Mumbo: A Rule-Based Implementation of a Run-time Program Generation Language

Barış Aktemur [1,2]

Computer Science Department
University of Illinois
Urbana, IL, USA

Sam Kamin [3]

Computer Science Department
University of Illinois
Urbana, IL, USA

Abstract

We describe our efforts to use rule-based programming to produce a model of Jumbo, a run-time program generation (RTPG) system for Java. Jumbo incorporates RTPG following the simple principle that the regular compiler — or, rather, its back-end — can be used both for ordinary, static compilation and for run-time compilation. This tends to produce a run-time compiler that is inefficient but potentially subject to improvement by partial evaluation. However, the complexity of the language and compiler have made it difficult for us to achieve actual optimization. The model, written in Maude, preserves all the essential ingredients of Jumbo, but operates on a simplified language, called Mumbo. The simplification in the language together with Maude's support for code rewriting has allowed us to make rapid progress. We discuss the model in detail, the kinds of optimizations we have obtained, and the impact on the Jumbo project.

Key words: Run-time program generation, Rule-based programming

[1] This work was partially supported by National Science Foundation under grant CCR-0306221.

[2] Email: aktemur@cs.uiuc.edu

[3] Email: kamin@cs.uiuc.edu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

We describe our efforts to use rule-based programming to produce a model of a run-time program generation (RTPG) system for Java. Our group has developed such a system, called Jumbo [12,6,10,11], which is written in Java. However, our attempts to optimize RTPG in Jumbo have been stymied by the overall complexity of the compiler. The model we describe here is written in Maude [7] following the same approach as Jumbo but addressing a subset of Java; we call the system Mumbo, for "mini-Jumbo." The combination of a simplified language and the rule-based approach to programming supported by Maude has allowed us to have substantial success in optimization. We have learned several lessons from this model that we are now applying to Jumbo. This paper describes Mumbo and discusses the lessons we have learned from it. The project is a good example of using the power of rule-based programming to "bootstrap" a conventionally-programmed system.

Run-time program generation is an approach to program optimization in which data discovered only at run time are used to produce a more efficient version of a program. There are a number of systems [8,16,19,3,4] that facilitate the production of run-time program generators by allowing the dynamically generated code to be specified in the source language. All these systems are at least superficially similar. They include a code-quotation syntax — we'll use quote brackets `$<` and `>$` — to specify code that will be generated at run time. This code can have unknown parts, or *holes*, that will be filled in at run time using the result of some calculation that, for whatever reason, cannot be performed statically. Holes are indicated by an anti-quoted expression within a piece of quoted code; we will use the notation `‘(`e`)`, where e is an expression representing the computation of the missing piece of program. Consider the following code:

```
$< obj.foo(‘(param)); >$
```

Here we have a hole which will be filled in with a `Code` object named `param`, as in:

```
Code param = $< x >$;
Code c = $< obj.foo(‘(param)); >$ ;
```

The ability of RTPG to achieve speed-ups over fixed, statically generated code depends upon the speed of the generated code and also the speed of the code generation process. Indeed, if a generated piece of code is executed just once, it is almost certainly not going to be worth the time it takes to produce it. At some number of repetitions, the cost of code generation is finally paid off and efficiency dividends begin to be paid; that number is called the *crossover point*. Minimizing the cost of RTPG, and thereby minimizing the crossover point, is a goal of all RTPG systems.

We have produced an RTPG system for Java, called Jumbo. Jumbo is distinguished from the systems mentioned above by the fact that it compiles

all of Java 1.4, and, more fundamentally, by its method of construction and the consequent generality of the code it can generate at run time.

To optimize RTPG, other systems do two things: They limit the degree of variability that can occur in generated code, and they write a run-time compiler specialized to emit machine instructions quickly in this constrained environment. Most of them, for example, require that any variables appearing in quoted code must be declared in that same piece of quoted code; variable capture, in which a variable is used in one quoted fragment which is then used to fill a hole in another quoted fragment which contains the variable's declaration, is not allowed. (This requirement is also necessary to permit static type-checking of generated code.) Knowing the type of a variable, even if the precise way in which the variable will be used is not known, allows the run-time compiler to emit code quickly.

We have taken a different tack in Jumbo. We believe, based on numerous examples, that such restrictions make run-time program generators less useful and harder to write than they can be. Our approach is based on the observation that, if a compiler is written in compositional form — meaning the compilation of any code piece is a function only of the compilation of its subcomponents — then we can use the back end of the static compiler — which does not deal with "holes" in code at all — as the RTPG mechanism. (Here, "compilation" means something different from "machine code;" it means some value from which machine code can easily be produced — details later in the paper.) This achieves a very high level of generality, as almost any syntactic fragment — including, for example, declarations — can be abstracted out of a fragment. On the other hand, it leaves little room for optimization. After all, the back end of the compiler is presumably already as efficient as its programmers could make it. But, in fact, this approach permits a different avenue of optimization: partial evaluation of the back end of the compiler. Consider a program fragment like this:

```
$< System.out.println(x+1); >$
```

We cannot generate code for this because we do not know the type of `x`; in Java, it could even be a string, with `+` representing concatenation. But we can *partially* generate code: we know there will be a call to `println`, and that the operator is either addition or concatenation, rather than multiplication or anything else. Roughly speaking, we could generate code like this:

```
if (x of type String)
  emit code to concatenate "1" to x
else if (x a number) {
  t = minimum type above the type of x and int;
  emit code to coerce x to t, if necessary
  emit code to coerce 1 to t, if necessary
  emit appropriate type of add instruction
}
emit call to println
```

Keep in mind that the static compiler we are starting from is not written to handle RTPG. It has no special mechanism for deferring the look-up of types to run-time, as we have done here. Rather, our job is to take the compiler as given and partially evaluate it to obtain this code.

In summary, Jumbo works by writing a compiler that is in compositional style but is conventional in the sense that it knows nothing about RTPG. The idea is to optimize a back end using whatever information the quoted code fragments give us. Fragments that are "more static" in that they, for example, make no use of variable capture, will *naturally* be more efficient. But more dynamic uses will still be accommodated. The programmer pays for the amount of dynamic-ness he needs. But this all depends on the ability to partially evaluate the compiler's back end. That is the topic of this paper.

We have tried for some time to partially evaluate Jumbo in this way. We have experienced partial success (building on a system written by Lars Clausen as part of his PhD thesis [6]). We have written a number of optimizations that can transform the code generators produced initially by the compiler. But we have yet to produce code like that described above, in which the run-time code generator has been reduced to its essentials, and we have yet to produce really significant speed-ups. That is partly because Java is complicated (many "obvious" transformations are not valid), and partly because the compiler is not written in a way that makes it amenable to transformation. For this reason, we undertook to write a model of Jumbo in Maude, using a simplified language. With Mumbo, we have been able to achieve the code we expected. This paper describes Mumbo, lists the optimizations we have implemented, shows how they lead to significant optimization of RTPG, discusses the limitations of the optimization rules, and concludes with the lessons we have learned from Mumbo, which we are applying to Jumbo.

Complete details on Mumbo can be found in [2], and the source code is available online at `http://pinatubo.cs.uiuc.edu/~aktemur/mumbo`.

2 Maude

We implemented Mumbo in Maude [7]. Maude is a powerful tool supporting rewriting logic. In Maude one can define *equations* or *rules*. Rules are used to describe concurrent transitions in a system. In Mumbo, we do not have concurrency. Hence, we only use equations, which may also be conditional.

Maude supports modular development, and it can execute the given equations [4]. It also includes verification and proof tools, but we do not use these tools in Mumbo.

[4] For this purpose, equations have to be executable. For details on executability requirements, see [7].

3 Jumbo

Jumbo [6,12,11,10] is a staged compilation system for `Java`, allowing run-time program generation. It provides a high degree of programmer control, source level specification and binary-level operation. With Jumbo, it is possible to produce code without invoking a compiler at run-time. Since many computers have a Java run-time but no compiler, this is an important practical feature.

As discussed earlier, the Jumbo programmer specifies code to be generated at run-time by placing it within quotation brackets: `$<` and `>$`. From the programmer's point of view, these brackets behave very much like ordinary string quotes, but the values represented are of type `Code`, not `String`, and ordinary string operations cannot be applied. Accordingly, the enclosed piece of program is not arbitrary, but it can be almost any parsable fragment.

A quoted `Java` fragment can have *holes* inside — spots that will be filled with `Code` values not known at code-writing time. The syntax for holes is backquote (`‘`) followed by a syntax category, followed by a `Java` expression of type `Code` in parentheses. Consider

```
public Code infiniteLoopGen(Code body){
  return $< while(true){
              ‘Stmt(body)
            } >$;
}
```

This method can be called as:

```
infiniteLoopGen($< if(i == 3) break; i++; >$);
```

This call would give us `Code` equivalent to:

```
while(true){
  if(i == 3)
    break;
  i++;
}
```

This code can now be used in a context where `i` is defined.

For expressions of primitive type, there is a second type of anti-quotation, one which 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`.

`Code` is the main class in the Jumbo implementation. A `Code` value represents the *partially* compiled version of a program fragment and is represented as a method. Its argument is the information about the usage context of that fragment that is needed to fully compile the fragment down to virtual machine code; its result is the virtual machine code thus calculated. Because it is a method, this program fragment is represented as virtual machine code, rather than as source or as a syntax tree. This property of Jumbo favors both security and efficiency. Since methods are not first-class-citizens, we represent a `Code`

value as a function object, which provides `eval` as the "function application" method.

More information on Jumbo is available in [6,10,11,12]. We will say no more about Jumbo until the end of the paper, when we will discuss the impact of the Mumbo project on the Jumbo compiler. Jumbo can be obtained at http://loome.cs.uiuc.edu/Jumbo/index.php.

4 Mumbo

Mumbo is a typed, object-oriented language supporting run-time program generation. It can be considered as a simplified version of Jumbo. It consists of five main parts: Syntax, Preprocessing, Semantics, Compiler, and Optimization (i.e. Analyzers and Transformers). Each part is defined in Maude, except the compiler, which is implemented in Mumbo.

We execute Mumbo programs after compiling them to a virtual machine code, called `LowLevel`. This makes Mumbo model Jumbo more truly because Jumbo also is compiled to virtual machine code — the JVM bytecode. Furthermore, at run-time, Mumbo produces `LowLevel` code, similar to Jumbo producing JVM bytecode.

`LowLevel` is a 3-address code language we defined in about 400 lines of Maude. Its syntax and semantics are inspired by LLVM (Low Level Virtual Machine) [13]. It provides register operations such as addition, equality testing, (un)conditional branching, etc. In `LowLevel`, it is possible to define and invoke functions. The programmer can define vtables, and structs. This makes `LowLevel` appropriate to be the virtual machine code of an object-oriented language like Mumbo. We do not give details on `LowLevel` as it is not the focus of this paper.

Mumbo programs are compiled to `LowLevel` by the Mumbo compiler. The compiler is implemented in Mumbo. In Maude, the semantics of Mumbo becomes executable and we use this executable semantics to execute the Mumbo compiler. This is possible because from the standpoint of semantics, the compiler is an ordinary Mumbo program. This way, the Mumbo compiler is able to compile itself. We explain the compiler in more detail in Section 4.4.

In the following sections we explain the syntax, preprocessing stage, semantics, compiler and the optimization rules of Mumbo.

4.1 Syntax

Mumbo is an expression oriented language; there is no distinction between statements and expressions. Mumbo uses `$<` and `>$` brackets, as in Jumbo, to define program fragments. Caret (`^(...)`) is the anti-quotation character [5] for expressions. In addition to that, `^I(...)`, `^B(...)` and `^S(...)` can be

[5] We could not use the backquote (‘) character as in Jumbo because it conflicts with a built-in operator in Maude.

used to lift integers, booleans and strings, respectively. `^F(...)` and `^M(...)` are used for anti-quoting a field and a method, respectively (the `F` and `M` characters being needed to allow parsing of the enclosing program fragment). Below are some important parts of the syntax.

- Strings are defined by placing built-in strings of Maude or quoted identifiers (`Qid`s) between square brackets, such as `["abc"]` or `['xyz]`.
- `Qid`s are used as names. `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 defined as commutative in Maude. This does not hold in Mumbo because of possible side effects.
- A field of an object can be accessed with `->`, such as `'obj -> 'f`.
- Methods have the following syntax:

  ```
  op ___[_]_:_ : MMethodFlag MName MParamList MVarDecls
                 MName MExp -> MMethod .
  op ____:_ : MMethodFlag MName MParamList MName MExp -> MMethod .
  ```

  The method flag can be either `method` or `final method`. Then comes the name of the method, followed by the parameter list. A parameter list is in the form: `('a`$_1$ `: 't`$_1$`, ..., 'a`$_n$ `: 't`$_n$`)` where $n \geq 0$, `'a`$_n$ is the name of the nth parameter and `t`$_n$ is its type. The local variables used in the body of the method are 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 is in the form: `['a`$_1$ `: 't`$_1$`, ..., 'a`$_n$ `: 't`$_n$`]`. Lastly comes the return type of the method, followed by a colon, followed by the method body.
- Classes have the syntax:

  ```
  op __extends__ : MClassFlag MName MName MMethods -> MClass .
  op __extends___ : MClassFlag MName MName MFields MMethods -> MClass .
  ```

  The class flag can be either `class` or `final class`. Then comes the name of the class, followed by the keyword `extends`, followed by the name of the superclass. After the class header come the field list and the methods. The user can simply omit fields if the class has none.
- A method of an object is called by sending it a message:

  ```
  op send___ : MExp MName MExpList -> MExp .
  ```

  In this syntax, the first argument is the target, which is expected to evaluate to an object, and the second argument is the name of the message (method) followed by the list of arguments.
- A program is an executable unit consisting of zero or more classes and a `main` method. `main` is similar to a method, but it does not have a parameter

list or a return type.

```
op _main‘[_]_ : MClasses MVarDecls MExp -> MProgram .
```

We believe that the remaining syntactic issues will be clear from the examples throughout this paper.

4.2 Preprocessing

In Mumbo, as in many real systems, code quotation is not in the language's abstract syntax, but is preprocessed away. The preprocessing stage in Mumbo has two mutually recursive functions: `preprocess` and `code`. `preprocess` is an identity function for all syntactic units except code quotation. It just calls itself recursively on subcomponents of those syntactic units. When it comes across a quotation, it calls `code`. Below are some of the equations defined for `preprocess`.

```
eq preprocess(X) = X . --- X is a name
eq preprocess(if BE then E else E') =
     if preprocess(BE) then preprocess(E) else preprocess(E') .
eq preprocess($< E >$) = code(E) .
```

`code` converts a fragment to a `Code` object representing the fragment. When an anti-quotation is seen, `code` reduces to `preprocess`. Below are some of the equations defined.

```
eq code(X) = new 'getNameCode([X]) . --- X is a name
eq code(if Be then E else E') =
     new 'ifThenElseCode(code(Be), code(E), code(E')) .
eq code(^(E)) = preprocess(E) .
```

So, every program fragment given in brackets is converted to an expression — an instantiation of a `'Code`[6] object representing the program fragment enclosed. For instance

- `set 'a = $< 2 >$` becomes `set 'a = new 'integerConstantCode(2)`
- `$< send self 'foo() >$` becomes

```
new 'invocationCode(new 'getSelfCode(), ['foo],
                    new 'LinkedList(NIL, NIL))
```

- `$< ^('c)>$` becomes `'c`

4.3 Semantics

The semantics of the language is defined via Maude equations. Mumbo applies dynamic dispatching when calling a method of an object. When an object is created with the `new` command, its `'initialize` method is called automatically. `'initialize` is a special method in Mumbo, and its explicit invocation is not allowed (like constructors in Java). Subclasses can override methods of their superclass. The interpretation of the `final` flag for classes,

[6] In Mumbo `Qids` are used as names. So `'Code` in Mumbo is the same as `Code` in Jumbo

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 must 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 must have distinct names.

In Mumbo there is no distinction between statements and expressions. Blocks evaluate to what their last expression evaluates to. Variable assignments and loops evaluate to 0. A method returns what its body evaluates to. The `new` command returns a reference to the object it instantiated.

Mumbo does not have static fields/methods. This causes problems when generation of unique numbers is required, which happens in the Mumbo compiler. On the other hand, we would like to keep the language simple. Thus, we chose to add special keywords to the language, instead of static fields. When evaluated, `GENSYM` returns a new virtual-machine name in the form (`% 'sX`)[7], where `X` is a unique number. Similarly, `GENLABEL` returns a new virtual-machine label in the form (`'lX`), where `X` is a unique number. These two special expressions are handled as part of the semantics.

As noted above, the preprocessing stage removes all the quotations. Hence, quotation and anti-quotation are not part of the kernel, and we do not define semantics of those.

We mainly follow the style provided in [18,14], to define the semantics. We define an `eval` operation for each expression type. At the top level we have

```
op eval : MProgram MState -> MValue .
```

which evaluates a Mumbo program starting with an initial `State`. The `State` includes all the necessary information, including an `Environment`, which maps names to locations, and the `Store`, which maps locations to values. As its result, `eval` returns the value of the program.

We do not give the full source code of semantics because of space limitations.

4.4 Compiler

In this section we explain the Mumbo compiler. The compiler is implemented in Mumbo and produces `LowLevel` code. The result of executing the `LowLevel` code after compiling a program with the compiler is the same as executing the program with the equational semantics.

In Section 3 we stated that every object of type `'Code` has an `'eval` method. In Mumbo the `'eval` method takes a parameter of type `'Env`. `'Env` contains all the necessary information for that fragment of code to produce corresponding low-level code. This includes (1) the list of local variables and

[7] In the target low-level code, quoted identifiers are used as labels, and quoted identifiers prefixed with a percent sign are used as names.

parameters of the enclosing method, (2) the list of fields of the enclosing class, (3) the name of that class, (4) the name of that class's superclass, and (5) the list of all classes in the current program. The `'eval` method returns a `'ClosedCode` object. It is a tuple containing 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[8]. 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)
  }
```

The compiler consists of a `'Code` class for each syntactic element, plus `'ClosedCode`, `'Env` and `'LinkedList`, and several auxiliary classes. The compiler is completely implemented in Mumbo. Like Jumbo, it is *compositional*. As we mentioned in Section 4, we take advantage of the executable semantics to execute the compiler — which is just a Mumbo program from the point of view of semantics.

The compiler is a set of classes and the zero-ary function `MCompiler` represents this set (`op MCompiler : -> MClasses .`). The most important property of the compiler is that it is *side-effect-free*. In other words, it is implemented in *functional style*. We achieve this by defining *every field as final*. When there is need to change the state of an object, it returns 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)
```

The `'Code` class defines the `'generate` method. When we have a `'Code` object representing a full program or class, we can call `'generate` to get the corresponding `LowLevel` code. Generation starts with an empty environment.

```
final method 'generate() ['env : 'Env] 'string :
{ set 'env = new 'Env(new 'LinkedList(NIL, NIL), ---var. list
                      new 'LinkedList(NIL, NIL), ---field list
                      [""],                      ---enclosing class name
                      [""],                      ---superclass name
```

[8] Recall that `LowLevel` is a 3-address code language. So, the names are explicit. We would not need `'lowlevelName` if it were stack code, like the JVM bytecode.

```
                    new 'LinkedList(NIL, NIL)) ; ---list of classes
  send (send self 'eval('env)) '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()
```

For static time compilation, one can use the `compile` operation. `compile` is used to obtain the corresponding `LowLevel` code for a given program. It is defined as follows.

```
op compile : MProgram -> MValue .
eq compile(P) = eval(MCompiler
                  main [noVars]
                    send (preprocess($< P >$)) 'generate(), initialState) .
```

In other words, this operation is used to pass Mumbo programs to the Mumbo compiler. An illustration is given in Figure 1.

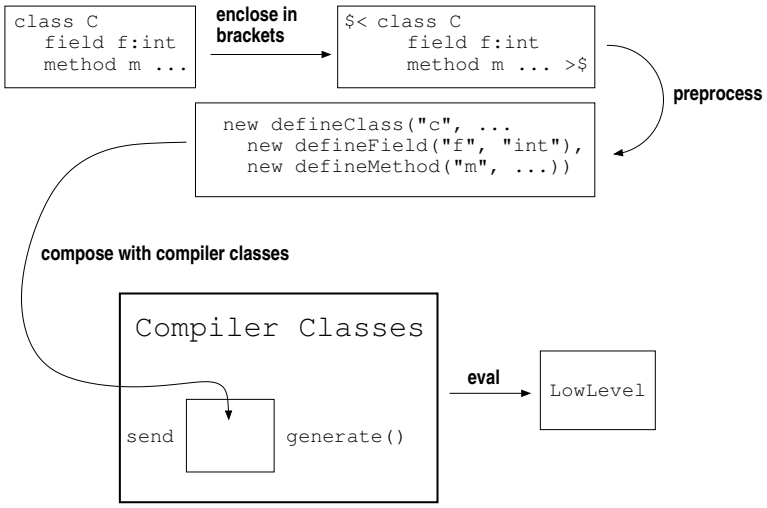

Fig. 1. Compiling a Mumbo class.

4.5 Analyzers and Transformers

For ordinary programs, source-level optimization performed by the compiler is rarely very effective. However, in Mumbo, it is the primary means of optimizing run-time code generators. These are made up of the code of the compiler (the `'Code` class and so on), but are inaccessible to the user. In short, this is a context in which source-level optimization can be effective.

In this section we explain the source-level optimizers of Mumbo. By means of equational logic and matching, Maude makes it easy to traverse the syntax tree of a program. There are two basic kinds of traversal: analyzers and transformers. Analyzers traverse the syntax tree and collect information; transformers modify the tree [9]. To define these traversals easily, we first define a module called `VISIT`, which defines an operation called `visit`:

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

The first argument to this operation is the name of the operation we wish to apply on the syntax tree node (e.g: `constantPropagation`). The second argument is the node to traverse. The last argument is data. This argument is used when we would like to pass information from nodes to nodes. `visit` returns a pair of an expression and data. The returned expression can be used to replace the existing expression; the returned data can be used for various purposes. The `visit` operation is generic enough to be used to transform the tree, to analyze it, or to transform and analyze at the same time. It is similar to the traversal functions of [20] and the Visitor pattern of [9].

The default behavior of traversal is defined in the `IDENTITY` module. For each syntactic component we define `visit` to apply recursively on the subtrees 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`). To illustrate, let's look at how it is defined for addition:

```
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] .
```

First the left operand is visited by passing it the incoming data. Then the right operand is traversed, but this time we pass it the information we obtained from the left operand. This data may be different from the original. After the right operand is also visited, the final data obtained from this traversal is returned together with the new operands, which may be different from the original operands as a result of transformation. The default behavior of `visit` is defined in the same manner for other syntactic units.

Note that the default traversal behavior has the `[otherwise]` attribute. This is a key point in the way that when we would like to have a traversing function doing some particular work on the tree, we just need to define its behavior for the nodes we are interested in. For the other nodes, default behavior will be applied. Below is an example:

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

This operation is defined in module `REPLACE` and it replaces an expression

[9] In [20], transformers are classified as pure transformers and transformers which also collect information. We do not make their distinction.

with another expression. The expression to be replaced is the first element of `replaceData`. Its second argument is the expression that will replace the old one. We only need to define the above equation (and the constant `replace`, and `replaceData`) to make this transformer work. When we are traversing the expression `E`, which we would like to replace, Maude applies this equation and returns `E'` as its replacement. For other expressions, the identity function is applied. This traversal strategy is achieved with the use of the `[otherwise]` attribute.

The optimizers require information about the program to be able to transform the program. For instance, use-def analysis is required to propagate constants. Hence, we would like to be able to store information on nodes. For this purpose we define a new syntax tree node:

```
op info : MExp MInfoData -> MExp .
```

`info` is a closure, which contains an expression and related data, and replaces the expression. Various elements can be stored as part of the encapsulated data, including use-def, gen-kill and type information. So, an analyzer, when traversing the tree, simply leaves some information on the visited node, and continues its traversal. 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 equation in the `TAG` module:

```
 var InfD : MInfoData . var N : Nat .
 vars E BE E' BE' : MExp . vars D1 D2 : MRewData .
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 number from the incoming data, increments the number and recursively traverses the body and the condition, and finally leaves the tag as part of the encapsulated information.

We define various traversers related to program analysis:

- FreeVars: Finds free variables in a method body.
- Info: Encapsulates expressions in `info` packages. Required to keep information on nodes.
- 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 the variable is local, or a field.
- IsFinal: Determines if the accessed field is final.

- CollectDefs: Collects all the definitions found in a tree.
- GetDefs: Finds the definition of a particular variable.
- MayHaveAUse: Determines if a variable may have a use in the program.
- Escapes: Determines if a particular variable escapes from the current method.
- GenKill: Computes gen-kill sets of expressions. See [1] for a detailed explanation of gen-kill sets, why they are useful, and how to compute them.
- UD: Computes use-def chains. Can do *may* or *must* analysis.
- Reset: Removes the analysis results collected so far, and recomputes them by running Tag,Type, IsLocal, IsFinal, GenKill and UD passes, in this given order.

Similar to analyzers, we define transformers. There are mainly two kinds of transformers: those requiring analysis results and the rest. The list of transformers requiring analysis is below.

- Replace: Replaces a particular expression with a given expression.
- Inline: Inlines a particular method call, if possible.
- Unroll: Unrolls a particular loop.
- CopyAssignment: If possible, propagates a variable assigned to another variable.
- ConstantPropagation: Propagates constants.
- NilCheck: Replaces `E equals NIL` with `False` if it is guaranteed that `E` is not `NIL`.
- UselessDef: Removes useless definitions.
- UselessDecl: Removes useless variable declarations from the method header.
- FieldValue: If possible, extracts the values of final fields from objects.
- UselessNew: Removes creation of an unused object, if its constructor is side-effect-free. Recall that the compiler is implemented in a side-effect-free style. Thus, UselessNew can help a lot in optimization.
- Cleanup: Applies Reset, ConstantPropagation, CopyAssignment, NilCheck, FieldValue, UselessDef, UselessNew and UselessDecl in a fixed-point iteration, in this given order. None of these is code-expanding. Therefore cleanup is guaranteed to terminate.
- Auto: Automatically inline methods and unroll loops in a fixed-point iteration.
- SpecializeClass: Uses Auto and Cleanup to create an optimized version of a `'Code` class, which is specialized to a specific quoted-code. SpecializeClass is the top-level operation. It, directly or indirectly, uses other operations to produce the optimized classes.

Other transformers do not need analysis results (i.e. they do not need out-of-context information). Removing redundant blocks and reducing if-

statements are examples of such. Because of the matching capability of Maude, it is trivial to define them. We do not give the full set here, but provide some of them below:

```
eq { Elb ; { Elb2 } } = { Elb ; Elb2 } .
eq if True then E else E' = E .
eq if BE then E else E = { BE ; E } .
eq E and True  = E .
eq E ** 1 = E .
```

In the transformers, Auto may seem dangerous. It automatically inlines methods and unrolls loops until the code does not change anymore. To prevent falling into infinite loops, we use some heuristics. For instance, we do not unroll loops if they are already unrolled and not reduced. When a loop is unrolled, it becomes nested inside an if-statement:

```
do E while BE
```

becomes

```
{ E ; if BE then do E while BE else 0 }.
```

So, if a loop is directly inside an if-statement, we conclude that it was unrolled and we do not unroll it. The heuristic we have for inlining is as follows: If the method is recursive, we inline it only if the target and the arguments are *definable*. An expression is said to be definable if all the variables it contains are definable. A variable is definable if its definition exists within the method we are inside, and if the right-hand-side of this definition is also definable. For example, we say that a variable is non-definable, if it is a parameter and it is not assigned a value inside the method it is passed to.

In Section 5 we will see that, with the given set of optimization rules, it is possible to reduce code generators to the form described in Section 1.

5 Optimization

The analyses and transformations we provided in Section 4.5 are well-known, well-documented [1,15], and exist in many compilers. We argue that the given set is adequate for substantial optimization of program generators at source level. We show several examples. Recall our goal: to obtain code that looks like the hand-written program generator in Section 1.

Our first example is a program fragment defining a method with no free variables.

```
$< method 'square('x : 'int) 'int :
     'x ** 'x
>$
```

The preprocessing stage converts it to the following expression:

```
new 'defineMethodCode(False,            --- method is non-final
  ['square] ,                           --- name of the method
  send new 'LinkedList(NIL,NIL) 'add(   --- param list
```

```
      new 'defineVarCode(['x], ['int])), --- param named 'x
  new 'LinkedList(NIL,NIL),             --- no declared variables
  ['int],                               --- return type
  --- body of the method below: mult. of two vars named 'x and 'x
  new 'binOpCode(["mul"],
                 new 'getNameCode(['x]),
                 new 'getNameCode(['x])))
```

This constructor call creates an object of the `'defineMethodCode` class (a subclass of `'Code`). When the represented quoted-code is to be converted to low-level code, the `'eval` method of the `'defineMethodCode` object is called. Thus, it is this method that we need to optimize. Here is the definition of `'eval` in `'defineMethodCode`:

```
--- 'paramList, 'varList, 'body and 'name are class fields
method 'eval('env : 'Env)
  ['lowlevel : 'string, 'klasName : 'string, 'param : 'List,
   'bodyVal : 'ClosedCode, 'paramName : 'string] 'ClosedCode :
{
  --- prepare environment for this method
  set 'env = send (send (send 'env 'resetForNewMethod())
                  'addList('paramList)) 'addList('varList) ;

  set 'klasName = 'env -> 'currClass ;
  set 'lowlevel = 'klasName # [" - "] # 'name # [" (% 'self"] ;
  --- iterate over paramList to form parameter list in LowLevel
  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()
  } ;
  --- start the body of the function with 'entry basic block
  set 'lowlevel = 'lowlevel # [" ) { 'entry : "] ;
  --- evaluate the body
  set 'bodyVal = send 'body 'eval('env) ;
  --- concatenate everything and return
  set 'lowlevel = 'lowlevel # (send 'bodyVal 'getCode())
                # [" br 'end ; "] # ["'end : return "]
                # (send 'bodyVal 'getName()) # [" ; } "] ;
  new 'ClosedCode('lowlevel, [" $NONAME-FOR-METHOD$ "])
}
```

We cannot optimize this method in itself. But it will occur within a newly created class (produced by the SpecializeClass module) in which some final fields have been given values — in particular, `'body` is the expression that was given as the last argument to the constructor (`new 'binOpCode(["mul"], ...)`). In the context of this specialized class, the `'eval` code can be substantially optimized.

To be specific, SpecializeClass creates a class called `'defineMethodCode--2786` identical to `'defineMethodCode` except for the assignment to some

fields such as `'body`. The call to the `'defineMethodCode` constructor above is replaced by

```
new 'defineMethodCode-2786()
```

The `'eval` method of `'defineMethodCode-2786()` is optimized, producing this greatly improved version:

```
method 'eval('env : 'Env)
  ['lowlevel : 'string, 'tempCode : 'string,
   'enclosingClassName : 'string, 'sym : 'string] 'ClosedCode :
{
  set 'enclosingClassName = 'env -> 'currClass ;
  set 'lowlevel = 'enclosingClassName # [" - 'square (% 'self"] ;
  set 'lowlevel = 'lowlevel # [", % 'x"] ;
  set 'lowlevel = 'lowlevel # [" ) { 'entry : "] ;
  set 'sym = GENSYM ;
  set 'tempCode = 'sym # [" = mul(% 'x, % 'x) ; "] ;
  set 'lowlevel = 'lowlevel # 'tempCode
           # [" br 'end ; 'end : return "] # 'sym # [" ; } "] ;
  new 'ClosedCode('lowlevel, [" $NONAME-FOR-METHOD$ "] )
}
```

This is the code that will be invoked when this object is to be compiled to low level code. This is what we were aiming to get in Section 1. Note that it is not ready to emit code yet: We still need to know the name of the enclosing class to give the function's name, which will be provided by the environment argument when the `'eval` method is called, and we cannot do anything about `GENSYM`'s. But it is as good as we can get. Measured in number of rewrites, this code is approximately four times the speed of the unoptimized version.

If we had a hole in the code, we would have to leave the call to its `'eval` method as it is. Our next example is a quoted expression with a hole:

```
$< ^('hole) ** 10 >$
```

We omit the definition of the `'eval` method of the multiplication operator. The optimized `'eval` method is

```
method 'eval('env : 'Env)
  ['tempCode : 'string,'lowlevel : 'string, 'lval : 'ClosedCode,
   'tempSym : 'string,'sym : 'string] 'ClosedCode :
{
  set 'sym = GENSYM ;
  set 'lval = send 'left 'eval('env) ;
  set 'tempSym = GENSYM ;
  set 'tempCode = 'tempSym # [" = add(0,10) ; "] ; --- constant 10
  set 'lowlevel = ('lval -> 'lowlevelcode) # 'tempCode
                # 'sym # [" = mul("] # ('lval_6 -> 'name)
                # [", "] # 'tempSym # [") ; "] ;
  new 'ClosedCode('lowlevel,'sym)
}
```

In this method, `'left` is a field keeping the value of the left operand of the multiplication operator, i.e. the hole.

We noted earlier that it is difficult to obtain significant performance improvements by source-level optimization of ordinary code, for the simple reason that the programmer has already optimized the code to her satisfaction. So why have we been able to optimize this source code so effectively? The code we are optimizing — most particularly, the body of the method — was machine generated. At the semantic level at which the programmer was operating — writing pieces of quoted code — it is *impossible* to optimize the code. What we have accomplished with these optimizations is to allow the programmer to operate at that level rather than at the level represented by the optimized code — the level of machine instructions.

5.1 Limitation of Optimizations

In the previous examples we successfully obtained the code we were expecting. Now we work on an example for which the existing optimizations fail to produce the ideal code. Consider the following code:

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

The optimized compilation code for this expression is the following (in pseudo code):

```
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 can produce with the existing set of optimizations 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
```

In this code, we have twice the conditions we had in the ideal code. 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 an equation like the following, we would be able to achieve the ideal code.

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

This may seem easy at the first sight, but the expressions we need to prove side-effect-free may include a method invocation on an object which is passed from the client. This necessitates an expensive 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.

5.2 Timing results

We give some more example of optimization, but rather than show the optimized code, we provide benchmarks to illustrate the speedups we obtain.

For benchmarking, we compile both non-optimized and optimized versions of a program. All examples have multiple 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. This gives us two `LowLevel` codes accomplishing the same task. Maude automatically reports the number of rewrites applied in evaluating the code. We report this number as the performance of the program. (It is questionable whether this interpretation of performance would reflect real performance. We provide these benchmarks to give an intuition of how much speedup we might get. So, these numbers should not be taken as representing the real performance.)

5.2.1 Example One: A Class with Two Holes

As the first example, we create a class which has holes in place of a field and the body of a method. The codes for field and body come from separate methods.

```
final class 'ClassGen extends 'object
  method 'incompleteClass('body : 'Code, 'field : 'Code) 'Code :
    $< class 'Gen extends 'object
         ^F('field)
         method 'bar() 'int :
           ^('body)
    >$
  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()
```

The non-optimized version of this program runs in 26219 rewrites. When optimized, the performance is 12256 rewrites. So in this case, run-time generation cost was reduced by 53%.

5.2.2 Example Two: Loop Unroll

As the second example, we give three different, but similar, implementations of loop unrolling (adapted from [10]).

```
final class 'LoopUnroll extends 'object
  --- unrolled code is put into this context
  method 'clientContext ('body : 'Code) 'Code :
    $< class 'Generated extends 'object
         field 'x : 'int
         method 'unrolled() 'int :
           ^('body)
    >$

  --- Unroll 1 ---
  method 'test1('n : 'int) 'string :
    send (send self 'clientContext(
          send self 'unroll1('n, $< 99 >$))) 'generate()

  --- repeat the given code 'c for 'n times.
  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 'clientContext(
          send self 'unroll2('n, $< 99 >$, $< 'x >$))) 'generate()

  --- this time keep track of an iteration variable
  method 'unroll2('n : 'int, 'c : 'Code, 'itervar : 'Code) 'Code :
    if ('n equals 0) then
        $< set ^('itervar) = 0 >$
    else
        $< { ^(send self 'unroll2('n -- 1, 'c, 'itervar)) ;
             set ^('itervar) = ^('itervar) ++ 1 ;
             ^('c) }
        >$
```

```
  --- Unroll 3 ---
  method 'test3('n : 'int) 'string :
    send (send self 'clientContext(
          send self 'unroll3('n, new 'CodeFun())))  'generate()

  --- the code to be repeated is obtained from a function
  method 'unroll3 ('n : 'int, 'F : 'CodeFun) 'Code :
    if ('n equals 0) then
      $< 0 >$
    else
      $< { ~(send self 'unroll3('n -- 1, 'F)) ;
           ~(send 'F 'iteration('n)) } >$

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

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

We invoked all three `'test` methods, with various numbers of iterations to unroll. Table 1 shows that run-time generation cost was reduced by up to 60%.

Test	n	Original	Optimized	Speedup(%)
1	1	17577	9286	47
1	10	52722	23164	56
1	100	404172	161944	60
1	200	794672	316144	60
2	1	30897	15871	49
2	10	139572	65002	54
2	100	1226322	556312	55
2	200	2433822	1102212	55
3	1	17743	9611	46
3	10	54958	26540	52
3	100	427108	195830	54
3	200	840608	383930	54

Table 1
Results of loop-unrolling test.

6 Related Work

In Mumbo, we have used the `info` closure to store data-flow information on the nodes. We also defined the default traversal behavior for each syntactic unit, so that whenever we want to write a new traverser, we only need to give the behavior for the nodes we are interested in. This approach is similar to the Visitor pattern in object-oriented programming [9].

In [20], traversal functions are discussed. In this approach, built-in traversal strategies can be used for operations, so that the implementation of the default behavior is machine generated, instead of being done by the programmer. In strategic programming, these traversal behaviors can be expressed by strategies [21].

In [17,5], scoped dynamic rules are discussed. This approach allows adding and removing rules dynamically. This way, optimizations, such as constant propagation and copy assignment, can be implemented more concisely. In some cases, the need for analysis results disappears because the control flow information can be expressed by dynamic rules. Furthermore a multiple-pass task can be done in a single pass. Although, from the programmer point of view, scoped dynamic rules are more complicated to control than usual rules, and transformations may be more difficult to design, they potentially decrease the load of the programmer significantly. We are curious about how our transformations in Mumbo would be expressed using scoped dynamic rules.

7 Conclusion

As we have noted, we were far more successful in optimizing Mumbo than we have thus far been in optimizing Jumbo. To see if we can draw any lessons from the former that can be applied to the latter, we need to consider what works in Mumbo, and why.

The first lesson from Mumbo is that its operations (the subclasses of `'Code`) are *side effect-free*, i.e. all fields are final. This has a deep impact on optimizability. Its central advantage is that we can use the value that has been assigned to a field. When side effects are present, the conditions under which this can be done — that is, under which we can guarantee that the field has not been modified since its assignment — are so constrained that it is often impossible. We have begun to look at restructuring Jumbo to make as many fields final as possible. (It is sometimes argued that the functional programming style can be as efficient as imperative programming, but it is rarely argued that it can be *more* efficient; this may be a case where it is.)

Another lesson of Mumbo is related to the use of named classes. Jumbo makes heavy use of anonymous classes when defining subclasses of the `Code` class. It seems a good programming practice but anonymous classes may cause a variety of difficulties in program analysis and optimization. Mumbo's SpecializeClass transformation shows how named classes can be used in place

of anonymous classes while eroding those difficulties.

We believe that Mumbo is an effective application of rule-based programming. Defining Mumbo in Maude brought convenience especially in implementation of the optimizations.

To summarize, we have presented a language with an RTPG facility, called Mumbo, a compiler for that language, and a set of optimizations for RTPG. The language is meant as a simplified model of Jumbo, a compiler for Java. Mumbo can be very helpful as a "testbed" to quickly experiment with new ideas that would otherwise take significant time if implemented in Jumbo. Using Maude, we have been able to quickly prototype Jumbo and experiment with source-level optimizations. This has produced valuable insights that we are now rolling back into Jumbo.

Acknowledgement

When implementing syntax and semantics of Mumbo and `LowLevel`, we adapted code pieces from class notes of Prof. Grigore Roşu. Anonymous reviewers provided very valuable feedback on a previous version of this paper. Philip Morton suggested the name "Mumbo". The `.maude` files for Mumbo are available at http://pinatubo.cs.uiuc.edu/~aktemur/mumbo.

References

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

[2] Aktemur, B., "A Rule-Based Model of a Run-time Program Generation System," Master's thesis, University of Illinois at Urbana-Champaign (2005).

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

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

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

[6] Clausen, L., "Optimizations In Distributed Run-time Compilation," Ph.D. thesis, University of Illinois at Urbana-Champaign (2004).

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

[8] Engler, D. R., W. C. Hsieh and M. F. Kaashoek, *'C: A Language for High-level, Efficient, and Machine-independent Dynamic Code Generation*, in: *POPL '96:*

Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1996), pp. 131–144.

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

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

[11] Kamin, S., *Program Generation Considered Easy*, in: *PEPM '04: Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation* (2004), pp. 68–79.

[12] Kamin, S., L. Clausen and A. Jarvis, *Jumbo: Run-time Code Generation for Java and its Applications*, in: *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization* (2003), pp. 48–56.

[13] *Low Level Virtual Machine*, http://llvm.cs.uiuc.edu.

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

[15] Muchnick, S., "Advanced Compiler Design and Implementation," Morgan Kaufmann, 1997.

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

[17] Olmos, K. and E. Visser, *Composing Source-to-Source Data-Flow Transformations with Rewriting Strategies and Dependent Dynamic Rewrite Rules*, in: R. Bodik, editor, *14th International Conference on Compiler Construction (CC'05)*, Edinburgh, 2005.

[18] Roşu, G., *CS422 Class Notes*, http://fsl.cs.uiuc.edu/~grosu/classes/2004/fall/cs422/ (2004).

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

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

[21] Visser, E., *A Survey of Rewriting Strategies in Program Transformation Systems*, in: B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, Electronic Notes in Theoretical Computer Science **57** (2001).