Rapid and Robust Compiler Construction Using Template-Based Metacompilation

C. van Reeuwijk

Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
C.vanReeuwijk@cs.tudelft.nl

Abstract. We have developed Tm, a template-based metacompiler. Given a set of data-structure definitions and a template, Tm generates files that instantiate the template for the given data structures. With this process, Tm is able to generate program code to manipulate these data structures. Since it uses templates, the generated code is not restricted to a specific programming language: any sufficiently powerful programming language can be targeted.
Tm has been used for a wide variety of tasks and languages. However, it was designed to support compiler construction, and most applications have been in that area.
In this paper we outline Tm, and describe our experiences with using it to construct a static compiler for Java. As we will show, it has significantly accelerated implementation of the compiler. Almost 75% of its source code is generated by Tm, allowing us to rapidly implement a much more robust and sophisticated compiler than would have been possible otherwise.

1 Introduction

In an earlier paper [6] we described Tm (short for Template Manager), a template code generator. Given a set of data-structure definitions and a template, Tm generates an output file that is an expansion of the template using the data structure definitions (Fig. 1).

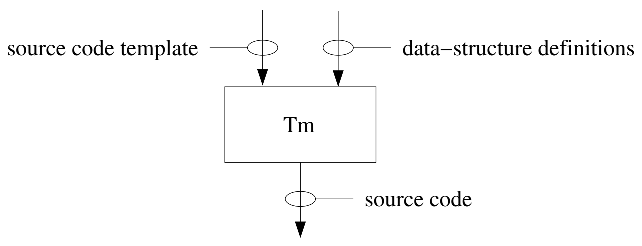


Fig. 1. Given a source code template file and a set of data-structure definitions, Tm generates a source code file.

For example, the following definitions could be used to represent connections between electronic components:

```
connection = Wire: { name:string } | Bundle: { l:[connection] };
```

A connection is either a single wire or a bundle of connections, represented by types `Wire` or `Bundle` respectively. Both are subtypes of `connection`. A `Wire` contains a `string` field, a `Bundle` contains a list of `connections`.

Now consider the following Tm template:

```
.foreach t ${typelist}
typedef str_$t *$t;
.endforeach
```

The two lines starting with a dot form a Tm command that iterates over the defined types, and assigns the current type to variable `t`. The remaining line is written to the output in each iteration. The two `$t` expressions are references to variable `t` that are substituted by Tm.

Executing this template using the type definitions shown above results in:

```
typedef str_Wire *Wire;
typedef str_Bundle *Bundle;
typedef str_connection *connection;
```

Tm templates are programs for the Tm macro language. When executed, these programs can generate source code for another programming language. This process is called *metaprogramming*, since it is a metalevel above 'normal' programming. The approach that Tm uses is called *static metacompilation* or *template metaprogramming* since it is done at compile time, not at runtime.

Because it uses templates, Tm is neutral with respect to the target language of the generated files. Various users have written templates for programming languages such as Miranda, Pascal, C, C++, Lisp, Clean and Java, but also for targets such as Unix shells, the Unix streaming editor (`sed`), and configuration files for various programs. The most common target is the C programming language.

Tm supports file inclusion in its templates, so code can be shared between projects, and standard templates can be provided for common code. An extensive set of standard templates have been developed for C, and for many programs the code provided by these templates is sufficient.

Tm has proved to be very useful in a large variety of projects. To illustrate this, we will examine the use of Tm and its C templates in the construction of Timber [9,10], a parallelizing Spar/Java compiler. Using Tm has had a profound impact on the implementation of Timber. Nearly 75% of its source code is generated by Tm. Code templates strongly encourage code reuse, since a code template section is repeatedly expanded, and entire code templates are re-used between projects. Moreover, code templates can automate a number of error-prone tasks, such as dependency calculations between node types. For all these reasons, using Tm has allowed us to implement a far more robust, powerful and adaptable compiler than otherwise would have been possible.

The source code of the Timber compiler is available for downloading from [4], the source code of Tm itself is available for downloading from the Tm website [5].

The paper is organized as follows: In Section 2 we describe related work. In Sections 3 and 4 we give an overview of Tm. In Section 5 we describe the C templates of Tm. In

Section 6 we describe the Timber compiler and the impact of `Tm` on its implementation, and in Section 7 we draw some conclusions.

2 Metaprogramming Languages

In a sense, the most popular metaprogramming language is formed by the directives of the C/C++ preprocessor. Unfortunately, it is a very weak language lacking even simple features such as iteration or string manipulation. Generic macro processors such as the Unix tool '`m4`' have also been used for metaprogramming, but such a macro processor has no knowledge about the data-structure definitions for which code must be generated. However, for effective metaprogramming the metaprogramming language must know the data-structure definitions of a program, and the relation between them. This allows the metaprogram to generate code that is tailor-made for a specific data structure.

Knowledge about the data-structure definitions can be provided at run time or at compile time. At run time, the knowledge can be provided through a set of inquiry functions that list the types in a program, list members of a type, etc. Such inquiry functions are available in languages like Java, Smalltalk, and Python. This approach is called *dynamic metaprogrammimg*. For example, Java associates a `java.lang.Class` object with every object in a program. Query methods of the `Class` object allow the program to list methods, constructors and fields of the class, and to obtain detailed information about these methods.

The advantage of dynamic metaprogramming is that the same language is used for programming and metaprogramming, obviating the need to learn a new language. However, since the metaprogramming is done at run time, is difficult to compile the output of the metaprogram. The alternative, interpretation, results in slower execution. Also, dynamic metaprogramming languages are usually not designed for large-scale metaprogramming, so that extensive templates are cumbersome to implement. Finally, dynamic metaprogramming is inherently restricted to a single programming language.

When the knowledge about the data-structure definitions is provided at compile time, this is called *static metaprogramming* or *static metacompilation*. Knowledge about the data-structure definitions can be extracted from the target language, or can be provided as definitions in the metaprogramming language. The first approach requires tight integration with the programming language under it. It is used, for example, in Willink's Flexible Object Generator [11,12] (FOG). He replaces the standard preprocessor of C++ with a much more powerful metacompiler integrated with C++. Unfortunately, although FOG can access C++ class definitions, it does not allow computations on the relations between classes. This is a restriction of FOG, not one that is inherent to the used approach. The approach *is* inherently restricted to a single programming language.

It is also possible to construct a static metacompiler that is independent of the underlying programming language. With this approach, the data-structure definitions are part of the metalanguage, and metaprograms must generate data-structure definitions for the target language. This makes the metacompiler fully independent of the target language. This is the approach used by `Tm`, although we strictly separate the macrolanguage and the data-structure definition language.

The same approach is used by AutoGen [1]. It shares many features with Tm, but was designed for general code construction tasks. In contrast, Tm was designed to generate manipulation code for data structures, and in particular to assist in compiler construction. AutoGen's macro language does not have Tm's rich set of functions and commands to access and manipulate data-structure definitions. Also, it lacks the rich set of C templates that Tm provides.

3 Tm Data-Structure Definitions

A Tm data-structure definition file, such as the one shown in Fig. 2, consists of a series of definitions of Tm *types*. A Tm type is either a *class*, or a *tuple*. For example, in Fig. 2 `origin` is a tuple type, all others are class types. Both classes and tuples can contain an arbitrary number of fields.

```
expr = { org:origin } +
    VarExpr: { nm:string } |
    AddExpr: { l:expr, r:expr } |
    SubExpr: { l:expr, r:expr } |
    NegExpr: { x:expr } |
    ConstExpr: { n:int } |
    CallExpr: { fn:string, parms:[expr] };

origin == ( file:string, line:int );
```

Fig. 2. A typical set of Tm type definitions to represent expressions in a programming language.

3.1 Fields

Each field of a tuple or class consists of a *name* and a *type*. The type can be either a simple type, written as the name of the type; or a list type, written by surrounding a type with a square bracket pair ('`[`' and '`]`'). List types denote lists of arbitrary length, whose length can change at run-time. For example, the following are all valid fields:

```
line:int          file:string
points:[point]    words:[[char]]
```

3.2 Class Types

In its simplest form, a class type consists of a list of fields separated by commas, and surrounded by curly braces. Like all type definitions, it must be terminated by a semicolon ('`;`'). For example:

```
origin = { file:string, line:int };
```

A class can also inherit from other types. For example:

```
ifStatement = statement + { cond:expr, then:block, else:block };
```

means that the `ifStatement` class inherits the fields of the `statement` class.

A class can be defined to be *virtual* by using the '`~=`' operator instead of the '`=`' operator. This indicates that the class itself will never be created, only subclasses of this class. For example:

```
statement ~= { org:origin };
```

To allow compact and clear specification of a class with many subclasses, subclasses can be specified in the class itself. For example:

```
statement = { org:origin } +
    ifStatement: { cond:expr, then:block, else:block } |
    whileStatement: { cond:expr, body:block } |
    forStatement: { var:string, bound:expr, body:block } |
    assignStatement: { lhs:expr, rhs:expr }
    ;
```

Every labeled component is called an *alternative*; every alternative defines a subclass with the name of its label. A class containing alternatives is always virtual. Thus, the definition above is equivalent with:

```
statement ~= { org:origin };
ifStatement = statement + { cond:expr, then:block, else:block };
whileStatement = statement + { cond:expr, body:block };
forStatement = statement + { var:string, bound:expr, body:block };
assignStatement = statement + { lhs:expr, rhs:expr };
```

3.3 Tuple Types

A *tuple* consists of a list of fields separated by commas and surrounded with parentheses. Like all type definitions, it must be terminated by a semicolon (‘`;`’). For example:

```
origin == ( file:string, line:int );
```

The ‘`==`’ operator introduces a tuple type.

A tuple can inherit from other types. For example, the following tuple inherits from `statement`:

```
ifStatement == statement + ( cond:expr, then:block, else:block );
```

A tuple statement cannot contain alternatives or multiple lists of fields.

A tuple type can always be converted to an equivalent class type; tuples are provided for compactness and efficiency.

3.4 Restrictions

A number of restrictions are enforced on the type definitions:

- A type can not have the same name as a previously defined type.
- A type can not, directly or indirectly, inherit from itself.
- A type can not, directly or indirectly, inherit the same type twice.
- A type can not have two fields with the same name, or inherit a field with the same name as one of its own fields.

4 The Tm Template Language

The Tm template language is an untyped interpreted programming language to manipulate Tm type definitions and text. It is powerful enough to generate code for arbitrary programming languages, and for metalevel computations such as generating sequence numbers, calculating the dependencies between types, and calculate the transitive closure of these dependencies.

In Tm templates, all lines starting with a dot ('.') are commands. Lines that do not start with a dot are copied to the output. In both command lines and output lines, expressions starting with a $ are expanded. Expressions of the form `$()` denote variable references, expressions of the form `$[]` denote arithmetic expressions, expressions of the form `${}` denote function invocations, and all other expressions of the form `$<letter>` denote variable references to the variable `<letter>`. For example, the template:

```
.set n 4
.set words for while goto
int br[$n,${len $(words)}];
int ht[$[$n*${len $(words)]}]];
```

will produce:

```
int br[4,3];
int ht[12];
```

The function `len` calculates the length of the list it is given, in this case the list assigned to variable `words`. The `$[]` expression in the declaration of `ht` multiplies the calculated length by `n`.

There are also functions to list the defined types, list the field names of a given type, retrieve the type of a given field, manipulate strings, etc. There are also commands to include files, define macros, etc. For further details see [7].

5 The Tm C Templates

As part of the core Tm distribution a number of templates for the C programming language are provided. These templates have been used in a large range of programs, including Tm itself and in the Timber compiler described below. It is useful to distinguish three different types of template: *administration* templates, which generate code for general-purpose administration of types, *tree walker* templates, that generate code to visit particular nodes in a tree, and *analysis* templates, that generate code to traverse a tree and collect information about the nodes in the tree.

For example, using the type definitions of Fig. 2 consider the following template:

```
.set wantdefs rdup_origin
.set basename demo
.include tmc.ct
```

The variable `wantdefs` is set to the list of functions that should be generated. In this case only the function `rdup_origin` is requested. The last line includes the standard administration template file `tmc.ct`. The code in this file will generate the requested function.

From this template, Tm will generate a function rdup_origin that creates a duplicate of an origin instance. The C templates automatically generate other functions when they are necessary to implement the requested functions. In this case, the template will also generate a function new_origin that, given a string and an integer, creates a new instance of origin.

5.1 Administration

The C administration code templates can generate code to:

- Create and destroy instances of the defined types.
- Read and write an ASCII representation of instances of these types.
- Compare two instances.
- Manipulate lists: append to, insert in, delete from, reverse, concatenate.
- Duplicate type instances.

For example, to create new instances of the types of Fig. 2, the following functions can be generated:

```
origin new_origin( int line, string file );
expr new_VarExpr( origin org, string nm );
expr new_AddExpr( origin org, expr l, expr r );
expr new_CallExpr( origin org, string fn, expr_list parms );
expr_list new_expr_list();
```

To recursively free instances of these types, the following functions can be generated:

```
void rfre_origin( origin e );
void rfre_expr( expr e );
void rfre_expr_list( expr_list l );
```

As explained above, the C templates automatically generate other functions if they are necessary to implement the requested functions.

5.2 Tree Walkers

It is often necessary to traverse (‘walk’) a tree, and visit all nodes of a specific type. For example, in the types of Fig. 2 we might want to visit all NegExpr nodes containing a ConstExpr, and replace them with a new ConstExpr. The action to be performed on each node must be written by the user. However, code is also needed to traverse the tree and ensure that all instances of the target nodes are visited, and Tm can take care of that. Appendix A shows a tree walker to implement our example, here we only briefly describe its requirements and features.

The tree walker template requires the following from the programmer:

- A list of node types to start the walk from, and a list of node types to visit.
- Action functions for all node types that must be visited.
- Macros for generating signatures and invocations of the walker functions.

From this information Tm computes the set of nodes to walk, and generates appropriate walker functions. The action functions provided by the user are copied to the output file, and together they form a complete tree walker.

By letting the user specify the signature of the walker and action functions, the tree walkers are flexible enough to pass arbitrary information into the tree walk, and to accumulate arbitrary information during the tree walk.

Using a tree walker has the usual advantages of code templates: extensive code re-use. Moreover, the tree-walker template automates the calculation of the required traversal. Since that is an error-prone task that must be repeated after every change or addition to the data structures, automation greatly improves the reliability of the traversal code.

A tree walker is similar in concept to the *visitor pattern* that has been proposed as a design pattern for object-oriented programming [3]. In both cases we wish to apply operations on a set of node types in a tree. The visitor pattern is implemented by adding a method to all node types. These methods implement a walk over the entire tree. During the walk, nodes are passed to a visitor method that applies the appropriate method for that type of node. A different type of walk over the tree only requires the definition of a different visitor method.

Although the visitor pattern has some of the advantages of a Tm template, it also has a number of drawbacks. In particular, it is still necessary to implement the (generic) tree walk by hand. Moreover, the visitor methods are often complicated since the correct action for every type of node must be determined and executed. Finally, the entire tree is always visited, even if a particular walk does not require it.

In contrast, for a Tm tree walker all tree traversal and type inspection code is generated; the user only needs to supply the code for the operations on the visited types.

5.3 Analyzers

One specific type of tree walker is used to collect information about a tree. For example, we might want to estimate the size of the generated code, determine whether an expression has side-effects, or collect the variables that are used in a code fragment. We call such tree walkers *analyzers*, and we provide a specialized template to generate them. An analyzer must not modify the tree it walks, and its operation must be a *reduction* operation. Typical reduction operators are boolean *and* and *or*, summation (for example to calculate the estimated size of a code fragment), and list concatenation (for example to collect all variable names in a code fragment).

The analyzer template requires the following from the programmer:

- A list of node types to start the walk from, and a list of node types to visit.
- For all the node types to visit, a classification of the node. The method can be `ignore` (do not visit this node), `reduction` (the value is the reduction of the values of its fields, possibly combined with a given constant), `constant` (the value is the given constant), or `function` (the value is computed by a user-supplied function).
- The type of the analysis result (e.g. `int`).
- The reduction operator to apply (e.g. addition).
- The neutral element of the reduction (e.g. 0).
- A macro to generate walker function signatures.
- Optionally, a termination test expression.

From this information the set of nodes to walk is computed, and appropriate walker functions are generated. The termination test expression allows useless tree walks to be cut off. For example, once the intermediate result of a boolean *and* reduction is *false*, the traversal can stop, since the result will always be *false*.

6 Application of Tm in the Timber Compiler

Tm and its C templates are used extensively in the Timber compiler [9,10], a static compiler for a superset of Java [4]. To illustrate the usefulness of Tm we will describe the impact that the use of Tm has had on the compiler.

Internally, the Timber compiler consists of three modules (Fig. 3): a frontend that translates Spar/Java to an intermediate representation called Vnus [2,8], a number of parallelization engines that rewrite Vnus, and a backend that translates Vnus to C++ code.

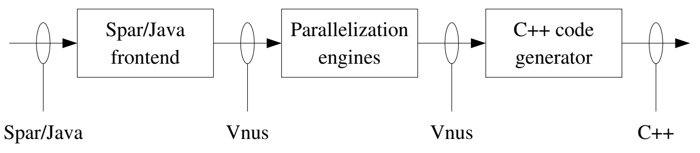
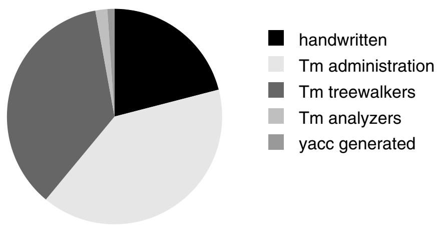


Fig. 3. Data flow in the Timber compiler.

To give an indication of the amount of work Tm has saved us, we will show statistics comparing the number of lines of hand-written and generated code[1]. We calculate the amount of generated code by counting the lines in the generated source files, and subtracting the number of lines in the template file. For the amount of hand-written code we count the lines in the non-generated source files, and in the template files.



	lines	%
handwritten	119,555	26.3%
Tm administration	175,744	38.6%
Tm tree walkers	144,643	31.8%
Tm analyzers	10,698	2.4%
yacc	4,533	1.0%

Fig. 4. Code origin for the entire Timber compiler.

[1] This comparison is meaningful because the style of the code generated by our Tm templates is similar to what we write ourselves.

We assign each line of code that is passed to the C compiler to one of the following five categories: hand-written, generated by yacc, or generated by a Tm administration, tree walker or analysis template.

Figure 4 shows the statistics for the entire compiler. In subsequent sections we will show the statistics for the individual compiler phases.

As these figures show, nearly 75% of the compiler code is generated by Tm. Roughly half of the generated code is for administration, and the other half implements tree walkers. Only a small fraction of the generated code is devoted to analyzer tree walkers. One reason for this is that analyzer tree walkers are a fairly recent development; some analysis operations are still done in hand-written code, even though in a new implementation an analyzer tree walker would be used.

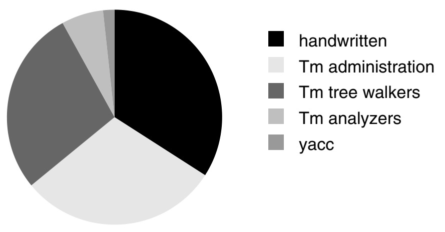
The Timber compiler has taken an estimated five person-years to implement: three person-years to implement a static Java compiler, and two to implement the language extensions and the parallelization engines. The resulting compiler is able to compile large programs and large parts of the standard Java library to efficient executables.

6.1 Communication between Engines

The Timber compiler consists of independent programs, called *engines*, that are 'glued' together with a shell script. Internally, each engine represents the program as a tree of Tm types. Communication between the compiler engines is implemented using Tm-generated functions. These functions print a tree to a textual representation in a file, and convert this textual representation back into a tree.

6.2 The Spar/Java Frontend

Figure 5 shows the code generation statistics for the Spar/Java frontend.



	lines	%
handwritten	50,718	34.1%
Tm administration	44,640	30.0%
Tm tree walkers	41,557	27.9%
Tm analyzers	9,372	6.3%
yacc	2,554	1.7%

Fig. 5. Origin of the frontend code.

The frontend parses Spar/Java, applies the semantic checks required by Java on the program, applies a number of optimizations, and generates Vnus. A number of tree walkers implement distinct compiler phases. In order of their application they do the following:

- Rewrite some constructs to simplify the remaining phases.
- Register class declarations in the symbol table.
- Register methods and constructors in the symbol table.
- Bind variables, types, methods and constructors.
- Check correctness of the program.
- Apply a number of code optimizations, e.g. inlining, constant folding.
- Add garbage-collection administration code.
- Eliminate unused variable declarations.

Other tree walkers implement auxiliary operations that work on fragments of code instead of an entire program. They do the following:

- Mark variables that are only read as ‘final’.
- Rename variable references (used in method inlining).
- List the scope names of a code fragment.
- List variables that are not bound in the given code fragment.
- Do constant folding on an expression.
- List the assigned variables of a code fragment.
- Update the use count of the methods used in a code fragment.
- Rewrite ‘return’ statements to ‘goto’ statements (used in method inlining).

A number of analyzer tree walkers are also used, which do the following:

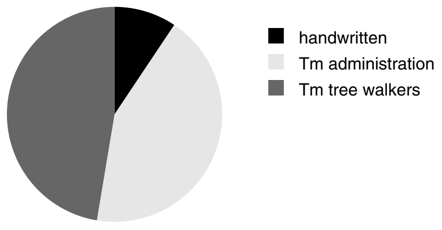
- Estimate the size of a given code fragment.
- Determine whether an expression is constant.
- Determine whether an expression requires the garbage-collection administration to be up-to-date.
- Determine whether a code fragment alters the state of the garbage-collection administration.
- Determine whether an expression has side effects.
- Determine whether an expression evaluates to zero.

6.3 The Parallelization Engines

Figure 6 shows the code generation statistics for the parallelization engines.

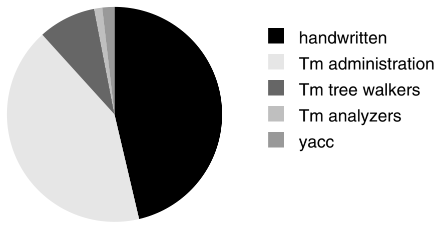
The parallelization engines transform implicitly parallel Vnus programs (sequential programs with parallelization annotations) to explicitly parallel Vnus programs. The engines are implemented as a set of 57 rules that each apply a simple rewrite operation on the Vnus program. These rules are implemented as tree walkers. Some example rules are:

- Search for loops that only contain a communication statement for a single element, and replace them by code that communicates all elements in a single message.
- Exchange loops in a loop nest when this is profitable.
- Simplify `if` statements with a constant `true` or `false` condition.



	lines	%
handwritten	18,534	9.4%
Tm administration	85,536	43.3%
Tm tree walkers	93,680	47.4%

Fig. 6. Origin of the parallelization code.



	lines	%
handwritten	50,303	46.3%
Tm administration	45,568	42.0%
Tm tree walkers	9,406	8.7%
Tm analyzers	1,326	1.2%
yacc	1,979	1.8%

Fig. 7. Origin of the backend code.

6.4 The Vnus Backend

Figure 7 shows the code generation statistics for the Vnus backend.

The backend translates Vnus code to C++ code. Similar to the frontend, a number of tree walkers implement distinct compilation phases (checking, optimization), and a number of other tree walkers serve as auxiliary functions (constant folding, tests, etc.).

7 Conclusions

Our template-based metacompiler Tm is able to generate an extensive range of functions to manipulate data structures. Since it uses templates, the generated code is not restricted to a specific programming language.

Since Tm provides a full programming language for template implementation, it is possible to write highly sophisticated templates, for example the tree walker templates described in Section 5.2, and the analyzers described in Section 5.3.

As we have shown, the use of Tm has had a profound impact on the implementation of Timber, our Spar/Java compiler. Nearly 75% of the source code of the compiler is generated by Tm, allowing rapid implementation of the compiler, and resulting in a much more robust and sophisticated compiler than would have been possible otherwise. Consequently, in three person-years we have been able to implement a Java compiler that is able to correctly compile large parts of the standard library to efficient executables. Our extensions to Java, and the parallelization engines were implemented in two person-years.

References

[1] AutoGen website. URL: autogen.sourceforge.net.

[2] P.F.G. Dechering, J.A. Trescher, J.P.M. de Vreught, and H.J. Sips. V-cal: a calculus for the compilation of data parallel languages. In C.-H. Huang et. al., editor, *8th Intl. Workshop, Languages and Compilers for Parallel Computing*, number 1033 in LNCS, pages 388–395, Columbus, Ohio, USA, August 1995. Springer Verlag.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, January 1995.

[4] C. van Reeuwijk. Timber download page. www.pds.twi.tudelft.nl/timber/downloading.html.

[5] C. van Reeuwijk. Tm website. www.pds.twi.tudelft.nl/∼reeuwijk/software/Tm.

[6] C. van Reeuwijk. Tm: a code generator for recursive data structures. *Software – Practice and Experience*, 22(10):899–908, October 1992.

[7] C. van Reeuwijk. Template manager reference manual. PDS Technical Report PDS-2000-003, Delft University of Technology, May 2000. www.pds.twi.tudelft.nl/reports/2000/PDS-2000-003.

[8] C. van Reeuwijk. The Vnus language specification, version 2.1. PDS Technical Report PDS-2000-002, Delft University of Technology, May 2000. www.pds.twi.tudelft.nl/reports/2000/PDS-2000-002.

[9] C. van Reeuwijk, A. van Gemund, and H.J. Sips. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency – Practice and Experience*, 11(9):1193–1205, November 1997.

[10] C. van Reeuwijk, F. Kuijlman, and H.J. Sips. Spar: a set of extensions of Java for scientific computation. *Concurrency and Computation: Practice and Experience*, accepted for publication.

[11] Edward D. Willink. *Meta-Compilation for C++*. PhD thesis, Computer Science Research Group, University of Surrey, UK, June 2001.

[12] Edward D. Willink and Vyacheslav B. Muchnick. An object-oriented preprocessor fit for C++. *IEE Proceedings – Software*, 147:49–58, April 2000.

A Example Tree Walker

Given the data structures of Fig. 2, the following tree walker template generates code to rewrite all `NegExpr` instances containing a `ConstExpr` to a new constant expression. To do this, an action function `fold_NegExpr_action` is defined. See the comments in the template and the generated code for further explanation.

```
.macro generate_walker_declaration v t
static $t fold_$t_walker( $t $v );
.endmacro
.macro generate_walker_signature v t
static $t fold_$t_walker( $t $v )
.endmacro
.. Given an indent, an expression, its real type and its
.. perceived type, generate invocation of an action.
.macro generate_action_call i x t n
.if ${eq $t $n}
$i$v = ($t) fold_$t_action( $x );
.else
$i$v = ($t) fold_$t_action( ($t) $x );
.endif
.endmacro
.. Given an indent, an expression, its real type and its
.. perceived type, generate invocation of a walker.
.macro generate_walker_call i v t n
.if ${eq $t $n}
$i$v = ($t) fold_$t_walker( $v );
.else
$i$v = ($t) fold_$t_walker( ($t) $v );
.endif
.endmacro
.. If 't' has an action, invoke it, else invoke its walker
.macro generate_descent_call i v t n
.if ${member $t $(actors)}
.call generate_action_call "$i" "$v" "$t" "$n"
.else
.call generate_walker_call "$i" "$v" "$t" "$n"
.endif
.endmacro
.set actors NegExpr
.. Insert the macros required for tree walking.
.insert tmcwalk.t
.. Calculate which types must be visited.
.set visit_types ${call calc_treewalk "expr" "$(actors)"}
.. Generated forward declarations for the walker functions
.call generate_walker_forwards "$(visit_types)"

static expr fold_NegExpr_action( NegExpr x )
{
.call generate_walker_call "    " x NegExpr NegExpr
    if( x->x->tag == TAGConstExpr ){
        ConstExpr res = (ConstExpr) (x->x);
        x->x = exprNIL;
        rfre_expr( x );
        res->n = -res->n;
        return (expr) res;
    }
    return x;
}

.. Generate the walker functions.
.call generate_walker "$(visit_types)"
```

When this template is executed, the following code is generated:

```
/* ----------- Generated forward declarations start here ----------- */

/* Forward declarations. */
static AddExpr fold_AddExpr_walker( AddExpr e );
static SubExpr fold_SubExpr_walker( SubExpr e );
static NegExpr fold_NegExpr_walker( NegExpr e );
static expr_list fold_expr_list_walker( expr_list e );
static CallExpr fold_CallExpr_walker( CallExpr e );
static expr fold_expr_walker( expr e );

/* ----------- Generated forward declarations end here ----------- */

static expr fold_NegExpr_action( NegExpr x )
{
    x = (NegExpr) fold_NegExpr_walker( x );
    if( x->x->tag == TAGConstExpr ){
        ConstExpr res = (ConstExpr) (x->x);
        x->x = exprNIL;
        rfre_expr( x );
        res->n = -res->n;
        return (expr) res;
    }
    return x;
}

/* ----------- Generated code starts here ----------- */
/* Walker for class AddExpr. */
static AddExpr fold_AddExpr_walker( AddExpr e )
{
    e->l = (expr) fold_expr_walker( e->l );
    e->r = (expr) fold_expr_walker( e->r );
}

/* Walker for class SubExpr. */
static SubExpr fold_SubExpr_walker( SubExpr e )
{
    e->l = (expr) fold_expr_walker( e->l );
    e->r = (expr) fold_expr_walker( e->r );
}

/* Walker for class NegExpr. */
static NegExpr fold_NegExpr_walker( NegExpr e )
{
    e->x = (expr) fold_expr_walker( e->x );
}

/* Walker for list expr_list. */
static expr_list fold_expr_list_walker( expr_list e )
{
    {
        unsigned int ix;

        for( ix=0; ix<e->sz; ix++ ){
            e->arr[ix] = (expr) fold_expr_walker( e->arr[ix] );
        }
    }
}

/* Walker for class CallExpr. */
static CallExpr fold_CallExpr_walker( CallExpr e )
{
    e->parms = (expr_list) fold_expr_list_walker( e->parms );
}

/* Walker for class expr. */
static expr fold_expr_walker( expr e )
{
    switch( e->tag ){
        case TAGAddExpr:
            e = (AddExpr) fold_AddExpr_walker( (AddExpr) e );
            break;

        case TAGSubExpr:
            e = (SubExpr) fold_SubExpr_walker( (SubExpr) e );
            break;

        case TAGNegExpr:
            e = (NegExpr) fold_NegExpr_action( (NegExpr) e );
            break;

        case TAGCallExpr:
            e = (CallExpr) fold_CallExpr_walker( (CallExpr) e );
            break;

        default:
            break;

    }
}


/* ----------- Generated code ends here ----------- */
```