Source-to-Source Code Generation Based on Pattern Matching and Dynamic Programming

Weimin Chen[†], Volker Turau[‡]

TR-93-047
August, 1993

Abstract

This paper introduces a new technique for source-to-source code generation based on pattern matching and dynamic programming. This technique can be applied to all source and target languages which satisfy some requirements. The main differences to conventional approaches are the complexity of the target language, the handling of side effects caused by function calls and the introduction of temporaries. Code optimization is achieved by introducing a new cost-model. The technique allows an incremental development based on improvements of the target-library. These require only a modification of the rewriting rules since those are separated from the pattern matching algorithm. Experience of an successful application of our technique is given.

† GMD-IPSI, Integrated Publication and Information Systems Institute, Dolivostr. 15, 64293 Darmstadt, Germany.
E-mail: chen@darmstadt.gmd.de

‡ On leave from: Fachhochschule Giessen-Friedberg, Fachbereich MND, Wilhelm-Leuschner-Str. 13, 61169 Friedberg, Germany.
E-mail: turau@prfhfb.fh-friedberg.de

1. Introduction

In this paper a new technique for generating optimized code for object-oriented programming languages based on pattern matching and dynamic programming is presented. It can be applied to translation on a source to source basis. Pattern matching and dynamic programming have been applied before in compilers based on the back-end approach, i.e. where the target code of the compiler consists of hardware-oriented instructions. However, when applying this technique to target languages which are high level programming languages, several essential differences need to be overcome:

- the target language is far more complex than machine code,
- the evaluation order can not be guaranteed by a simple traversal strategy for the syntax tree, since function calls may cause side effects,
- temporal variables must be introduced, and
- a new cost–model for code optimizing is needed.

Our technique is based on a set of rewriting rules, which allow us to handle functions with a variable number of arguments as well as temporal variables. A proof is given, that the evaluation order in the generated code is consistent with that of the source code.

The code generation starts from the syntax tree and is done in the following way. All templates in the tree–rewriting rules are matched against the subtrees of the syntax tree during a depth–first traversal of the tree. At each node, the costs are used to determine the best match and, the selected subtree is replaced by the associated replacement node. The dynamic programming algorithm allows the rules to be written in any order and obviates the need to deal with pattern matching conflicts. It produces code that is optimal with respect to the cost provided.

Compilers based on the front-end approach are used to save the big effort to generate machine code instead the translation is into another programming language for which a compiler exists [4, 15]. This approach is very useful for compilers of new languages, because it allows a flexible and extensible implementation. Furthermore, it is platform independent.

The general situation is depicted in figure 1. The front-end compiler performs syntax and semantic analysis. Then the phase of code generation starts. The output of this phase is a semantically equivalent program in the target language which contains calls to function, which are provided by a target library. The target compiler utilizes this library to generate executable code.

The introduced technique can be applied to all source and target languages which satisfy some requirements. The problems which arise when this techniques are applied in the front-end approach are dicussed and are of independent interest.

The remainder of this paper is organized as follows. Section 2 states the background for the source- and target-languages and details the translation rules. Section 3 outlines the general principle of our technique and presents the new features. Section 4 discusses the control facility in the pattern matching. Section 5 reports the experience gained in applying the new technique. Section 6 surveys related work and section 7 concludes the paper with a discussion of our results.

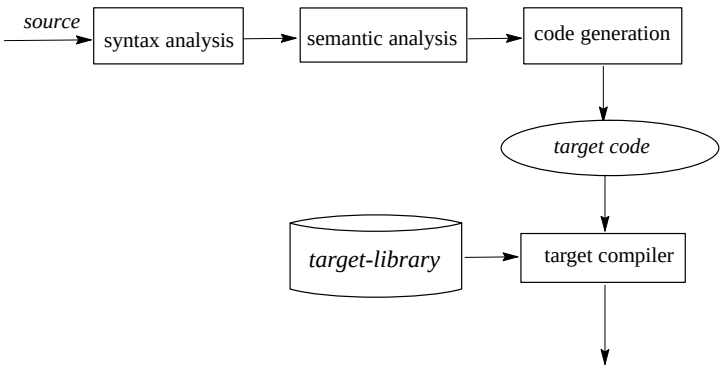


Fig. 1. The module structure of the front-end compiler

2. Background

In this section the requirements of the source- and target-languages and of the translation rules are discussed. The focus of our work is on the basic concept of object-oriented programming languages: the class-concept. A class defines *properties*[1] and *methods*.

Source language:

- The language is strongly typed. It can be assumed that the properties of the classes are globally identified during the phase of semantic analysis, where rename techniques are required. E.g. a property-reference can be written as p@expr, where the expression expr is an instance of a class while p is a property of that class. In the phase of semantic analysis, the type of expr can be statically determined so that it is possible to globally identify all properties of all classes. Let $\mathcal{P}$ be the set of all globally identified properties.
- The methods of a class may cause side effects changing the values of properties in $\mathcal{P}$. Most object-oriented languages support *dynamic dispatching*. As a consequence, in general it is not possible to statically bind a generic method invocation to a concrete method at compilation-time. For a strongly typed language, however, it must be possible to determine the set of methods (in the corresponding classes) which *may* be invoked for the given (generic) method invocation. At compile-time, we can globally identify the methods of all classes, and define the set of these methods as $\mathcal{M}$. For every $m \in \mathcal{M}$ let $\mathcal{P}(m) \subseteq \mathcal{P}$ be the set of those properties which *may* be effected by the method m. Let $\mathcal{M}_D(m) \subseteq \mathcal{M}$ be the *dispatching-set* of those methods which *may* be dispatched for a given method invocation $m \in \mathcal{M}$. We denote
  $$\mathcal{M}^* = \{\{m\} \mid m \in \mathcal{M}\} \cup \{\mathcal{M}_D(m) \mid m \in \mathcal{M}\}.$$
  The set $\mathcal{M}^* \subseteq \mathbf{2}^{\mathcal{M}}$ can be constructed at compilation-time. Thus, for each (generic) method invocation $m \in \mathcal{M}$, we can always find the unique corresponding element $m^* \in \mathcal{M}^*$ at compilation-time, depending on the context that method m is stated as dynamic dispatched or not. Therefore, for each (generic) method invocation $m \in \mathcal{M}$, let $\mathcal{P}^*(m)$ be the set of those properties which may be effected by an invocation of m:
  $$\mathcal{P}^*(m) = \bigcup_{m' \in m^*} \mathcal{P}(m')$$
  where $m^* \in \mathcal{M}^*$ corresponds to m.
  In the following discussion, a method invocation is written as $m(arg_0, \ldots, arg_n)$[2], where $m \in \mathcal{M}$ is globally identified and $\mathcal{P}^*(m)$ is already constructed at compilation-time.
- The body of each method $m \in \mathcal{M}$ may define local variables, which are available in the method body only. Let $\mathcal{V}(m)$ denote the set of these local variables defined in method m. In contexts that do not lead to ambiguity, we denote this set as $\mathcal{V}$.

Target language:

- The language supports the mechanism of function invocation.
- Functions are allowed to have a variable number of arguments.
- The language supports function polymorphism. Therefore, for a source expression we can ignore the types of the operands when applying pattern matching to code generation. E.g. the polymorphic function
  `VOID writeProp(STRING, OBJECTS, TYPE)`,
  where **`TYPE`** can be any target type, represents writing the v_0 to p@v by
  `writeProp("p",v,`v_0`)`,
  regardless of what type of v_0 is.
- The language supports type casting.

Translation rules:

- All source data types of the same kind are mapped into the same target data type. Hence, only a fixed set of target data types is defined in the target language. For example, all source class-types are mapped into a single target type: `OBJECTS`. All source array-types are mapped into a single target type: `ARRAY`. Each primitive source type is mapped into an individual target type, e.g. INTEGER is mapped into `INT`.
- The body of each method is translated into a corresponding function.
- There exists a set of functions of the target language which represent the basic *statements*/*expressions* of the source language. These are provided in the interface of the target-library shown in Fig. 1.

1. Different languages have different concepts, e.g. in C++ it is called *member attribute*, in Smalltalk it is called *instance variable*.

2. Some languages, e.g. C++, write a method call as $o.m(arg_1, \ldots, arg_n)$. By syntax transformation, it is possible to write that as $m(o, arg_1, \ldots, arg_n)$.

- All property accesses are translated into function calls in which the properties are passed as identifier-reference arguments. The above function `writeProp` is an example. Another example is the function

  VOID `readProp(`*STRING*`,` *OBJECTS*`)`

  which represents reading the source expression p@v by

  (***TYPE***)`readProp("p",v)`.

  where the word ***TYPE*** is the name of the target type of the expression p@v, and is needed for type casting.
- The unit of the translation is a statement. Each statement is translated into one or more target statements.

It is not the intention of this paper to cover all object-oriented paradigms. Rather the focus is on source-to-source translation at the statement/expression level.

In the following, the source/target code is represented by Helvetian/`courier` font respectively. Letters p, v, and m to denote the elements of $\mathcal{P}$, $\mathcal{V}$, $\mathcal{M}$ respectively.

3. Principle of pattern matching

In the conventional approaches for code generation which are based on pattern matching, the machine description is separated from the code generation algorithm. They employ a formal machine description and use a code-generator-generator to automatically code generators. In this scheme, pattern matching is used to replace interpretation with case analysis. Rewriting rules may be tree-structured [3] or linear [7]. Correspondingly, pattern matching is performed by heuristic search or parsing.

3.1. Rewriting rules

A pattern matching system *R* is a finite set of *rewriting rules*, each of the form $l \leftarrow r$, where *l* and *r* are terms built from the set of labelled symbols and a set of *patterns*.

We use tree structures to represent the patterns. To illustrate this idea, consider an example.

Source expression: $p@v_1$

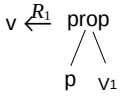
Syntax tree:

Rewriting rule:

R_1: $v \leftarrow$ prop (p, v_1) $\quad v \leftarrow$ (prop↑ttp)`readProp("p",`v_1`)`

(where prop↑ttp denotes the target type (ttp) at node prop)

Pattern matching:



Target code:

$v \leftarrow$ (***TYPE***)`readProp("p",`v_2`))`

where ***TYPE*** denotes the value of prop↑ttp.

In this example, suppose that the variable v_1 in the target code has been declared corresponding to the source variable v_1, in an appropriate mapping.

Consider another example. Suppose the polymorphic function

TYPE `sum(`***TYPE***`,...)`,

where ***TYPE*** is an appropriate target type,

represents the source expression $v_1+v_2+ \ldots + v_n$ by

`sum(`v_1`,`v_2`, ...,`v_n`)`

for any $n > 1$. This code can be matched by the following patterns:

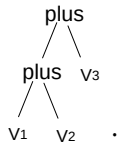
R_2: $v \leftarrow$ plus (β_1, β_2) $\quad v \leftarrow$ `sum(`β_1`,`β_2`)`

R_3: $\beta \leftarrow$ plus (β_1, β_2) $\quad \beta \leftarrow \beta_1,\beta_2$

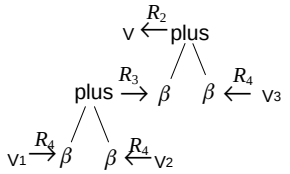
R_4: $\beta \leftarrow v \quad \beta \leftarrow v$

In the above rules, β denotes a nonterminal symbol while the code of β consist of arguments for a function call.

E.g. given an expression $v_1+v_2+v_3$, the syntax tree is



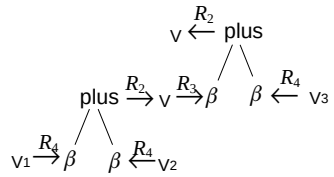
The matching procedure can be



so that the target code is

$v \leftarrow$ `sum(`v_1`,`v_2`,`v_3`).` (1)

Note that the matching strategy is not unique. Another matching procedure can be



so that the target code is

$v \leftarrow$ `sum(sum(`v_1`,`v_2`),`v_3`).` (2)

As a preliminary estimation, code (1) is better than code (2), since in (1) only one function invocation is required.

As another example consider an array with multiple dimensions. Suppose a target function

VOID `arr(`*ARRAY*`,`*INT*`,...)`

represents reading a source expression $v_0[v_1][v_2]...[v_n]$ by

(***TYPE***)`arr(`v_0`,`v_1`,...,`v_n`)`

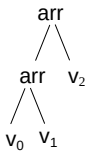
where ***TYPE*** is the target type of the source expression. The corresponding rewriting rules are defined as follows:

R_5: $v \leftarrow$ arr (θ, v_1) $\qquad v \leftarrow$ (arr↑ttp)`arr(`θ`,`v_1`)`

R_6: $\theta \leftarrow$ arr (θ_1, v) $\qquad \theta \leftarrow \theta_1$`,`$v$

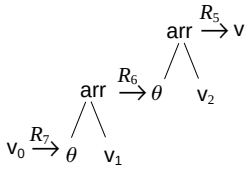
R_7: $\theta \leftarrow v$ $\qquad \theta \leftarrow v$

E.g. given an expression $v_0[v_1][v_2]$, the syntax tree is



(3)

The matching procedure can be



and the corresponding code is

$v \leftarrow$ (***TYPE***)`arr(`v_0`,`v_1`,`v_2`).`

Again there are several different matching procedures, e.g. another matching will generate the code

$v \leftarrow$ (***TYPE***)`arr((`*ARRAY*`)arr(`v_0`,`v_1`),`v_2`).`

3.2. Costs of rules

In the conventional approach, each rewriting rule has a designated cost, so that the optimization degree for the target code can be measured by the total cost of rules matched. In the conventional approach, the target code of each rewriting rule consists of hardware-oriented instructions. It is relatively easy to designate a cost to a rule, because the instructions are relatively simple. However, in our front-end approach where the target code is composed of a series of function calls, the cost of a rule relies on how the target functions are implemented. Generally speaking, the cost of a rule can be estimated either by analyzing the function implementation, or by counting up the average execution time corresponding to the individual function calls.

In some situations, the cost of a general function call depends not only on the function implementation itself, but also on what the actual arguments are. For example, suppose function `union(`SET_1`,`SET_2`)` is the target code for a source expressions that builds the union of two sets. Thus, the cost of the function `union` will dynamically depend on the space-complexity of the arguments. Unfortunately, analyzing this space complexity at the compilation-time is a difficult task and in some cases impossible. This fact leads to an approach where each function call gets assigned an estimated average cost regardless of the dynamic space complexity of the arguments.

There are two kinds of target functions: functions with a fixed arity, and functions with a variable arity. In the first case, the cost of a function call is a fixed nonnegative value; in the second case, the cost of a function call depends linearly on the number of the arguments. For instance, consider the function `sum` where a variable number of arguments is allowed. The cost of this function call can be calculated by the following formula:

$$cost(\texttt{sum}(\texttt{arg}_1, \texttt{arg}_2, ..., \texttt{arg}_n)) = c_0 + n \times c_a$$

where c_0 is the cost in the case no arguments are present, and c_a is the incremental cost for each new appended argument.

This linear model is motivated by the following fact: in the body of a function, a loop is always used in order to access iteratively the arguments appearing in the function call. Normally, the number this loop is executed linearly depends on the number of the arguments. Meanwhile, the cost of the function call will linearly depend on the number of these runs.

For the expression $v_1+v_2+v_3$ two different target codes were generated. By the above model, the costs of alternatives (1) and (2) are $c_0 + 3c_a$ and $2c_0 + 3c_a$, respectively. Therefore, as we have stated, alternative (1) is better than (2). According to this model, the costs of rules R_2, R_3, and R_4 must be designated as c_0, 0, and c_a, respectively.

The discussion above shows that a target function call may be generated by matching a sequence of rewriting rules. Therefore, the cost of a function call is equal to the sum of the costs of the corresponding rules.

In order to simplify the cost model, the set of rewriting rules is restricted in a way that only the root can have variable arity. In general, the cost of a rule is a nonnegative val-

ue, which may depend on the arity of the root of the tree-pattern, if the root has variable arity.

3.3. Evaluation order

In a syntax tree, the evaluation order among its nodes is always fixed, e.g. left-right or right-left. On the other hand, the evaluation order in the target code is also fixed. It is necessary to ensure that both evaluation orders are conformable.

We denote the set of rewriting rules by Σ. Let $\mathsf{operand}_1$ and $\mathsf{operand}_2$ be two arbitrary different operands of the syntax tree. Let $\mathtt{arg}_1$ and $\mathtt{arg}_2$ be the corresponding operands of the target code. Now two kinds of orders are considered. One is the evaluation order between $\mathsf{operand}_1$ and $\mathsf{operand}_2$ in the syntax tree. The other one is the evaluation order between $\mathtt{arg}_1$ and $\mathtt{arg}_2$ in the target code. We say, Σ is *consistent*, if these orders are conformable for all pairs of operands.

In the context of this paper, we assume that the evaluation order in the syntax tree is right-left and in the target code the arguments of a function call are also evaluated in right-left order. A sufficient criteria to ensure that the set of patterns is consistent is that if an operand appears on the right-hand side of another one then the code corresponding to the first operand must appear on the right-hand side of the code corresponding to the second one.

It is clear that a non-consistent set of rewriting rules or illegal traversal order of the syntax tree can result in a target code with an illegal evaluation order. However, the correct traversal order of the syntax tree as well a consistent set of rewriting rules are still insufficient to ensure the correct evaluation order in the target code by any pattern matching.

The problem is the following. A method $\mathsf{m} \in \mathcal{M}$ may cause side effects modifying some properties in $\mathcal{P}^*(\mathsf{m})$. If a statement or expression contains $\mathsf{m} \in \mathcal{M}$ as well as some properties of $\mathcal{P}(\mathsf{m})$, then the evaluation order between m and the properties in $\mathcal{P}^*(\mathsf{m})$ must be conformable between the source- and the target-code. This will be illustrated by an example.

On the one hand, the target code of $\mathsf{m} \in \mathcal{M}$ can be matched by the following rules:

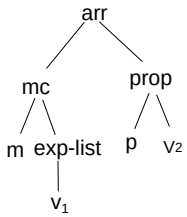
R_8: $\mathsf{v} \leftarrow \mathsf{mc}(\mathsf{m}, \zeta)$ $\quad \mathsf{v} \leftarrow \mathtt{m}(\zeta)$

R_9: $\mathsf{v} \leftarrow \mathsf{mc\text{-}dpth}(\mathsf{m}, \zeta)$ $\quad \mathsf{v} \leftarrow \mathtt{dispatch}(\mathtt{"m"}, \zeta)$

R_{10}: $\zeta \leftarrow \mathsf{exp\text{-}list}(\mathsf{v}^*)$ $\quad \zeta \leftarrow \mathsf{v}^*$

R_8 and R_9 describe the rules for method calls in the context of static binding and dynamic dispatching. In R_{10}, the iteration notion "*" indicates that the arity of node exp-list is variable and ≥ 0. Correspondingly, the code of ζ consists of the concatenation of the code son's code. Later this will form the arguments of a function call.

Therefore, given an expression $\mathsf{m}(\mathsf{v}_1)[\mathsf{p}@\mathsf{v}_2]$, the syntax tree is



(4)

Matched by rules R_1, R_5, R_7, R_8, and R_{10}, the target code is

$$\mathsf{v} \leftarrow (\textbf{\textit{TYPE}})\mathtt{arr}(\mathtt{m}(\mathsf{v}_1), \mathtt{readProp}(\mathtt{"p"}, \mathsf{v}_2)). \quad (5)$$

Consider the case that the following new rule

R_{11}: $\mathsf{v} \leftarrow \mathsf{arr}(\mathsf{v}_1, \mathsf{prop}(\mathsf{p}, \mathsf{v}_2))$ $\quad \mathsf{v} \leftarrow \mathtt{arr}(\mathsf{v}_1, \mathtt{"p"}, \mathsf{v}_2)$

is introduced in order to translate the source expression $\mathsf{v}_1[\mathsf{p}@\mathsf{v}_2]$ into

$$\mathsf{v} \leftarrow (\textbf{\textit{TYPE}})\mathtt{arr}(\mathsf{v}_1, \mathtt{"p"}, \mathsf{v}_2)$$

Here the property p is dereferenced in the function body $\mathtt{arr}$ but the value of v_1 is evaluated in the argument part, i.e. p is evaluated after v_1. In this case the evaluation order is not sensitive, so that the code is still correct.

However, if the tree (4) is matched by the rules R_8, R_{10} and R_{11}, then the target code becomes

$$\mathsf{v} \leftarrow (\textbf{\textit{TYPE}})\mathtt{arr}(\mathtt{m}(\mathsf{v}_1), \mathtt{"p}_2\mathtt{"}, \mathsf{v}_2). \quad (6)$$

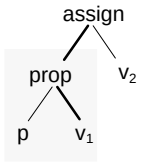
In the case $\mathsf{p} \in \mathcal{P}^*(\mathsf{m})$, thus the evaluation order between the method m and the property p is illegal.

This example illustrates that the problem of illegal evaluation order does not result from the traversal strategy for the syntax tree, but *does* result from the control in pattern matching.

In this source-to-source background, the issue of evaluation order comes from method calls which may cause side effects. Besides this case, the evaluation order is not sensitive where the code generator may evaluate expressions in an order that reduces the cost of evaluation.

3.4. Temporal variables

So far we merely considered the rewriting rules which are used to read expressions. However, for rules involving reading and writing of variables or properties a problem arises. Consider the source statement $v_1.p:=v_3$ with the following syntax tree

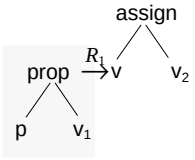


(7)

As we assumed in section 2, the target code must be

`writeProp("p",`v_1`,`v_2`).`

However, if rule R_1 is chosen to match the shadowed part as follows



the target code would be wrong, since the code of R_1 just returns the value of $v_1.p$ and does not perform the actual update. Consequently, the target code only changes the (hidden) temporary v but $v_1.p$.

4. Control in pattern matching

Several tree-pattern matching algorithms have been presented. For code generation applications, a scheme proposed by Hoffman and O'Donnell [13] appears promising, and it is successfully applied in the system *Twig* for code generator-generators by Aho, Ganapathi, and Tjiang [3]. They suggested that template matching can be done efficiently by extending the Aho-Corasick multiple-keyword pattern-matching algorithm [1] into a top-down tree-pattern matching algorithm. Afterwards, applying the bottom-up heuristic search, the cheapest tree-pattern matching can be found. This algorithm can be extended to our application. In this paper we do not intend to discuss the details of the algorithm. Instead, we state some key problems which occur in our application, but do not occur in the conventional approaches and present our solutions.

4.1. Controlling the evaluation order

Controlling the evaluation order can be divided into two categories. One is the order between *methods and methods*. Another one is the order between *methods and properties*.

Without loss of generality, in this section we can assume that the method call is always in the context of static binding. From the view of controlling the evaluation order, a method call in the context of static binding or dynamic dispatching does not make an essential difference.

In the syntax tree, we introduce the notion $n \succ m$ if node m is an ancestor of node n, and the notion $n \succcurlyeq m$ if $n=m$ or $n \succ m$.

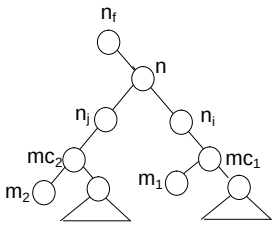
We have the following results.

THEOREM 1. Suppose the set of rewriting rules is consistent and that only rules R_8 and R_9 contain nodes for method calls. Then for any pattern matching the evaluation order between methods and methods is always correct.

PROOF: Let $m_1, m_2 \in \mathcal{M}_b$ and mc_1 and mc_2 their corresponding nodes in the subject tree c. Let the traversal of the subject tree be the bottom-up and the right to left order. Suppose node mc_1 appears before node mc_2. Now we prove that, by any pattern matching, the code of m_1 is always evaluated before m_2.

Clearly in the case that $mc_1 \prec mc_2$ the code for m_1 is embedded in the argument of a function call which is the code for node m_2. Hence, the code for m_1 is evaluated before that for m_2.

Note that the case $mc_1 \succ mc_2$ can not occur. Consider the remaining case of $mc_1 \not\succ mc_2$. Suppose node n is the first common ancestor of nodes mc_1 and mc_2. By any matching, there always exists a node n_f ($n_f \preccurlyeq n$) such that at n_f there is a function call f generated. According to the assumption that only the rules R_8 and R_9 contain nodes for method calls, the code of m_1 and m_2 must be embedded in two different arguments, arg_i and arg_j. Moreover, arg_i and arg_j are not identifier-references (c.f. the definition of rule R_8). Suppose the code arg_i and arg_j are generated at nodes n_i and n_j respectively. Hence, the relation $n \prec n_i \preccurlyeq mc_1$ and $n \prec n_j \preccurlyeq mc_2$ holds, while n_i must be on the right-hand side of n_j since mc_1 is on the right-hand side of mc_2.



Therefore, in the function call f, argument arg_i must be on the right-hand side of argument arg_j so that all code in arg_i is evaluated before code in arg_j, hence the code of m_1 is evaluated before m_2. ❑

Before discussing how to control the evaluation order between methods and properties, we introduce some notations. For each node n in the syntax tree let $\mathcal{M}(n) \subseteq \mathcal{M}$ be the set of those methods which occur in the subtree rooted at node n. Moreover, at node n, let $\mathcal{P}^*_{\mathcal{M}}(n)$ be the set of those properties which may be effected by the methods in $\mathcal{M}(n)$, i.e.

$$\mathcal{P}^*_{\mathcal{M}}(n) = \bigcup_{m \in \mathcal{M}(n)} \mathcal{P}^*(m) \ .$$

At node n, denote the target code associated with label l as *code*(l, n). Here we can regard *code*(l, n) as a form of some arguments[3]. Let $\mathcal{P}^*_{code}(l, n) \subseteq \mathcal{P}$ be the set of those properties which occur as identifier-references arguments in *code*(l, n). For example if *code*(l, n) is

$$\texttt{"}p_1\texttt{"},\texttt{f(v,"}p_2\texttt{")},\texttt{"}p_3\texttt{"}$$

then $\mathcal{P}^*_{code}(l, n) = \{p_1, p_3\}$. Note that p_2 does not belong to $\mathcal{P}^*_{code}(l, n)$ because `f(v,"`p_2`")` is an argument and but `"`p_2`"` is not.

THEOREM 2. Suppose the set of rewriting rules is consistent and that only rules R_8 and R_9 contain nodes for method calls. Given a syntax tree $\mathcal{T}$, suppose $\mathcal{T}$ is matched by a set $M = \{[R_i', n_i]\}$ where the root of rewriting rule R_i' matches at node n_i in $\mathcal{T}$. Assume that label l_i^j is the *j*–th leaf (counted in left-right order) of R_i' and matches at node n_i^j in $\mathcal{T}$. The target code generated by *M* has a correct evaluation order between methods and properties iff the following condition holds:

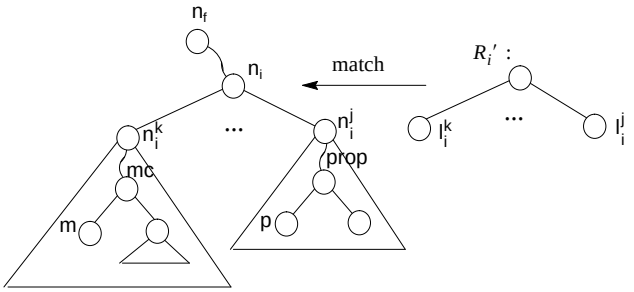
$$\mathcal{P}^*_{\mathcal{M}}(n_i^k) \cap \mathcal{P}^*_{code}(l_i^j, n_i^j) = \varnothing,$$
for all i, j and k such that $k < j$.

PROOF: In the syntax tree $\mathcal{T}$, let $p \in \mathcal{P}$, $m \in \mathcal{M}$ and let mc be the corresponding mc-node. Assume that the code of m and the code of p are involved in a (nested) function call. It is easy to verify, that the code of p occurs on the right hand side of m iff node p appears before the node mc during the traversal of the subject tree $\mathcal{T}$ in the bottom-up and the right-left order.

Sufficiency: According to the restriction that in the set of rewriting rules only rules R_8 and R_9 contain nodes for method calls, if the code of p occurs on the left-hand side of m, then their evaluation order is always correct, since p is always evaluated after m. Now assume that the code of p occurs on the right-hand side of m and $p \in \mathcal{P}^*(m)$, while in the target code the content of p is evaluated after m. This means that there is a function call in the form of

`f(...,code`$_m$`,...,"p",...)`,

where `code`$_m$ acts as an argument of `f` and directly or indirectly includes the code of m. Therefore, there must be a match like the following:



where the function call `f` is generated at node n_f and $k < j$. Now $\mathcal{P}^*(m) \subseteq \mathcal{P}^*_{\mathcal{M}}(n_i^k)$ and $p \in \mathcal{P}^*_{code}(l_i^j, n_i^j)$. This leads to a contradiction since $p \in \mathcal{P}^*_{\mathcal{M}}(n_i^k) \cap \mathcal{P}^*_{code}(l_i^j, n_i^j) \neq \varnothing$.

Necessity: Assume that there is a match as above such that $\mathcal{P}^*_{\mathcal{M}}(n_i^k) \cap \mathcal{P}^*_{code}(l_i^j, n_i^j) \neq \varnothing$, i.e. there exists a p in the subtree rooted at node n_i^j (so that $p \in \mathcal{P}^*_{code}(l_i^j, n_i^j)$) and a node mc in the subtree rooted at node n_i^k (such that $p \in \mathcal{P}^*(m) \subseteq \mathcal{P}^*_{\mathcal{M}}(n_i^k)$). By this match, there is a function call of the form

`f(...,code`$_m$`,...,"p",...)`

generated. Similar to the first part of the proof, the code `code`$_m$ acts as an argument of `f` and directly or indirectly includes the code of m. Consequently, in the function call `f`, m is evaluated before p. This leads to a illegal evaluation order between m and p which is a contradiction. ❑

In an algorithm, $\mathcal{P}_{\mathcal{M}}(n)$ can be calculated recursively in a bottom-up order for each node n in the syntax tree $\mathcal{T}$. It is independent of any pattern matching. On the other hand, the value of $\mathcal{P}^*_{code}(l, n)$ must be kept at node n associated with the label l during pattern matching. In the conventional approaches, the bottom-up heuristic search is applied, where at each node n associated with the relevant label l only the cheapest match of all rewriting rules $l \leftarrow r$ is stored [3, 13]. In our algorithm, however, at each node n associated with the relevant label l, different matches that can generate different $\mathcal{P}^*_{code}(l, n)$ must be stored (due to theorem 2). As a consequence, our algorithm for pattern matching will use more space than the conventional one.

3. In the case that the code is a function call, we can regard the whole function call as an argument

4.2. Handling of temporal variables

One way of handling updates for temporal variables is to apply the pattern matching only to certain subtrees of the syntax tree.

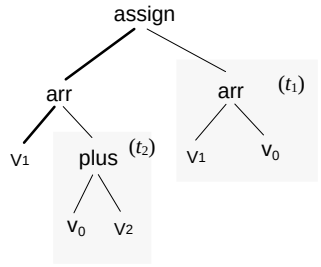
Given a source statement which modifies an object, the path from the node corresponding to the modified object to the node corresponding to the modifying operation[4] is called *writing-path.*

If a syntax tree includes a writing-path, then pattern matching is only applied to those subtrees which do not include nodes of the writing-path. In other words, for the subtree which includes nodes of the writing-path, code generation must be performed by traditional case analysis, since the current matching strategies are not suited for syntax trees containing writing-paths.

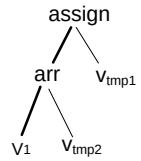
Informally speaking, given a syntax tree which includes a writing-path, pattern matching is applied to the subtrees obtained after deleting the nodes of the writing-path as well as the edges leading to those nodes.

For example, in the tree (7), the bold path is the writing path so that pattern matching is only applied to the subtrees p and v_2 (they are already nodes).

For a more illustrative example consider the statement, $v_1[v_0+v_2] := v_1[v_0]$. The syntax tree is



where the bold path is the writing-path. Thus, pattern matching is applied to the shadowed subtrees (t_1) and (t_2). Afterwards, the syntax tree becomes



which must be treated by traditional case analysis.

4. We do not regard that the node of method call `mc` is a modified operation, although certain method calls can cause side effect.

5. Experience

We implemented this front-end approach successfully to translate the schema language *VML* into C++ [5]. The general issues of the *VML* compiler are described in [16]. *VML* is the modelling language of the object-oriented database system *VODAK* developed at GMD-IPSI. The target language C++ satisfies the requirements stated in section 2. In fact, from the point of view of compilation at statement level, we did not rely on other object-oriented features of C++. By this experience several notes can be concluded as follows.

First, assigning costs to rewriting rules is a difficult task and may undergo several revisions in order to reflect the changes during the system maintenance. When introducing new rewriting rules, the following assumption must be obeyed:

ASSUMPTION 1. The semantics of rewriting rules does not rely on the costs of the rules. ❑

Secondly, we assumed that the arguments of a target function call are evaluated in right-to-left order. However, if the arguments consist of more than one identifier-reference (E.g. we can define a polymorphic function call

`sumProp(`"p_1"`,`v_1`,...,`"p_n"`,`v_n`)`

to represent a source expression $p_1@v_1.+ \ldots + p_n@v_n$, for any $n > 0$.), then the evaluation order among these arguments will rely on the function implementation. For instance, in the function `sumProp` the values of p_1 to p_n must be evaluated in right-to-left order. Nevertheless, this requirement somehow can lead to an implementation in a zigzag way. E.g. in the body of function `sumProp`, the arguments should be counted in left-to-right order, because a variable number of arguments is allowed. Thus, it is necessary to introduce a stack in order to push all arguments, and latter to pop them in the reverse order for the purpose of dereferencing these identifier-reference arguments. Clearly this way more space would be required compared to the more direct way of dereferencing these arguments as soon as they are encountered. Fortunately, the correctness of theorem 1 and 2 does not rely on this restriction.

ASSUMPTION 2. The bodies of target functions do not rely on the order in which their identifier-reference arguments are evaluated. ❑

Third, as mentioned above, pattern matching only needs to control the evaluation order either between methods and methods or between methods and properties. If a rewriting rule specifies more than one method call or a mix-

ture of method calls and properties, then pattern matching would be powerless to control the dereferencing order among these method-calls and/or properties, since the control is already transferred to the body of target functions. As a consequence of assumption 2, this case would not be under control. Therefore, there is a necessity to restrict the patterns to separate method calls from other kinds of nodes.

ASSUMPTION 3. The set of rewriting rules must be consistent, such that only rules R_8 and R_9 contain nodes for method calls. ❑

6. Comparison with previous work

Previous research in pattern-directed code generation can be broadly classified into two categories: LR parsing and tree-pattern matching. With the former one, the code generator is constructed as a syntax-directed translator in which a linearized prefix form of the IR tree is parsed by an LR parser built from a context-free grammar that describes the target machine [7, 8, 10-12]. With the later one, the approach employed direct tree-pattern matching techniques [1-3, 6, 13]. But all of them are focusing on source-to-machine-code translation.

On the other hand, the current approach for source-to-source code generation is still the traditional case analysis. This has been very useful for many applications. In general, the algorithm for case analysis is distributively defined at each node in the tree, and refers to the synthesized and inherited attributes. In the source-to-source background, one target instruction may cover a variable depth of the tree while there may be different alternatives to be chosen for the same tree. It is necessary for case analysis to consider the different combinations from the current node to its children and/or parent, and even to deeper levels. By far the largest problem of this approach is that the definitions of the target instructions are difficult to modify, since the algorithm for case analysis must follow this modification also.

Benefited from the theoretical and practical work that has been done on the conventional dynamic programming, this paper extends the tree pattern matching into source-to-source code generation. However, there are several differences. The main point is that the pattern matching algorithm must explicitly control the evaluation order. In our source-to-source background, instructions are function calls in which properties are passed as identifier-reference arguments. As a result, the dereferencing order may violate the semantic. In conventional approaches, however, the issue of evaluation order is not resolved in the phase of pattern matching. It is assumed that the evaluation order is correct provided a traversal strategy for the syntax tree specifies a legal order [14], since the syntax tree must be rewritten into machine-level (intermediate representation) before applying the pattern matching. Consequently, the rewritten tree already uniquely determines the evaluation order. Suppose, for example, the syntax tree (4) is translated into machine code. Before applying pattern matching, the tree must be rewritten into a form of function calls so that choosing the machine code corresponding to code (5) or (6), which specify different evaluation orders, is actually already determined in this phase, not in pattern matching.

7. Discussion & Conclusion

This paper extends conventional dynamic programming into source-to-source code generation. Based upon this approach, the procedure for optimization shows good openness. In the conventional background, target instructions are predefined by hardware and are fixed. In contrast, in our application the target instructions are a series of function calls and can be developed incrementally. Generally speaking, initially primitive instructions can be defined in the target-library. Latter, special composite instructions can be added to improve the efficiency, especially frequently occurring groups of primitive instructions can be defined as composite instructions. Although from the point of view of semantic completeness, the composite instructions are redundant, but they play an important role in the optimization, since more direct representations lead to a higher performance.

To facilitate an incremental development based on the improvements of the target-library, a modular design is needed. The dynamic programming approach aims to achieve this goal. Improvements of the target-library require only modifications of the rewriting rules since those are separated from the pattern matching algorithm.

Acknowledgements

We thank Wolfgang Klas and Peter Muth, for reading a preliminary version of this paper and suggesting improvements for its presentation.

References

1. AHO, A. V., AND CORASICK, M. J. Efficient string matching: An aid to bibliographic search. *Comm. ACM 18,6* (June 1975), pp. 333-340.

2. AHO, A. V., AND GANAPATHI, M. Efficient string matching: An aid to code generation. In *Proc. of the ACM Symp. on Principles of Programming Languages.* ACM, New York, 1985, pp. 334-340.
3. AHO, A. V., GANAPATHI, M., AND TJIANG, S. W. K. Code Generation Using Tree Matching and Dynamic Programming. *ACM Trans. Program. Lang. Syst. 11. 4* (Oct. 1989). pp. 491-516.
4. AGRAWAL, R., DAR, S., AND GEHANI, N. The O++ database Programming Language: Implementation and Experience. Proc. on *Data Engerienging* 1993.
5. CHEN, W.-M., AND TURAU, V. VML Code Generatin Based on Tree-Pattern Matching and Dynamic Programming. Tech. Rep. 755, 1992, GMD-IPSI.
6. FRASER, C. W. Automatic Generation of code generators. Ph.D. Dissertation, Yale University, New Haven, Conn., 1977.
7. GLANVILLE, R. S. A Machine Independent Algorithm for Code Generation and Its Use In Retargetable Compilers. Ph.D. Dissertation, University of California, Berkeley, Dec. 1977.
8. GLANVILLE, R. S., AND GRAHAM, S. L. A new method for compiler code generation. In *Proc. ACM Symp. on Principles of Programming Language.* ACM, New York, 1978, pp. 231-240.
9. GORLEN, K., ORLOW, S., AND PLEXICO, P. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley & Sons, 1990.
10. HENRY, R. R. Graham-Glanville Code Generators, Ph.D Dissertation, Computer Science Division, Electrical Engineering and Computer Science, Univ. of California, Berkeley, 1984
11. HENRY, R. R., AND DAMRON, P. C. Performance of Table-Driven Code Generators Using Tree-Pattern Matching. Tech. Rep. 89-02-02, Dept. of Computer Science, University of Washington, Seattle. Feb. 1989.
12. HENRY, R. R. Encoding Optimal Pattern Selection in a Table-Driven Bottom-Up Tree-Pattern Matcher. Tech. Rep. 89-03-04, Dept. of Computer Science, University of Washington, Seattle. Feb. 1989.
13. HOFFMAN, C. W., AND O'DONNELL, M. J. Pattern Matching in Trees. *J. ACM 29, 1* (1982), pp. 68-95.
14. LANDWEHR, R., JANSOHN, H. S., AND GOOS, G. Experience with an automatic Code Generator Generator. *ACM SIGPLAN Notices* 23, 7 (1982), pp. 56-66.
15. RICHARDSON. J. E., AND CAREY. M. J. Persistence in the E Language: Issues and Implementation. *Software — Practice & Experience* 19, 12(Dec. 1989), 1115-1150.
16. TURAU, V., AND CHEN, W.-M. VML Compiler: Issues and Implementation. (submitted for publication)