 'LinkedList(NIL,NIL),
                         ['Gen],
                         ['object],
                         'sym858 -> 'classInfos) ;
  --- add the field to the environment
  set 'tempEnv_856 = send 'v_846 'addToEnv('sym853) ;
  set 'sym6443 = new 'Env(new 'LinkedList(NIL,NIL),
                          'tempEnv_856 -> 'fieldList,
                          { set 'sym8968 = 'tempEnv_856 -> 'currClass ;
                            'sym8968 },
                          'tempEnv_856 -> 'superClass,
                          'tempEnv_856 -> 'classInfos) ;
  set 'lowlevel_4602 = 'sym8968 # [" $ 'bar (% 'self"] ;
  set 'lowlevel_4602 = 'lowlevel_4602 # [" ) { 'entry : "] ;
  --- evaluate body of the method
  set 'bval_4600 = send 'sym4791 'eval('sym6443) ;
  set 'lowlevel_4602 = 'lowlevel_4602 # 'bval_4600 -> 'lowlevelcode
```

```
                    # [" br 'end ; 'end : return "]
                    # 'bval_4600 -> 'name # [" ; } "] ;
  new 'ClosedCode('lowlevel_4602, [" $NONAME-FOR-CLASSES$ "] )
}
```

In this case, we do not know the field of the class and the body of the method. Therefore we cannot inline the methods which add the field to the environment and evaluate the body. This optimized version runs in 12486 rewrites, whereas the original program runs in 26525. (Recall that, we are also separately optimizing the quoted-codes representing the method body and the field.) This makes a 53% speedup. Since we have holes in this case, it was expectable that we get less speedup than the previous test with a complete class.

6.1.3 A Binary Expression

This time, we have a binary expression, multiplication, inside brackets. One of the operands of the operation is a hole, while the other one is a constant.

```
final class 'ClassGen extends 'object
  method 'stmtWrapper('x : 'Code) 'Code :
    $< ^('x) ** 10 >$
  method 'test() 'Code :
    send (send self 'stmtWrapper($< 'x >$)) 'generate()

main [noVars]
  send (new 'ClassGen()) 'test()
```

Sending the `'generate` message to a code object which is not a program or class doesn't normally make sense. But this time it is helpful to measure how much we optimize the `'eval` method of the code object representing the binary expression. We do not produce a specialized class for `$< 'x >$` appearing in the `'test` method, just to see the effect of optimization on the binary-expression code.

```
final class 'ClassGen extends 'object
  method 'stmtWrapper('x : 'Code) [noVars] 'Code :
    new 'binOpCode-3('x)
  method 'test() [noVars] 'Code :
```

```
    send send self 'stmtWrapper(new 'getNameCode( ['x] ) )  'generate()

final class 'binOpCode-3 extends 'Code
  final field 'l : 'Code
  method 'eval('env_5 : 'Env)
  ['lowlevel_226 : 'string,'lowlevel_6 : 'string,'lval_7 : 'ClosedCode,
   'sym_227 : 'string,'sym_9 : 'string] 'ClosedCode :
  { set 'sym_9 = GENSYM ;
    set 'lval_7 = send 'l 'eval('env_5) ;
    set 'sym_227 = GENSYM ;
    set 'lowlevel_226 = 'sym_227 # [" = add(0,10) ; "] ;
    set 'lowlevel_6 = 'lval_7 -> 'lowlevelcode # 'lowlevel_226 # 'sym_9
                    # [" = mul("] # 'lval_7 -> 'name
                    # [", "] # 'sym_227 # [") ; "] ;
    new 'ClosedCode('lowlevel_6,'sym_9)
  }
  method 'initialize('x_4 : 'Code) [noVars] 'int :
    set 'l = 'x_4

main [noVars]
  send (new 'ClassGen()) 'test()
```

Above is the code we get after optimization. In the original code of the `'binaryOpCode` class, we have fields to hold the left operand, the right operand and the name of the binary operation. In the optimized version note that we no longer have the latter two fields. This is because we propagated the information that the right operand of the operation was an integer constant, 10, and that the operation was multiplication. However, we cannot do anything about the left operand. We just keep the field and the call to its `'eval` method. This optimization brings 23% speedup (5307 rewrites for the original program and 4090 for the optimized version).

6.2 LoopUnrolling

After testing relatively simple cases, we now test a classical example of run-time program generation: loop unrolling. We have implemented three slightly different versions of unrolling. They are adapted from [13].

```
final class 'LoopUnroll extends 'object
```

```
  method 'context('b : 'Code) 'Code :
     $< class 'Printer extends 'object
          field 'x : 'int
          method 'initialize () 'int :
            ^('b)
      >$

  --- Unroll 1
  method 'test1('n : 'int) 'string :
    send (send self 'context(send self 'unroll1('n, $< 99 >$))) 'generate()
  method 'unroll1('n : 'int, 'c : 'Code) 'Code :
    if('n equals 0) then
      $< 0 >$
    else
      $< { ^('c) ; ^(send self 'unroll1('n -- 1, 'c)) } >$

  --- Unroll 2
  method 'test2('n : 'int) 'string :
    send (send self 'context(
          send self 'unroll2('n, new 'CodeFun()))) 'generate()
  method 'unroll2 ('n : 'int, 'F : 'CodeFun) 'Code :
    if ('n equals 0) then
      $< 0 >$
    else
      $< { ^(send self 'unroll2('n -- 1, 'F)) ;
           ^(send 'F 'iteration('n))
         } >$

  --- Unroll 3
  method 'test3('n : 'int) 'string :
    send (send self 'context(
          send self 'unroll3('n, $< 99 >$, $< 'x >$))) 'generate()
  method 'unroll3 ('n : 'int, 'c : 'Code, 'itervar : 'Code) 'Code :
    if ('n equals 0) then
      $< set ^('itervar) = 0 >$
    else
      $< { ^(send self 'unroll3('n -- 1, 'c, 'itervar)) ;
           set ^('itervar) = ^('itervar) ++ 1 ;
           ^('c) } >$

class 'CodeFun extends 'object
  method 'iteration('n : 'int) 'Code :
    $< ^I('n) >$

main [noVars]
  send (new 'LoopUnroll()) 'test1(1)
```

Recall that the compilation of every quoted-code in this program is optimized. We do not give those versions here and only provide the performance comparison. We execute `'testX` methods, with different numbers of iterations. The results are in Table 6.1. They show that even when the iteration number is 1, we get a speedup of 47% in the worst case.

Test	n	Original	Optimized	Speedup(%)
1	1	19639	10151	48
1	10	54784	24029	56
1	100	406234	162809	60
1	200	796734	317009	60
2	1	19805	10476	47
2	10	57020	27405	52
2	100	429170	196695	54
2	200	842670	384795	54
3	1	31509	16405	48
3	10	142938	67939	53
3	100	1257228	583279	54
3	200	2495328	1155879	54

Table 6.1: The effect of optimization in the loop-unrolling example.

6.3 Exponentiation

Another classical example of RTPG is exponentiation. A power function, which is specialized to a fixed exponent can be generated at run-time for better performance.

```
class 'ExpTest extends 'object
  method 'getExp('n : 'int) ['r : 'Code, 'i : 'int] 'Code :
  { set 'r = $< 1 >$ ;
    set 'i = 0 ;
    while(not('i equals 'n)) {
      set 'r = $< ^('r) ** 'x >$ ;
      set 'i = 'i ++ 1
    } ;

    $< class 'MyPower extends 'object
         method 'power('x : 'int) 'int :
```

```
            ^('r)
      >$
    }
    method 'test('k : 'int) 'string :
      send (send self 'getExp('k)) 'generate()

main [noVars]
  send (new 'ExpTest()) 'test(1)
```

Similar to the loop-unrolling example, we again run the program with several different values as the exponent. The results are given in Table 6.2.

Exponent	Original	Optimized	Speedup(%)
1	18649	9183	51
10	41086	20901	49
100	265456	138081	48

Table 6.2: The effect of optimization in the exponentiation example.

In this benchmark, a significant portion of the speedup may be obtained from the optimization of the quoted-code representing the generated class. To see how much speedup we can get, we ran another test, in which, only the code for the unfolded exponentiation is optimized and the enclosing class is not present. In this test, we call the `'eval` method of the optimized code. The results are given in Table 6.3. When the power is 1, we do not get as good performance as we obtained in the previous version, but the performance increases as the value of exponentiation increases.

Exponent	Original	Optimized	Speedup(%)
1	5031	3692	27
10	27468	15410	44
100	251838	132590	47

Table 6.3: The effect of optimization in the exponentiation example. Only the unfolded exponentiation is optimized.

Chapter 7

Further Discussions

In Chapter 6 we were able to get code which is specialized for a specific quoted-code. The resulting code we obtained looked like exactly what we would like to see. However, there are cases in which we cannot get the ideal code out of optimization. In this chapter we discuss such cases. We give two examples to illustrate them. In this chapter, when providing source code, we use pseudo code because it is sufficient for our purposes and it is more readable.

7.1 The Limit of Optimizations

In this section we work on an example for which the existing set of optimizations fails to produce the ideal code. Consider the following code:

```
$< 'x ** 'x >$
```

The ideal back-end for this code would be

```
if('x is local)
  'left   = emit code to get local variable 'x
  'right  = emit code to get local variable 'x
  'result = emit code to multiply 'left and 'right
else if('x is a field)
  'left   = emit code to access the field 'x
  'right  = emit code to access the field 'x
  'result = emit code to multiply 'left and 'right
```

```
else
  error
```

However, the code we produce looks like this:

```
if('x is local)
  'left = emit code to get local variable 'x
else if('x is a field)
  'left = emit code to access the field 'x
else
  error

if('x is local)
  'right = emit code to get local variable 'x
else if('x is a field)
  'right = emit code to access the field 'x
else
  error

'result = emit code to multiply 'left and 'right
```

This shows that, with the current set of optimizations, for instance, we cannot use the information that if `'x` is a local variable on the left side, then it is a local variable on the right side, too. With a transformation like the following, we would be able to achieve the ideal code.

$$
\begin{aligned}
&[\![\texttt{if } be \texttt{ then } e_1 \texttt{ else } e_2 \ ; \ eb \ ; \ \texttt{if } be \texttt{ then } e_3 \texttt{ else } e_4]\!] \\
&\Rightarrow \ [\![\texttt{if } be \texttt{ then } \{e_1 \ ; \ eb \ ; \ e_3\} \texttt{ else } \{e_2 \ ; \ eb \ ; \ e_4\}]\!] \\
&\textbf{if } sideEffectFree(be) \\
&\bigwedge sideEffectFree(e) \\
&\bigwedge sideEffectFree(e_1) \\
&\bigwedge sideEffectFree(e_2)
\end{aligned}
$$

This may seem easy at the first sight, but the situations we end up with usually require that we check method calls to see whether they are side-effect-free. This necessitates a difficult analysis of the program. This transformation also causes code explosion, especially if the duplicated code *eb* is large. We do not anticipate that the speed-up we would achieve with this kind of a transformation would be worth implementing it. Therefore we do not include it in our set of optimizations.

7.2 Adding Local Variable Declaration as an Expression

In Mumbo, all the local variables of a method are declared as part of the method header. Once we start evaluating the body the method, nothing can change the environment. Recall that in the compositional compiler, a fragment passes the environment when invoking the `'eval` method of its subfragment(s), but does not receive any environment as part of the tuple returned from the invocations. Consider the following code piece:

```
$< method 'm('a : 'int) ['b : 'int] 'int :
   { set 'b = 'f -- 'a ;
     ^('hole) ;
     'f ** 'b
   }
>$
```

The variables local to the method defined in this quoted-code are `'a` and `'b`. We know that `'f` is either a field or it is undeclared. (It is an error in the latter case. This is detected when we receive the environment to generate the `LowLevel` code.) No value of `'hole` can change this fact. However, in Jumbo, variable declarations can be passed as code pieces. Now assume that we have the ability of declaring variables in Mumbo, in any part of the method. This would make it possible that `'hole` declares `'f` as a local variable. In this case, we have a different situation: The first occurrence of `'f` is either a field, or it is undeclared. The second occurrence of `'f` is either a local variable, or a field, or it is undeclared. This limits our chances for optimization by introducing one more condition which we were able to eliminate previously.

If we introduce the ability to declare variables in the method body, Mumbo models Jumbo more closely. For this reason we rewrote the compiler. In the new version, `'ClosedCode` has an additional field, `'env`, to keep an environment. When a program fragment passes the received environment to one of its sub-fragments, it will get a (possibly) new environment back. This new environment is passed to the next sub-fragment. Thus, a code object representing a local variable declaration would add itself to the environment, and return the changed environment to its parent fragment. Actually, every

code object now has the chance to change the environment. This modification to the compiler makes it possible to introduce to the language not only variable declarations, but also anonymous classes (Mumbo does not have them). We, however, did not syntactically extend Mumbo with these features. Our purpose is to see how optimized code looks like when we give the code pieces the ability to change the environment.

With the new compiler, we have a situation similar to that presented in Section 7.1. The ideal optimized version of the compilation of the method body above would look like this:

```
--- the first line: set 'b = 'f --- 'a
if('f is a field in 'env)
  'left-1 = emit code to access the field 'f
else
  error

'right-1 = emit code to get local variable 'a
'sub = emit code to subtract 'right-1 from 'left-1
'lhs-1 = emit code to get pointer to local 'b
store 'sub to 'lhs-1

--- the second line: ^('hole)
'holeValue = send 'hole 'eval('env)
emit code of 'holeValue

--- the last line: 'f ** 'b
'newEnv = 'env of 'holeValue
if('f is local in 'newEnv)
  'left-2 = emit code to get local variable 'f
else if('f is a field in 'newEnv)
  'left-2 = emit code to access the field 'f
else
  error

'right-2 = emit code to get local variable 'b
'mul = emit code to multiply 'left-2 and 'right-2
```

In the multiplication, we can conclude that `'b` is a local variable, and emit code accordingly. This is valid because it was declared as a local variable, and by the semantics of local variable declaration, it will stay so. (If the `'hole` tries to declare it, it is an error,

and this will be detected inside the call to `'hole`'s eval method.) However, the optimized code looks like the following:

```
--- ... same as above ...
'newEnv = 'env of 'holeValue
if('f is local in 'newEnv)
  'left-2 = emit code to get local variable 'f
else if('f is a field in 'newEnv)
  'left-2 = emit code to access the field 'f
else
  error

if('b is local in 'newEnv)
  'right-2 = emit code to get local variable 'b
else if('f is a field in 'newEnv)
  'right-2 = emit code to access the field 'b
else
  error

'mul = emit code to multiply 'left-2 and 'right-2
```

With the current set of optimizations we cannot conclude that `'b` is still a local variable. The reason is that we need to check the scope of `'b` in the new environment returned from the evaluation of the hole. We do not know the dynamic type of `'hole`. Therefore its `'eval` method is virtual. At this level, we have no knowledge of what this method looks like. Hence, we cannot reason about the new environment returned, and the optimizers cannot reduce the expressions checking if `'b` is local or a field.

This limitation is more complicated than the one given in Section 7.1. In order to overcome it, the optimizers have to have knowledge of what the holes are capable of doing. There may also be other interesting situations. This is a subject of further research.

Chapter 8

Conclusion

In this thesis we have presented Mumbo, a model of Jumbo. Mumbo is a typed, object-oriented language supporting run-time program generation. It is a subset of Jumbo, which is an extension of Java featuring RTPG. Mumbo's implementation is based on equational logic. This is done in Maude, making it possible to execute Mumbo programs. Being a simplified version of Jumbo, Mumbo makes it easier to experiment with new ideas.

We have presented a set of optimizations and we have showed that by these optimizations, run-time generation can be optimized substantially. By developing heuristics based on the structure of the compiler, we have been able to automate the optimization process. These facts lead to some results that we are planning to apply to Jumbo. One of the most important properties of Mumbo compiler is its side-effect-freeness. We have shown that this property makes it more suitable to get higher speedups in run-time generation, and to achieve this more easily. It is often argued that functional programming is less efficient than imperative programming. This may be a case where it is more efficient.

The optimizations we presented are pretty standard and they have been adequate to obtain substantial amount of optimization. However, there exist some limitations of them. In some cases, we were close to, but could not reach optimal optimization. We discussed some of these cases in Chapter 7. Studying this issue remains as a subject to future research.

In Jumbo, anonymous classes are used extensively to instantiate `Code` objects. Anonymous classes may bring difficulties in some cases of program analysis and transformation. In Mumbo, we do not have anonymous classes. However, we were able to produce specialized versions of generic classes, as if they were anonymous. While doing that, we benefited from the fact that the compiler is implemented in side-effect-free style.

The consequences we have listed above motivate us to refactor Jumbo and convert it to side-effect-free style. This way we may be able to obtain high speedup in run-time generation, and whenever we have difficulties in analyzing anonymous classes, we can use methods similar to **SpecializeClass**.

Finally, defining Mumbo in Maude brought convenience especially in implementing the optimizations. We believe that this work can also be considered as an effective application of rule-based programming.

8.1 Related Work

In Section 5.1, we explained the `visit` operation, which, by defining the default traversal behaviour, allows us to introduce traversers easily; we need to give the visitor behaviour only for the nodes we are interested in. In object-oriented programming this can be done with Visitors [10]. For rule-based programming, traversal functions are introduced into the ASF+SDF system [31]. With a traversal function, the programmer selects a tree traversal behaviour for a function, and the default behaviour is provided automatically. In strategic programming, same result can be achieved via *strategies* [32].

In [23, 5] scoped dynamic rules are discussed. With these, it is possible to add and remove rules dynamically. The authors demonstrate how some optimizations like constant propagation, copy assignment can be defined using scoped dynamic rules. This is a powerful approach, and in some cases it reduces the number passes that should be done on the program being optimized. Furthermore, some analysis results (e.g: dominance) can be implicitly expressed by scoped dynamic rules, eroding the necessity to keep that analysis information. This decreases the load of the programmer implementing

the optimizations. The disadvantage of this approach is that scoped dynamic rules are more difficult to control and to reason about than usual equations. Therefore it is more difficult to design them.

Bibliography

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Aktemur, B., and Kamin, S. Mumbo: A Rule-based Implementation of a Runtime Program Generation Language. In *Proc. of the 6th Intl. Workshop on Rule-Based Programming* (April 2005). Nara, Japan.

[3] Attardi, G., Cisternino, A., and Kennedy, A. Codebricks: Code Fragments as Building Blocks. In *PEPM '03: Proc. of the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation* (2003), ACM Press, pp. 66–74.

[4] Bawden, A. Quasiquotation in Lisp. In *Partial Evaluation and Semantic-based Program Manipulation* (1999), pp. 4–12.

[5] Bravenboer, M., van Dam, A., Olmos, K., and Visser, E. Program Transformation with Scoped Dynamic Rewrite Rules. *Fundamenta Informaticae* (2005).

[6] Clausen, L. *Optimizations In Distributed Run-time Compilation*. PhD thesis, University of Illinois at Urbana-Champaign, 2004.

[7] Clavel, M., Durán, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J., and Talcott, C. *Maude Manual*, 2.1 ed. `http://maude.cs.uiuc.edu`, March 2004.

[8] Consel, C., Lawall, J. L., and Meur, A.-F. L. A Tour of Tempo: A Program Specializer for the C Language. *Sci. Comput. Program. 52*, 1-3 (2004), 341–370.

[9] Engler, D. R., Hsieh, W. C., and Kaashoek, M. F. 'C: A Language for High-level, Efficient, and Machine-independent Dynamic Code Generation. In *Proc. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '96)* (1996), ACM Press, pp. 131–144.

[10] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing, 1995.

[11] Gosling, J., Joy, B., and Steele, G. *The Java Language Definition*. Addison-Wesley, 1996.

[12] Grant, B., Mock, M., Philipose, M., Chambers, C., and Eggers, S. J. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science 248*, 1–2 (2000), 147–199.

[13] Kamin, S. Routine Run-time Code Generation. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)* (2003), ACM Press, pp. 208–220. Also appeared in: SIGPLAN Notices, vol. 38 (2003), pp. 44-56.

[14] Kamin, S. Program Generation Considered Easy. In *Proc. of the 2004 ACM SIGPLAN Symp. on Partial Evaluation and Semantics-based Program Manipulation (PEPM '04)* (2004), ACM Press, pp. 68–79.

[15] Kamin, S., Callahan, M., and Clausen, L. Lightweight and Generative Components-1: Source-Level Components. In *Proc. of the First Intl. Symp. on Generative and Component-Based Software Engineering (GCSE '99)* (2000), Springer-Verlag, pp. 49–64.

[16] Kamin, S., Callahan, M., and Clausen, L. Lightweight and Generative Components-2: Binary-Level Components. In *Proc. of the Intl. Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG '00)* (2000), Springer-Verlag, pp. 28–50.

[17] Kamin, S., Clausen, L., and Jarvis, A. Jumbo: Run-time Code Generation for Java and its Applications. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO '03)* (2003), IEEE Computer Society, pp. 48–56.

[18] Low Level Virtual Machine. `http://llvm.cs.uiuc.edu`.

[19] Meseguer, J., and Roşu, G. Rewriting Logic Semantics: From Language Specifications to Formal Analysis Tools. In *Lecture Notes in Computer Science vol. 3097* (June 2004), Springer, pp. 1–44.

[20] Muchnick, S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[21] Neverov, G., and Roe, P. Metaphor: A Multi-stage, Object-Oriented Programming Language. In *Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE 2004)* (Vancouver, Canada, October 2004), G. Karsai and E. Visser, Eds., vol. 3286 of *Lecture Notes in Computer Science*, Springer.

[22] Oiwa, Y., Masuhara, H., and Yonezawa, A. DynJava: Type Safe Dynamic Code Generation in Java. In *The 3rd JSSST Workshop on Programming and Programming Languages (PPL2001)* (March 2001).

[23] Olmos, K., and Visser, E. Composing Source-to-Source Data-Flow Transformations with Rewriting Strategies and Dependent Dynamic Rewrite Rules. In *14th Intl. Conf. on Compiler Construction (CC'05)* (Edinburgh, April 2005), R. Bodik, Ed.

[24] Plotkin, G. D. A Structural Approach to Operational Semantics. Tech. Rep. DAIMI FN-19, University of Aarhus, 1981.

[25] Poletto, M., Engler, D. R., and Kaashoek, M. F. tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation. In *Proc. of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI '97)* (1997), ACM Press, pp. 109–121.

[26] Poletto, M., Hsieh, W. C., Engler, D. R., and Kaashoek, M. F. 'C and tcc: A Language and Compiler for Dynamic Code Generation. *ACM Transactions on Programming Languages and Systems 21*, 2 (1999), 324–369.

[27] Roşu, G. CS422 Class Notes. `http://fsl.cs.uiuc.edu/~grosu/classes/2004/fall/cs422/`, 2004.

[28] Rountev, A. Precise Identification of Side-effect-free Methods in Java. In *IEEE International Conference on Software Maintenance* (2004), pp. 82–91.

[29] Taha, W., Calcagno, C., Leroy, X., and Pizzi, E. MetaOCaml. `http://www.metaocaml.org/`.

[30] Taha, W., and Sheard, T. MetaML and Multi-stage Programming with Explicit Annotations. *Theoretical Computer Science 248*, 1–2 (2000), 211–242.

[31] van den Brand, M. G. J., Klint, P., and Vinju, J. J. Term Rewriting with Traversal Functions. *ACM Trans. Softw. Eng. Methodol. 12*, 2 (2003), 152–190.

[32] Visser, E. A Survey of Rewriting Strategies in Program Transformation Systems. In *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)* (Utrecht, The Netherlands, May 2001), B. Gramlich and S. Lucas, Eds., vol. 57 of *Electronic Notes in Theoretical Computer Science*, Elsevier Science Publishers.

[33] Zook, D., Huang, S. S., and Smaragdakis, Y. Generating AspectJ Programs with Meta-AspectJ. In *Proc. of the Third Intl. Conf. on Generative Programming and Component Engineering (GPCE 2004)* (Vancouver, Canada, October 2004), G. Karsai and E. Visser, Eds., vol. 3286 of *Lecture Notes in Computer Science*, Springer, pp. 1–18.

Appendix A

Source Code of Mumbo Compiler

```
fmod M-COMPILER is extending MUMBO-SYNTAX .
  op MCompiler : -> MClasses .
  eq MCompiler = (

final class 'getNameCode extends 'Code
  final field 'name : 'string

  method 'initialize('n : 'string) 'int : set 'name = 'n

  method 'eval('env : 'Env)
  ['sym : 'string, 'lowlevel : 'string, 'temp : 'ClosedCode] 'ClosedCode :
  {
    if (send 'env 'isLocal('name)) --- either a local var or method parameter
    then
    {
      set 'sym = ["% "] # 'name ;
      set 'lowlevel = [""]
    }
    else if (send 'env 'isField('name)) then
         { set 'sym = (GENSYM) ;
           set 'temp = send self 'getPtr() ;
           set 'lowlevel = send 'temp 'getCode()
                         # 'sym # [" = load("] # (send 'temp 'getName())
                         # [")"] # [" ; "]
         }
         else
         { set 'sym = [" $CANNOT RESOLVE:$ "] # 'name ;
           set 'lowlevel = [""]
         } ;
    new 'ClosedCode('lowlevel, 'sym)
  }

  method 'evalLHS('env : 'Env, 'rhs : 'string)
```

```
  ['lowlevel : 'string, 'temp : 'ClosedCode] 'ClosedCode :
  {
    if (send 'env 'isLocal('name)) --- either a local var or method parameter
    then
    {
      set 'lowlevel = ["% "] # 'name # [" = copy("] # 'rhs # [") ; "]
    }
    else if (send 'env 'isField('name)) then
         { set 'temp = send self 'getPtr() ;
           set 'lowlevel = send 'temp 'getCode()
                    # ["store("] # 'rhs # [", "]
                    # (send 'temp 'getName()) # [")"] # [" ; "]
         }
         else
           set 'lowlevel = [" $CANNOT RESOLVE LHS:$ "] # 'name
         ;
    new 'ClosedCode('lowlevel, ["dummy"])
  }

  method 'addToEnv('env : 'Env) 'Env :
    send 'env 'addVar('name)

  method 'getPtr() ['temp : 'string, 'tempSym : 'string] 'ClosedCode :
  {
    set 'tempSym = GENSYM ;
    set 'temp = 'tempSym # [" = getelementptr(% 'self, "] # 'name # [")"] # [" ; "] ;
    new 'ClosedCode('temp, 'tempSym)
  }

final class 'getNILCode extends 'Code
  method 'eval('env : 'Env) ['sym : 'string, 'lowlevel : 'string] 'ClosedCode :
  {
    set 'sym = (GENSYM) ;
    set 'lowlevel = 'sym # [" = NULL ; "] ;
    new 'ClosedCode('lowlevel, 'sym)
  }

final class 'genSymCode extends 'Code
  method 'eval('env : 'Env) ['sym : 'string, 'lowlevel : 'string] 'ClosedCode :
  {
    set 'sym = (GENSYM) ;
    set 'lowlevel = 'sym # [" = NEWSYM ; "] ;
    new 'ClosedCode('lowlevel, 'sym)
  }

final class 'genLabelCode extends 'Code
  method 'eval('env : 'Env) ['sym : 'string, 'lowlevel : 'string] 'ClosedCode :
  {
    set 'sym = (GENSYM) ;
    set 'lowlevel = 'sym # [" = NEWLABEL ; "] ;
    new 'ClosedCode('lowlevel, 'sym)
  }
```

```
final class 'getSelfCode extends 'Code
  method 'eval('env : 'Env) 'ClosedCode :
    new 'ClosedCode([""], ["% 'self"])

final class 'integerConstantCode extends 'Code
  final field 'value : 'int

  method 'initialize('v : 'int) 'int :
    set 'value = 'v

  method 'eval('env : 'Env) ['sym : 'string, 'lowlevel : 'string] 'ClosedCode :
  {
    set 'sym = (GENSYM) ;
    set 'lowlevel = 'sym # [" = add(0,"] # 'value # [")"] # [" ; "] ;
    new 'ClosedCode('lowlevel, 'sym)
  }

final class 'stringConstantCode extends 'Code
  final field 'value : 'string

  method 'initialize('v : 'string) 'int :
    set 'value = 'v

  method 'eval('env : 'Env) ['sym : 'string, 'lowlevel : 'string] 'ClosedCode :
  {
    set 'sym = (GENSYM) ;
    set 'lowlevel = 'sym # [" = S(\""] # 'value # ["\")"] # [" ; "] ;
    new 'ClosedCode('lowlevel, 'sym)
  }

final class 'booleanConstantCode extends 'Code
  final field 'value : 'string

  method 'initialize('v : 'string) 'int :
    set 'value = 'v

  method 'eval('env : 'Env) ['sym : 'string, 'lowlevel : 'string] 'ClosedCode :
  {
    set 'sym = (GENSYM) ;
    set 'lowlevel = 'sym # [" = and(true,"] # 'value # [")"] # [" ; "] ;
    new 'ClosedCode('lowlevel, 'sym)
  }

final class 'binOpCode extends 'Code
  final field 'op : 'string
  final field 'left : 'Code
  final field 'right : 'Code

  method 'initialize('o : 'string, 'l : 'Code, 'r : 'Code) 'int :
  {
    set 'op = 'o ;
```

```
    set 'left = 'l ;
    set 'right = 'r
  }

  method 'eval('env : 'Env)
  ['sym : 'string, 'lowlevel : 'string,
   'lval : 'ClosedCode, 'rval : 'ClosedCode] 'ClosedCode :
  {
    set 'sym = (GENSYM) ;
    set 'lval = send 'left 'eval('env) ;
    set 'rval = send 'right 'eval('env) ;
    set 'lowlevel = (send 'lval 'getCode())
                  # (send 'rval 'getCode())
                  # 'sym # [" = "] # 'op # ["("]
                  # (send 'lval 'getName()) # [", "]
                  # (send 'rval 'getName()) # [")"] # [" ; "] ;
    new 'ClosedCode('lowlevel, 'sym)
  }

final class 'booleanNotCode extends 'Code
  final field 'value : 'Code

  method 'initialize('v : 'Code) 'int : set 'value = 'v

  method 'eval('env : 'Env)
  ['sym : 'string, 'lowlevel : 'string, 'val : 'ClosedCode] 'ClosedCode :
  {
    set 'sym = (GENSYM) ;
    set 'val = send 'value 'eval('env) ;
    set 'lowlevel = (send 'val 'getCode())
                  # 'sym # [" = not("] # (send 'val 'getName()) # [")"] # [" ; "] ;
    new 'ClosedCode('lowlevel, 'sym)
  }

final class 'ifThenElseCode extends 'Code
  final field 'test : 'Code
  final field 'thenb : 'Code
  final field 'elseb : 'Code

  method 'initialize('t : 'Code, 'b1 : 'Code, 'b2 : 'Code) 'int :
  {
    set 'test  = 't ;
    set 'thenb = 'b1 ;
    set 'elseb = 'b2
  }

  method 'eval('env : 'Env)
  ['sym : 'string, 'lowlevel : 'string, 'cval : 'ClosedCode,
   'tval : 'ClosedCode, 'fval : 'ClosedCode, 'tlabel : 'string,
   'flabel : 'string, 'nlabel : 'string] 'ClosedCode :
  {
    set 'sym = (GENSYM) ;
```

```
    set 'cval = send 'test  'eval('env) ;
    set 'tval = send 'thenb 'eval('env) ;
    set 'fval = send 'elseb 'eval('env) ;
    set 'tlabel = (GENLABEL) ;
    set 'flabel = (GENLABEL) ;
    set 'nlabel = (GENLABEL) ;

    set 'lowlevel = (send 'cval 'getCode())
                  # ["br "] # (send 'cval 'getName()) # [" ? "]
                  # 'tlabel # [" : "] # 'flabel # [" ; "] ;
    set 'lowlevel = 'lowlevel # 'tlabel # [" : "]
                  # (send 'tval 'getCode())
                  # 'sym # [" = copy("] # (send 'tval 'getName()) # [")"] # [" ; "]
                  # [" br "] # 'nlabel # [" ; "] ;
    set 'lowlevel = 'lowlevel # 'flabel # [" : "]
                  # (send 'fval 'getCode())
                  # 'sym # [" = copy("] # (send 'fval 'getName()) # [")"] # [" ; "]
                  # [" br "] # 'nlabel # [" ; "] ;
    set 'lowlevel = 'lowlevel # 'nlabel # [" : "] ;

    new 'ClosedCode('lowlevel, 'sym)
  }

final class 'fieldAccessCode extends 'Code
  final field 'target : 'Code
  final field 'field : 'string

  method 'initialize('t : 'Code, 'f : 'string) 'int :
  {
    set 'target = 't ;
    set 'field = 'f
  }

  method 'eval('env : 'Env)
  ['sym : 'string, 'lowlevel : 'string, 'temp : 'ClosedCode] 'ClosedCode :
  {
    set 'sym = GENSYM ;
    set 'temp = send self 'getPtr('env) ;
    set 'lowlevel = (send 'temp 'getCode()) # 'sym # [" = load("]
                  # (send 'temp 'getName()) # [")"] # [" ; "] ;
    new 'ClosedCode('lowlevel, 'sym)
  }

  method 'evalLHS('env : 'Env, 'rhs : 'string)
  ['lowlevel : 'string, 'temp : 'ClosedCode] 'ClosedCode :
  {
    set 'temp = send self 'getPtr('env) ;
    set 'lowlevel = (send 'temp 'getCode())
                  # ["store("] # 'rhs # [", "]
                  # (send 'temp 'getName()) # [")"] # [" ; "] ;
    new 'ClosedCode('lowlevel, ["dummy"])
  }
```

```
  method 'getPtr('env : 'Env)
  ['temp : 'string, 'tempSym : 'string, 'tarVal : 'ClosedCode] 'ClosedCode :
  {
    set 'tempSym = GENSYM ;
    set 'tarVal = send 'target 'eval('env) ;
    set 'temp = (send 'tarVal 'getCode()) # 'tempSym # [" = getelementptr("]
              # (send 'tarVal 'getName()) # [", "] # 'field # [")"] #[" ; "] ;
    new 'ClosedCode('temp, 'tempSym)
  }

final class 'assignmentCode extends 'Code
  final field 'lhs : 'Code
  final field 'rhs : 'Code

  method 'initialize('l : 'Code, 'r : 'Code) 'int :
  {
    set 'lhs = 'l ;
    set 'rhs = 'r
  }

  method 'eval('env : 'Env)
  ['sym : 'string, 'lowlevel : 'string,
   'rval : 'ClosedCode, 'lval : 'ClosedCode] 'ClosedCode :
  {
    set 'sym = (GENSYM) ;
    set 'rval = send 'rhs 'eval('env) ;
    set 'lval = send 'lhs 'evalLHS('env, (send 'rval 'getName())) ;
    set 'lowlevel = (send 'rval 'getCode()) # (send 'lval 'getCode()) ;
    set 'lowlevel = 'lowlevel # 'sym # [" = add(1,0)"] # [" ; "] ;
    new 'ClosedCode('lowlevel, 'sym)
  }

final class 'statementsCode extends 'Code
  final field 'stmts : 'List

  method 'initialize('l : 'List) 'int :
    set 'stmts = 'l

  method 'eval('env : 'Env)
  ['sym : 'string, 'lowlevel : 'string, 'stmt : 'List,
   'lastName : 'string, 'val : 'ClosedCode] 'ClosedCode :
  {
    set 'sym = (GENSYM) ;
    set 'stmt = 'stmts ;
    set 'lowlevel = [""] ;
    while(send 'stmt 'hasNext()) {
      set 'val = send (cast send 'stmt 'value() to 'Code) 'eval('env) ;
      set 'lowlevel = 'lowlevel # (send 'val 'getCode()) ;
      set 'lastName = send 'val 'getName() ;
      set 'stmt = send 'stmt 'next()
    } ;
```

```
    set 'lowlevel = 'lowlevel # 'sym # [" = copy("] # 'lastName # [")"] # [" ; "] ;
    new 'ClosedCode('lowlevel, 'sym)
  }

final class 'doWhileLoopCode extends 'Code
  final field 'test : 'Code
  final field 'body : 'Code

  method 'initialize('t : 'Code, 'b : 'Code) 'int :
  {
    set 'test = 't ;
    set 'body = 'b
  }

  method 'eval('env : 'Env)
  ['sym : 'string, 'lowlevel : 'string, 'tval : 'ClosedCode, 'bval : 'ClosedCode,
   'yeslabel : 'string, 'nolabel : 'string] 'ClosedCode :
  {
    set 'sym = (GENSYM) ;
    set 'tval = send 'test 'eval('env) ;
    set 'bval = send 'body 'eval('env) ;
    set 'yeslabel = (GENLABEL) ;
    set 'nolabel = (GENLABEL) ;

    set 'lowlevel = ["br "] # 'yeslabel # [" ; "]
                  # 'yeslabel # [" : "]
                  # (send 'bval 'getCode())
                  # (send 'tval 'getCode())
                  # ["br "] # (send 'tval 'getName()) # [" ? "]
                  # 'yeslabel # [" : "] # 'nolabel # [" ; "] ;
    set 'lowlevel = 'lowlevel # 'nolabel # [" : "]
                  # 'sym # [" = add(0, 1) ; "] ;
    new 'ClosedCode('lowlevel, 'sym)
  }

final class 'tempNameCode extends 'Code --- this is an auxiliary class
  final field 'name : 'string

  method 'initialize('n : 'string) 'int : set 'name = 'n

  method 'eval('env : 'Env) 'ClosedCode :
    new 'ClosedCode([""], 'name)

final class 'newInstanceCode extends 'Code
  final field 'klasName : 'string
  final field 'argList : 'List

  method 'initialize('n : 'string, 'a : 'List) 'int :
  {
    set 'klasName = 'n ;
    set 'argList = 'a
  }
```

```
  method 'eval('env : 'Env)
  ['sym : 'string, 'lowlevel : 'string, 'initVal : 'ClosedCode]  'ClosedCode :
  {
    set 'sym = (GENSYM) ;
    set 'lowlevel = 'sym # [" = malloc( "]
                # (send 'env 'objectLayout('klasName)) # [")"] # [" ; "] ;
    set 'initVal = send (new 'invokeCode(new 'tempNameCode('sym),
                                         ['initialize],
                                         'argList)) 'eval('env) ;
    set 'lowlevel = 'lowlevel # (send 'initVal 'getCode()) ;
    new 'ClosedCode('lowlevel, 'sym)
  }

final class 'invokeCode extends 'Code
  final field 'methodName : 'string
  final field 'argList : 'List
  final field 'target : 'Code

  method 'initialize('t : 'Code, 'n : 'string, 'a : 'List) 'int :
  {
    set 'target = 't ;
    set 'methodName = 'n ;
    set 'argList = 'a
  }

  method 'eval('env : 'Env)
  ['sym : 'string, 'lowlevel : 'string, 'arg : 'List, 'params : 'string,
   'argVal : 'ClosedCode, 'vtabptr : 'string, 'vtab : 'string,
   'funName : 'string, 'targetVal : 'ClosedCode]  'ClosedCode :
  {
    set 'sym = (GENSYM) ;
    set 'params = [""] ;
    set 'arg = 'argList ;
    set 'vtabptr = GENSYM ;
    set 'vtab = GENSYM ;
    set 'funName = GENSYM ;

    set 'targetVal = send 'target 'eval('env) ;
    set 'lowlevel = (send 'targetVal 'getCode()) ;

    while (send 'arg 'hasNext()) {
      set 'argVal = (send (cast send 'arg 'value()to 'Code)'eval('env)) ;
      set 'lowlevel = 'lowlevel # (send 'argVal 'getCode()) ;
      set 'params = 'params # [", "] # (send 'argVal 'getName()) ;
      set 'arg = send 'arg 'next()
    } ;
    set 'params = [", "] # (send 'targetVal 'getName()) # 'params ;

    ---dynamic dispatching
    set 'lowlevel = 'lowlevel
                # 'vtabptr # [" = getelementptr("] # (send 'targetVal 'getName())
```

```
                # [", 'VTABLE)"] # [" ; "]
                # 'vtab # [" = load("] # 'vtabptr # [")"] # [" ; "]
                # 'funName # [" = getfunctionname("] # 'vtab # [", "]
                # 'methodName # [")"] # [" ; "]
                # 'sym # [" = call ("] # 'funName # 'params # [")"] # [" ; "] ;
    new 'ClosedCode('lowlevel, 'sym)
  }

final class 'superCallCode extends 'Code
  final field 'methodName : 'string
  final field 'argList : 'List

  method 'initialize('n : 'string, 'a : 'List) 'int :
  {
    set 'methodName = 'n ;
    set 'argList = 'a
  }

  method 'eval('env : 'Env)
  ['sym : 'string, 'lowlevel : 'string, 'arg : 'List, 'params : 'string,
   'argVal : 'ClosedCode, 'funName : 'string]  'ClosedCode :
  {
    set 'sym = (GENSYM) ;
    set 'lowlevel = [""] ;
    set 'params = [""] ;
    set 'arg = 'argList ;
    set 'funName = GENSYM ;

    while (send 'arg 'hasNext()) {
      set 'argVal = (send (cast send 'arg 'value() to 'Code)'eval('env)) ;
      set 'lowlevel = 'lowlevel # (send 'argVal 'getCode()) ;
      set 'params = 'params # [", "] # (send 'argVal 'getName()) ;
      set 'arg = send 'arg 'next()
    } ;
    set 'params = [", % 'self"] # 'params ;

    set 'lowlevel = 'lowlevel
                # 'funName # [" = getfunctionname("] # (send 'env 'superClassName())
                # [", "] # 'methodName # [")"] # [" ; "]
                # 'sym # [" = call ("] # 'funName # 'params # [")"] # [" ; "] ;
    new 'ClosedCode('lowlevel, 'sym)
  }

final class 'castCode extends 'Code
  final field 'castTo : 'string
  final field 'exp : 'Code

  method 'initialize('c : 'string, 'e : 'Code) 'int :
  {
    set 'castTo = 'c ;
    set 'exp = 'e
  }
```

```
  method 'eval('env : 'Env)
  ['sym : 'string, 'lowlevel : 'string, 'val : 'ClosedCode] 'ClosedCode :
  {
    set 'sym = (GENSYM) ;
    set 'val = send 'exp 'eval('env) ;
    set 'lowlevel = (send 'val 'getCode()) # 'sym # [" = copy("]
                  # (send 'val 'getName()) # [")"] # [" ; "] ;
    new 'ClosedCode('lowlevel, 'sym)
  }

final class 'defineFieldCode extends 'Code
  final field 'final : 'bool
  final field 'name : 'string
  final field 'type : 'string

  method 'initialize('f : 'bool, 'n : 'string, 't : 'string) 'int :
  {
    set 'final = 'f ;
    set 'name = 'n ;
    set 'type = 't
  }

  method 'eval('env : 'Env) 'ClosedCode :
    new 'ClosedCode([""],[" $NONAME-FOR-FIELDS$ "])

  method 'addToEnv('env : 'Env) 'Env :
    send 'env 'addField('name)

  method 'getName() 'string :
    'name

final class 'defineVarCode extends 'Code
  final field 'name : 'string
  final field 'type : 'string

  method 'initialize('n : 'string, 't : 'string) 'int :
  {
    set 'name = 'n ;
    set 'type = 't
  }

  method 'eval('env : 'Env) 'ClosedCode :
    new 'ClosedCode([""], 'name)

  method 'addToEnv('env : 'Env) 'Env :
    send 'env 'addVar('name)

  method 'getName() 'string :
    'name

final class 'defineMethodCode extends 'Code
```

```
  final field 'final : 'bool
  final field 'name : 'string
  final field 'paramList : 'List
  final field 'varList : 'List
  final field 'retType : 'string
  final field 'body : 'Code

  method 'initialize('f : 'bool, 'n : 'string, 'p : 'List,
                     'v : 'List, 'r : 'string, 'b : 'Code) 'int :
  {
    set 'final = 'f ;
    set 'name = 'n ;
    set 'paramList = 'p ;
    set 'varList = 'v ;
    set 'retType = 'r ;
    set 'body = 'b
  }

  method 'eval('env : 'Env) ['lowlevel : 'string, 'klasName : 'string, 'param : 'List,
                             'bval : 'ClosedCode, 'paramName : 'string] 'ClosedCode :
  {
    *** define environment
    set 'env = send (send (send 'env 'resetForNewMethod())
                          'addList('paramList)) 'addList('varList) ;

    set 'klasName = send 'env 'currentClassName() ;
    set 'lowlevel = 'klasName # [" $ "] # 'name # [" (% 'self"] ;
    set 'param = 'paramList ;
    while (send 'param 'hasNext()) {
      set 'paramName = (send (cast send 'param 'value() to 'Code) 'getName()) ;
      set 'lowlevel = 'lowlevel # [", % "] # 'paramName ;
      set 'param = send 'param 'next()
    } ;
    set 'lowlevel = 'lowlevel # [" ) { 'entry : "] ;
    set 'bval = send 'body 'eval('env) ;
    set 'lowlevel = 'lowlevel # (send 'bval 'getCode()) # [" br 'end ; "]
                  # ["'end : return "] # (send 'bval 'getName())
                  # [" ; } "] ;
    new 'ClosedCode('lowlevel, [" $NONAME-FOR-METHOD$ "])
  }

  method 'getName() 'string :
    'name

final class 'VtablePair extends 'object
  final field 'vtab : 'string
  final field 'overriddens : 'List

  method 'initialize('v : 'string, 'o : 'List) 'int :
  {  set 'vtab = 'v ;
     set 'overriddens = 'o
  }
```

```
final class 'defineClassCode extends 'Code
  final field 'final : 'bool
  final field 'name : 'string
  final field 'superName : 'string
  final field 'fieldList : 'List
  final field 'methodList : 'List

  method 'initialize('fin : 'bool, 'n : 'string,
                     's : 'string, 'f : 'List, 'm : 'List) 'int :
  {
    set 'final = 'fin ;
    set 'name = 'n ;
    set 'superName = 's ;
    set 'fieldList = 'f ;
    set 'methodList = 'm
  }

  method 'eval('env : 'Env)
  ['lowlevel : 'string, 'meth : 'List, 'methVal : 'ClosedCode,
   'sname : 'string, 'spr : 'defineClassCode] 'ClosedCode :
  {
    *** define environment
    set 'env = send (send 'env 'resetForNewClass('name, 'superName))
                          'addList('fieldList) ;
    set 'sname = 'superName ;
    while not('sname equals ['object])
    {
      set 'spr = send 'env 'getClassInfo('sname) ;
      set 'env = send 'env 'addList(send 'spr 'getFieldList()) ;
      set 'sname = send 'spr 'getSuperName()
    } ;
    *** define environment

    set 'lowlevel = [""] ;
    set 'meth = 'methodList ;
    while (send 'meth 'hasNext()) {
      set 'methVal = cast send 'meth 'value() to 'Code ;
      set 'lowlevel = 'lowlevel # (send (send 'methVal 'eval('env)) 'getCode()) ;
      set 'meth = send 'meth 'next()
    } ;
    new 'ClosedCode('lowlevel, [" $NONAME-FOR-CLASSES$ "])
  }

  method 'classVTable('env : 'Env) 'string :
    send 'env 'classVTable('name)

  method 'vtable('overriddens : 'List)
  ['vtab : 'string, 'methNode : 'List, 'meth : 'defineMethodCode] 'VTablePair :
  {
    set 'vtab = [""] ;
    set 'methNode = 'methodList ;
```

```
    while (send 'methNode 'hasNext()) {
      set 'meth = cast send 'methNode 'value() to 'Code ;
      if (send 'overriddens 'has(send 'meth 'getName())) then
        0
      else {
        set 'vtab = 'vtab # [" {"] # (send 'meth 'getName())
                  # [","] # 'name # ["} "] ;
        set 'overriddens = send 'overriddens 'add(send 'meth 'getName())
      } ;
      set 'methNode = send 'methNode 'next()
    } ;
    new 'VtablePair('vtab, 'overriddens)
  }

  method 'getSuperName() 'string :
    'superName

  method 'getFieldList() 'List :
    'fieldList

  method 'getName() 'string :
    'name

  method 'getLayout()
  ['fNode : 'List, 'f : 'defineFieldCode, 'layout : 'string] 'string :
  {
    set 'layout = [""] ;
    set 'fNode = 'fieldList ;
    while (send 'fNode 'hasNext()) {
      set 'f = cast send 'fNode 'value() to 'Code ;
      set 'layout = 'layout # ["{"] # (send 'f 'getName()) # [", nil}"] ;
      set 'fNode = send 'fNode 'next()
    } ;
    'layout
  }

  method 'addToEnv('env : 'Env) 'Env :
    send 'env 'addClass(self)

final class 'defineProgramCode extends 'Code
  final field 'classes : 'List
  final field 'varDecls : 'List
  final field 'mainBody : 'Code

  method 'initialize('c : 'List, 'v : 'List, 'm : 'Code) 'int :
  {
    set 'classes = 'c ;
    set 'varDecls = 'v ;
    set 'mainBody = 'm
  }

  method 'addToEnv('env : 'Env) 'Env :
```

```
    send 'env 'addList('classes)

  method 'eval('env : 'Env)
  ['lowlevel : 'string, 'klas : 'string,
   'klasVal : 'defineClassCode, 'bval : 'ClosedCode] 'ClosedCode :
  {
    set 'env = send self 'addToEnv('env) ;
    set 'lowlevel = ["DEFS {'object : {'initialize, 'object}}"]
                  # (send self 'vTables('env))
                  # [" IMPL 'object $ 'initialize (% 'self ) { 'entry : "]
                  # ["br 'end ; 'end : return 0 ; } "] ;
    set 'klas = 'classes ;
    while (send 'klas 'hasNext()) {
      set 'klasVal = cast send 'klas 'value() to 'Code ;
      set 'lowlevel = 'lowlevel # (send (send 'klasVal 'eval('env)) 'getCode());
      set 'klas = send 'klas 'next()
    } ;

    set 'env = send 'env 'addList('varDecls) ;
    set 'lowlevel = 'lowlevel # [" 'main() { 'entry : "] ;
    set 'bval = send 'mainBody 'eval('env) ;
    set 'lowlevel = 'lowlevel # (send 'bval 'getCode()) # [" br 'end ; "]
                  # ["'end : return "] # (send 'bval 'getName())
                  # [" ; } "] ;
    new 'ClosedCode('lowlevel, [" $NONAME-FOR-PROGRAM$ "])
  }

  method 'vTables('env : 'Env)
  ['lowlevel : 'string, 'klas : 'List, 'klasVal : 'defineClassCode] 'string :
  {
    set 'lowlevel = [""] ;
    set 'klas = 'classes ;
    while (send 'klas 'hasNext()) {
      set 'klasVal = cast send 'klas 'value() to 'Code ;
      set 'lowlevel = 'lowlevel # [" "] # (send 'klasVal 'classVTable('env)) ;
      set 'klas = send 'klas 'next()
    } ;
    'lowlevel
  }

--------------------
--- Code Base  -----
--------------------
class 'Code extends 'object
  method 'eval('env : 'Env) 'ClosedCode :
    new 'ClosedCode([""], [""])

  method 'evalLHS('env : 'Env, 'rhs : 'string) 'ClosedCode :
    new 'ClosedCode([""], [""])

  method 'addToEnv('env : 'Env) 'Env :
    'env
```

```
  final method 'generate() 'string :
    send (send self 'eval(send self 'freshEnv())) 'getCode()

  final method 'generateWithClasses('lib : 'List) ['env : 'Env] 'string :
  {
    set 'env = send self 'freshEnv() ;
    set 'env = send 'env 'addList('lib) ;
    send (send self 'eval('env)) 'getCode()
  }

  method 'getName() 'string :
    [""]

  final method 'freshEnv() 'Env :
    new 'Env(new 'LinkedList(NIL, NIL), new 'LinkedList(NIL, NIL),
             [""], [""], new 'LinkedList(NIL, NIL))

  method 'classVTable('env : 'Env) 'string :
    [" $NO-CLASS-VTABLE-FOR-THIS-CODE$ "]

  method 'dummy('a : 'string, 'b : 'object) 'int :
    1

final class 'ClosedCode extends 'object
  final field 'name : 'string
  final field 'lowlevelcode : 'string

  method 'initialize('l : 'string, 'n : 'string) 'int :
  {
    set 'lowlevelcode = 'l ;
    set 'name = 'n
  }

  method 'getName() 'string :
    'name

  method 'getCode() 'string :
    'lowlevelcode

final class 'Env extends 'object
  final field 'varList : 'List
  final field 'fieldList : 'List
  final field 'currClass : 'string
  final field 'superClass : 'string
  final field 'classInfos : 'List

  method 'initialize('v : 'List, 'f : 'List, 'k : 'string,
                     's : 'string, 'i : 'List) 'int :
  {
    set 'varList = 'v ;
    set 'fieldList = 'f ;
```

```
  set 'currClass = 'k ;
  set 'superClass = 's ;
  set 'classInfos = 'i
}

method 'isLocal('name : 'string) 'bool :
  send 'varList 'has('name)

method 'isField('name : 'string) 'bool :
  send 'fieldList 'has('name)

method 'resetForNewMethod() 'Env :
  new 'Env(new 'LinkedList(NIL, NIL), 'fieldList,
           'currClass, 'superClass, 'classInfos)

method 'resetForNewClass('cc : 'string, 'sc : 'string) 'Env :
  new 'Env(new 'LinkedList(NIL, NIL), new 'LinkedList(NIL, NIL),
           'cc, 'sc, 'classInfos)

method 'superClassName() 'string :
  'superClass

method 'currentClassName() 'string :
  'currClass

method 'addField('n : 'string) 'Env :
  new 'Env('varList, send 'fieldList 'add('n),
           'currClass, 'superClass, 'classInfos)

method 'addVar('n : 'string) 'Env :
  new 'Env(send 'varList 'add('n), 'fieldList, 'currClass, 'superClass, 'classInfos)

method 'addClass('c : 'defineClassCode) 'Env :
  new 'Env('varList, 'fieldList, 'currClass, 'superClass, send 'classInfos 'add('c))

method 'addList('list : 'List)
['lNode : 'List, 'val : 'Code, 'tempEnv : 'Env] 'Env :
{
  set 'lNode = 'list ;
  set 'tempEnv = self ;
  while(send 'lNode 'hasNext()) {
    set 'val = cast send 'lNode 'value() to 'Code ;
    set 'tempEnv = send 'val 'addToEnv('tempEnv) ;
    set 'lNode = send 'lNode 'next()
  } ;
  'tempEnv
}

method 'layout('name : 'string)
['info : 'defineClassCode, 'superKlas : 'string, 'layout : 'string] 'string :
{
  if 'name equals ['object] then
```

```
    [""]
  else {
    set 'info = send self 'getClassInfo('name) ;
    if 'info equals NIL then
      [" $NO-CLASS-INFO-FOR "] # 'name # ["$ "]
    else {
      set 'superKlas = send 'info 'getSuperName() ;
      set 'layout = send 'info 'getLayout() ;
      'layout # (send self 'layout('superKlas))
    }
  }
}

method 'vtable('name : 'string, 'methList : 'List)
['info : 'defineClassCode, 'layout : 'string, 'vtabPair : 'VtablePair] 'string :
{
  if 'name equals ['object] then {
    if (send 'methList 'has(['initialize])) then
      [""]
    else
      [" {'initialize, 'object} "]
  }
  else {
    set 'info = send self 'getClassInfo('name) ;
    if 'info equals NIL then
      [" $NO-CLASS-INFO-FOR "] # 'name # ["$ "]
    else {
      set 'vtabPair = send 'info 'vtable('methList) ;
      set 'layout = 'vtabPair -> 'vtab ;
      set 'methList = 'vtabPair -> 'overriddens ;
      'layout # (send self 'vtable( send 'info 'getSuperName(), 'methList))
    }
  }
}

method 'objectLayout('name : 'string) 'string :
  ["{{'VTABLE, name("] # 'name # [")}"]
  # (send self 'layout('name)) # ["}"]

method 'classVTable('name : 'string) 'string :
  ["{"] # 'name # [" : "]
  # (send self 'vtable('name, new 'LinkedList(NIL, NIL))) # ["}"]

method 'getClassInfo('name : 'string)
['infoNode : 'List, 'info : 'defineClassCode,
 'res : 'defineClassCode] 'defineClassCode :
{
  set 'infoNode = 'classInfos ;
  set 'res = NIL ;
  while (send 'infoNode 'hasNext()) {
    set 'info = cast send 'infoNode 'value() to 'Code ;
    if ((send 'info 'getName()) equals 'name) then
```

```
        set 'res = 'info
      else
        0 ;
      set 'infoNode = send 'infoNode 'next()
    } ;
    'res
  }

-----------------
--- UTIL    -----
-----------------
final class 'LinkedList extends 'List
  method 'eval('env : 'Env)
  ['lowlevel : 'string, 'node : 'LinkedList, 'nodeVal : 'Code] 'ClosedCode :
  {
    *** define env
    set 'env = send 'env 'addList(self) ;
    set 'lowlevel = [""] ;
    set 'node = self ;
    while (send 'node 'hasNext()) {
      set 'nodeVal = cast send 'node 'value() to 'Code ;
      set 'lowlevel = 'lowlevel # (send (send 'nodeVal 'eval('env)) 'getCode());
      set 'node = send 'node 'next()
    } ;
    new 'ClosedCode('lowlevel, [" $NONAME-FOR-LIST$ "])
  }
  method 'add('v : 'object) 'List : new 'LinkedList('v, self)

class 'List extends 'Code
  final field 'value : 'object
  final field 'next : 'List

  method 'initialize('v : 'object, 'n : 'List) 'int :
  { set 'value = 'v ;
    set 'next = 'n
  }
  method 'add('v : 'object) 'List : NIL

  final method 'hasNext() 'bool :
    if 'next equals NIL then False else True

  final method 'next() 'List :
    'next

  final method 'value() 'object :
    'value

  final method 'has('v : 'object) 'bool :
  {
    if ('v equals 'value)
      then True
    else if (send self 'hasNext())
```

```
          then send 'next 'has('v)
        else False
  }

  final method 'addToEnv('env : 'Env) 'Env :
    send 'env 'addList(self)

) .

endfm
```