Tm: a code generator for recursive data structures *

C. van Reeuwijk
Delft University of Technology
Faculty of Electrical Engineering
P.O. Box 5031
2600 GA Delft
The Netherlands

Abstract

The transfer of data structures between programs is often carried out using binary or ad-hoc textual formats. However, this can result in ambiguous and non-portable file formats. The program 'Tm' (for 'template manager') prevents these problems by using a textual representation of the data structures and generating the code to read and write this representation from an abstract definition of the data structures. The same program is used to generate code for the simple data-structure manipulations that are necessary in almost every program. At the moment, code can be generated for the programming languages C, Pascal, Lisp and Miranda. Support for other languages is easily added.

Keywords: internal representation, interfacing, code generators.

Introduction

The transfer of structured data between programs is often carried out using binary or ad-hoc textual formats. However, this can result in ambiguous and non-portable file formats. For example, the Pascal type declaration

```
record foo x,y: integer; c: char; end;
```

is 'equivalent' to the C type definition

```
typedef struct { int x, y; char c; } foo;
```

*Research for this paper was supported by the European Community as part of ESPRIT project 881 (FORFUN).

This does not imply that it is easy to transfer data in these records and structures from one language to the other. Facilities that are provided for this purpose, like `get` and `put` in Pascal and `fread` and `fwrite` in C are useless, and may even cause problems if data is transferred between different implementations of the same language.

An effective way to solve this problem is to introduce a textual representation of the data. The binary read and write routines can then be replaced by text printing and parsing routines. It is now necessary, however, to define a suitable language for this representation; if this is not done properly, it leads to inconsistency or system dependency.

The program *Tm* (for *template manager*) solves this problem by generating the text printing and parsing routines automatically. Given a description of the data structures, Tm will generate the appropriate interface code for a number of programming languages.

Tm code generation is based on *templates*: source texts for the target programming language interspersed with text substitution and repetition commands for Tm. At the moment, Tm can generate code for C, Lisp, Miranda and Pascal, but code generation for other languages is easily added by writing the appropriate template.

Data structures for Tm are defined in a special data-structure definition language inspired by Miranda[1, 2]. The language allows the definition of tuples (similar to Pascal records and C structures), lists and constructors (similar to Pascal variant records), and thereby supports a very broad class of data structures.

Using the templates and the data-structure definitions, code can be generated to read and write a textual representation of the data structures. The templates are independent of the data structures: provided that the data structures are described as tuples, constructors or lists, Tm is able to generate the appropriate code.

Although Tm was originally designed for the generation of interface code, it soon became clear that it was useful to let it generate other functions as well. Some of these functions were necessary for the interface code anyway, but others were added at the request of the users. The templates for most languages can generate code for the following functions:

1. List manipulation (append, insert, concatenate).
2. Textual representation interface (read and write).
3. Dynamic memory allocation.

Practice has shown that adding functions beyond this is difficult: the functions to be added are either rarely useful or require interpretation of the data. For example, a 'sort' function would require a comparison function, and hence interpretation of the data.

Data-structure definition

Both the syntax of the data-structure definitions and the textual representation of the data structures are derived from the *tuple* and *algebraic* types of the functional programming language Miranda[1, 2]. In fact, it is often possible to use the textual representation as Miranda source text.

The types that can be defined in the data-structure definition file are *tuple* types and a *constructor* types.

A tuple is a group of elements of fixed length and order. Such a tuple type is similar to the records of Pascal, the structures of C and the tuples used in data-base systems. In Tm, a tuple type definition consists of a list of element types; each element is also given a name. For example,

```
foo == ( x:int, y:int, c:char );
```

defines a tuple type with element types `int`, `int` and `char`. The element names are `x`, `y` and `c`.

A constructor type consists of a number of alternative element groups. Constructor types are convenient for the representation of recursive structures. For example:

```
tree ::= Tree v:int l:tree r:tree | TreeNIL;
```

defines a constructor type.

The words `Tree` and `TreeNIL` are used to differentiate between possible alternatives and are called *constructors*. The character '`|`' is used to separate alternatives. By convention, constructors start with an upper case character, while type and element names start with a lower case character.

Constructor and tuple elements may be constructors and tuples, but the elements may also be of any other type. If the type name is not known to Tm, it is assumed that the type is defined elsewhere; such a type is called a *primitive type*. For example, in type `tree` type `int` is primitive.

Constructors and tuple elements can also be lists. Such lists can be of arbitrary length (including length 0). All elements must be of the same type. To indicate that a list is intended, the type name is surrounded by square brackets. For example, `[foo]` is a valid list type. Like constructor and tuple elements, list elements may be tuples, constructors or elements of primitive types. However, for implementation reasons Tm does not allow lists of lists. Thus, `[[foo]]` is *not* allowed.

Textual data representation

The type definitions described in the previous section are used as the syntax of the actual textual representation. For example, an instance of type `[foo]` is:

```
[( 3, 5, 'c' ),( -1, 42, '?' )]
```

and an instance of type `tree` is:

```
Tree 2 (Tree 6 TreeNIL TreeNIL) TreeNIL
```

Another instance of type `tree` is:

```
TreeNIL
```

The exact rules for this representation are described in the users' manual[3].

The representation can be parsed by a simple recursive descent parser. Such a parser is easily implemented in almost any programming language.

The text substitution language

Tm can generate code to read and write the textual representation described in the previous section for a number of programming languages. This is done by providing *templates* for the various languages that are filled in using the given data-structure definitions. Templates are source texts for the target programming language interspersed with text substitution and repetition commands for Tm. An example of a template is given in an appendix.

The templates for the various languages are implemented using the text substitution language described in this section. The casual user of Tm is expected to use only the standard templates that have been prepared for various programming languages, and needs only limited knowledge of the text substitution language.

In this paper, the text substitution language is only described superficially, the defining document is the users' manual[3].

Tm will copy all text from the template to the output file, with two exceptions:

1. All lines starting with a period ('.') are interpreted as commands to Tm. They are called *line commands*. Some line commands 'bracket' a number of lines. For example:

   ```
   .if 0
   True
   .else
   False
   .endif
   ```

   results in:

   ```
   False
   ```

   since Tm interprets 0 as the boolean value 'false'. There are also line commands for repetition and file inclusion.

   Another important line command is `.set`. It is used to assign to variables. For example:

```
.set blah foo bar
```

Assigns the words `foo bar` to the Tm variable `blah`.

Comment can be included in the template using two periods at the beginning of the line. For example:

```
.. This is a comment line.
```

2. Within all lines (both line commands and normal lines), the character '$' indicates the start of an expression that is evaluated by Tm. In the output the expression is replaced by the result of evaluation.

There are three forms of $ expressions:

1. Expressions of the form `${fn parm ... parm}` are function applications. The first word within the brackets is the function name, the remaining words are the parameters of the function. Tm provides functions for type definition access, arithmetic, comparison, list manipulation, version control, and various other functions. For example:

```
.set l h a c b d c
${sort $l}
${uniq $l}
${prod ${len $l} 7}
```

results in:

```
a b c c d h
a b c d h
42
```

An important class of Tm functions is that for data-structure access. These functions return information about the defined data structures. For example, `${typelist}` returns the list of defined types.

2. Expressions of the form `$[ expr ]` are numerical expressions. All the standard arithmetic operators are available. For example:

```
$[1+2+3]
```

results in:

```
6
```

3. Expressions of the form `$(varname)` are variable references. These variables are set with the line command `.set`, see above. If the variable name consists of a single character, the parentheses may be omitted. For example:

```
.set bar text
.set z substitution
Tm does $(bar) $z!
```

results in:

```
Tm does text substitution!
```

To see how this language can be used for code generation, a simplified fragment of the standard C code template is listed in an appendix.

The text substitution language is powerful enough to allow the implementation of complicated templates. For example, the templates for all supported languages require that the user supplies a list of wanted functions. From this list, the functions that are necessary to support these functions is deduced using the text substitution language.

Applications of Tm

The original purpose of Tm was to provide a well-defined interface between a parser program for the system description language Glass[4] and a number of programs that each give an interpretation of the system descriptions. Tm has proved to be very useful for this purpose, especially since it allows interpretation functions to be written in several languages. It has also been used for other, similar applications.

Tm has also been used extensively to define data structures that are only used within one program. This was done for the following reasons:

- Code that is generated by Tm is well tested. Essentially the same code (only differing in type names) is used for all data types in all generated code. Therefore, it has been exposed to a large number of runs in code that is generated for other programs, and, therefore, has been better tested than similar code written by hand for the same purpose.

- The generated functions are documented.

- Tm code assists debugging. For example, the C templates of Tm generate code to count the number of allocations and deallocations of each particular type, and the origin (source file and line) of the allocations. This has proven to be invaluable for the detection of memory leakage (memory allocation not followed by deallocation) and repeated deallocation of the same memory.

  Also, since there is a textual representation, inspection of the data is easily possible.

- Tm encourages a standard coding style. The standardization of Tm functions also forces more standardization on the functions that use the Tm functions. This enhances the clarity of the code.

These advantages have led some users of Tm not to use it to generate interface code, but simply to generate correct and debug-assisting code.

For example, one program that uses Tm consists of about 112500 lines of C code. Of these lines, 70000 were generated by Tm, the remaining 42500 were written by hand. This code was written in about four months.

Conclusion

By using Tm it is possible to define the format of data files in a flexible way. The data files that are defined in this way are easily read and written in any sufficiently powerful programming language, and Tm is able to generate code for that purpose automatically. The quality of the generated code is ensured, since it is essentially the same code that is exposed to a large number of runs for different types in many programs. This makes Tm-generated code useful even if no interfacing to data files is necessary. If a template for a programming language is not available, it may be provided by the user. This is a tedious but simple job.

References

[1] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, Englewood Cliffs, N.J., 1984.

[2] David Turner. Miranda: A non-strict functional language with polymorphic types. In J. P. Jouannaud, editor, *Functional programming languages and computer architecture*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[3] C. van Reeuwijk. Tm users' manual. Technical report, Delft University of Technology, Department of Electrical Engineering, Delft, 1992.

[4] H. Oolman, M. Seutter, and C. van Reeuwijk. Glass, a language for analog and digital circuit description, and its environment. *The EUROMICRO Journal on Microprocessing and Microprogramming*, 27(1–5):267–271, August 1989.

APPENDIX

To illustrate the use of the text substitution language of Tm, this appendix lists a simplified fragment of the standard C language template. Only the generation the structure definitions and of the tuple and list output functions is shown. The code generation will be demonstrated for the following tuples:

```
tuplea == ( na:int, nb:int, s:string );
tupleb == ( lbl:string, tl:[tuplea] );
```

In the example the following line commands are used:

`.foreach <var> <val>..<val>`

Set variable `<var>` to each of the values `<val>` and do a repeated translation of all following lines up to the next unbalanced `.endforeach` for each of these values of `<var>`.

`.if <expr>`

Evaluate `<expr>`. If `<expr>` is *false*, only translate all lines between the following unbalanced `.else` and `.endif`. If no `.else` is encountered, no translation is done.

If `<expr>` is *true*, translate the following lines up to the next unbalanced `.else` or `.endif` and if relevant skip all lines between `.else` and `.endif`.

`.set <var> <val>..<val>`

Set variable `<var>` to the given list of values.

Also, the following Tm functions are used (remember that in the template they occur as `${<function> <par> .. <par>}`):

`typelist`	Return the list of types defined in the data structure file.
`telmlist t`	Given a tuple type `t`, return the list of element names of that tuple. For a constructor type return an empty list.
`ttypeclass t e`	Given a tuple type `t` and a tuple element name `e`, return the type class of that element. Possible type classes are `single` and `list` for a single element and a list of elements respectively. If `t` is a constructor type, an error message is given.
`ttypename t e`	Given a tuple type `t` and a tuple element name `e`, return the type that element. If `t` is a constructor type, an error message is given.

The template to generate the data-structure definitions and the output functions is:

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

.foreach t ${typelist}
struct str_$t {
.foreach e ${telmlist $t}
.if ${eq list ${ttypeclass $t $e}}
```

```
    ${ttypename $t $e}_list $e;
.else
    ${ttypename $t $e} $e;
.endif
.endforeach
};

struct str_$t_list {
    unsigned int sz, room;
    $t *arr;
};

.endforeach
/* Forward declarations. */
.foreach t ${typelist}
void fprint_$t( FILE *f, $t t );
void fprint_$t_list( FILE *f, $t_list t );
.endforeach

.foreach t ${typelist}
void fprint_$t( FILE *f, $t t )
{
    putc( '(', f );
.set first 1
.foreach e ${telmlist $t}
.if $(first)
.set first 0
.else
    fputs( ",\n", f );
.endif
.if ${eq list ${ttypeclass $t $e}}
    fprint_${ttypename $t $e}_list( f, t->$e );
.else
    fprint_${ttypename $t $e}( f, t->$e );
.endif
.endforeach
    fputs( ")\n", f );
}

.endforeach
.foreach t ${typelist}
void fprint_$t_list( FILE *f, $t_list l )
{
    unsigned int ix;

    putc( '[', f );
    for( ix=0; ix<l->sz; ix++ ){
        if( ix!=0 ) fputc( ',', f );
        fprint_$t( f, l->arr[ix] );
```

```
    }
    fputs( "]\n", f );
}

.endforeach
```

The following code will be generated:

```
typedef struct str_tuplea_list *tuplea_list;
typedef struct str_tupleb_list *tupleb_list;
typedef struct str_tuplea *tuplea;
typedef struct str_tupleb *tupleb;

struct str_tuplea {
    int na;
    int nb;
    string s;
};

struct str_tuplea_list {
    unsigned int sz, room;
    tuplea *arr;
};

struct str_tupleb {
    string lbl;
    tuplea_list tl;
};

struct str_tupleb_list {
    unsigned int sz, room;
    tupleb *arr;
};

/* Forward declarations. */
void fprint_tuplea( FILE *f, tuplea t );
void fprint_tuplea_list( FILE *f, tuplea_list t );
void fprint_tupleb( FILE *f, tupleb t );
void fprint_tupleb_list( FILE *f, tupleb_list t );

void fprint_tuplea( FILE *f, tuplea t )
{
    putc( '(', f );
    fprint_int( f, t->na );
    fputs( ",\n", f );
    fprint_int( f, t->nb );
    fputs( ",\n", f );
    fprint_string( f, t->s );
    fputs( ")\n", f );
}
```

```
void fprint_tupleb( FILE *f, tupleb t )
{
    putc( '(', f );
    fprint_string( f, t->lbl );
    fputs( ",\n", f );
    fprint_tuplea_list( f, t->tl );
    fputs( ")\n", f );
}

void fprint_tuplea_list( FILE *f, tuplea_list l )
{
    unsigned int ix;

    putc( '[', f );
    for( ix=0; ix<l->sz; ix++ ){
        if( ix!=0 ) fputc( ',', f );
        fprint_tuplea( f, l->arr[ix] );
    }
    fputs( "]\n", f );
}

void fprint_tupleb_list( FILE *f, tupleb_list l )
{
    unsigned int ix;

    putc( '[', f );
    for( ix=0; ix<l->sz; ix++ ){
        if( ix!=0 ) fputc( ',', f );
        fprint_tupleb( f, l->arr[ix] );
    }
    fputs( "]\n", f );
}
```

The functions `fprintf_tuplea()` and `fprint_tupleb()` write a textual representation of a tuple to a file. The functions `fprintf_tuplea_list()` and `fprint_tupleb_list()` write a textual representation of a list of tuples to a file. It is assumed that the functions `fprint_int()` and `fprint_string` are provided elsewhere.

The example is a highly simplified version of an actual template for the following reasons:

- The real templates can generate other functions: allocation and freeing of constructors, tuples and lists, with associated administration to facilitate the detection of memory leakage; input of the textual representation; appending, inserting and deleting elements from lists; and concatenation of lists.
- The template only handles list an tuple types, not constructor types.

- All routines are generated for all types. In the real templates only those routines are generated that are requested by the user or are used by one of the other routines that are generated.
- Only ANSI C code (as opposed to 'classical' C code) is generated.
- In reality, there is a separate template to generate an `#include` file with declarations of all functions and data structures that must be visible outside the generated code file.