Program Templates:
Expression Templates Applied to Program Evaluation

Francis Maes

EPITA Research and Development Laboratory,
14-16 rue Voltaire, F-94276 Le Kremlin-Bicêtre cedex, France,
francis.maes@lrde.epita.fr,
WWW home page: http://lrde.epita.fr/

Abstract. The C++ language provides a two-layer execution model: static execution of meta-programs and dynamic execution of resulting programs. The Expression Templates technique takes advantage of this dual execution model through the construction of C++ types expressing simple arithmetic formulas. Our intent is to extend this technique to a whole programming language. The Tiger language is a small, imperative language with types, variables, arrays, records, flow control structures and nested functions. The first step is to show how to express a Tiger program as a C++ type. The second step concerns operational analysis which is done through the use of meta-programs. Finally an implementation of our Tiger evaluator is proposed.

Our technique goes much deeper than the Expression Templates one. It shows how the generative power of C++ meta-programming can be used in order to compile abstract syntax trees of a fully featured programming language.

1 Introduction

During the compilation process, an input program expressed in textual form is transformed by successive steps into executable code. As in any language, a C++ program will basically be evaluated during its execution. The interesting particularity of C++ is its ability to do some computations at compile-time using template constructions (the so-called meta-programs, see Veldhuizen (2002), Czarnecki and Eisenecker (2000), Järvi (1998) and appendix A for an example). This two-layer execution model corresponds to the usual concept of static (compile-time) and dynamic (execution-time) processing.

In C++, there is a technique called Expression Templates described by Veldhuizen (1995), which allows the exploitation of this two-layer execution model. This technique relies on transformations of simple arithmetic expressions at compile-time to increase the performances of the executable code. Moreover some evaluation can be done entirely statically with mechanisms such as constant propagation. This way, some computations usually done at execution-time are processed at compile-time.

The Expression Templates technique is based on the use of template classes. In order to work on expressions, we need a structural description of them. This is done by building a type that reflects the abstract syntax tree (AST) of the expression. Each node of this tree will be translated into a template class whose arguments are the node subtrees.

Usually, a program written in any language can also be expressed as an abstract syntax tree. The next natural step is to wonder whether it is possible to extend the Expression Templates technique to a whole programming language. Expressing a full program with a C++ type reflecting its AST could thus be made possible. In the remainder of this paper, this type will be called the TAT (Tree As Type). A TAT is a representation of an AST using a C++ type formalism.

Expressing a program in the TAT formalism would allow us to adapt the Expression Templates evaluation method to a whole program and therefore to take advantage of the two-layer execution model of C++ (see Haney and Crotinger (1999)). The entire process of compiling and executing a program expressed as a TAT corresponds to its evaluation.

To experiment this idea, we have to choose a programming language that does not have this two-layer execution model. We want this language to be simple and to have few constructions. Nevertheless, this language must at least include types, functions, records, arrays and flow control constructions. Tiger, a language defined by Appel (1997), corresponds to our needs: with only 40 rules in its EBNF grammar, it respects all our conditions.

This work is a proof of concept. No-one had previously mapped an entire language to a C++ meta-program. Those that consider C++ expression templates for prototype implementations should be interested in this project. Moreover, the C++ metalanguage is here introduced as an intermediate language. This point of view is different from the current trend of supporting meta-programming by designing metalanguages as extensions of existing programming languages. Our work initially inspired by Expression Templates goes very deeply into the possibilities of C++ meta-programs using several techniques discovered recently.

This paper begins with an overview of related work. Next, section 3 introduces the Tiger language, followed by a description of our architecture. Our first objective is to translate Tiger programs into TATs. When trying to do this, several problems arise (e.g. expressing lists). These are developed in section 4. Our second objective is to do some static processing on this TAT. This will require a structure called environment, and a form of static pointers detailed in section 5. Finally we want to evaluate a Tiger program expressed as a TAT using the C++ two-layer execution model. The implementation which allows this is described in section 6. This is followed by some interesting results related to this new technique. This paper will finish with a discussion about the possibilities of such mechanisms.

2 Related work

Our work is based on Expression Template. The Expression Template is at the basis of our work. This technique described by Veldhuizen (1995) has many known interests. In particular it allows to build the static AST of a C++ expression. This allows C++ meta-programs to work on C++ expressions seen as types. This can be useful for:

– **Rewriting statements into equivalent (but more efficient) ones**. This was the original intent of Expression Templates. This technique was first used to evaluate vector and matrix expressions in a single pass, without temporaries.
– **Building lambda terms**. Several libraries for doing functionnal programming in C++ are based on Expression Templates. Thanks to C++ meta-programs, several functionnal operations are possible on these lambda terms. The Fact library (Striegnitz and Smith (2000)) provides typical functional features such as currying, lambda expressions and lazy evaluation in C++. The Boost package also includes a library specialized in lambda expressions: the Boost Lambda Library (J. Jarvi (2002)). FC++ (McNamara and Smaragdakis (2001)) is a similar library inspired by the Haskell language. Our work has something to do with lambda term manipulations: we also manipulate TATs. But our intent is not to do functionnal operations on a TAT but to compile a whole program including functions and variables declarations.
– **Building any other structured expressions**, such as the group (2002) library which uses Expression Template in order to build EBNF rules. C++ meta-programs are then used to transform a grammar into a usable parser. In this library, C++ meta-programs deal with complex operations such as in our work.

The Expression Template is very useful but a bit complex to implement. PETE (Crotinger et al. (2000)) is a tool that aims at generating the needed code. Fact is built on top of PETE. This tool could help us to build a C++ front-end to our compiler. The idea of using template constructions in compilers has already been used for building a java compiler, see van Reeuwijk (2003).

3 Tiger evaluation and compilation

3.1 Tiger constructions

Tiger is an Algol-style language with a functional flavor. Two kinds of construction exist: declarations and typed expressions. Declarations are of three kinds: type, variable and function declarations. Four basic types exist: integers, strings, nil and void. New types can be built with records and arrays. Existing types can be renamed by a typedef mechanism. Tiger is not a first-order functionnal language: functions cannot be passed as parameters, neither as results.

Tiger has a nested `let-in-end` construction which makes it possible to declare nested scopes. A particular case of this is the ability to declare nested functions.

Except declarations, everything in Tiger is an expression: literals (strings and integers), unary and binary operations, left-values, function calls, array and record instantiations and flow control constructions: `if-then-else`, `while-do`, `for-to-do`, `break`.

3.2 Architecture

We use a front-end program which parses Tiger and does the semantic analysis: type checking, scopes and bindings. The output of this front-end is a C++ program which declares a TAT. Our front-end is based on techniques explained by Appel (1997).

The interesting thing is the remaining work: the program evaluation. This task is done in C++ through the static and the dynamic processing.

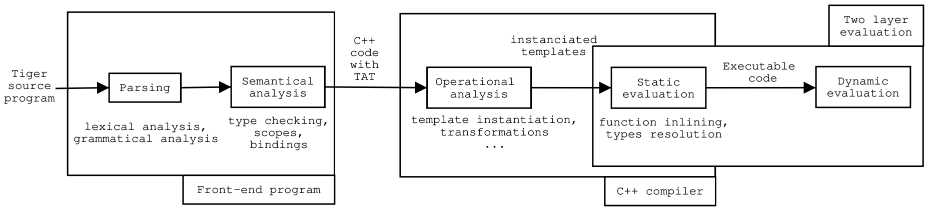

Fig. 1. Placement in the compilation chain

Our front-end associated with the C++ static processor is a compilation chain. Indeed the input of this chain is a textual Tiger program, and its output is an executable program.

3.3 Comparison with a standard compiler

A usual object oriented compiler first parses the program. It provides AST classes that are dynamically instantiated in order to build the programs abstract tree. At this point until the end of the compilation, successive transformations are applied until getting the executable code.

In our case, we provide a set of template classes corresponding to each node of the AST. During the compilation of a Tiger program, these templates are filled by our front-end giving us the TAT. At this point, the C++ compiler does successive transformations until getting the executable code.

An analogy can easily be done between our Tiger compiler and a standard compiler. Where a standard compiler provides AST classes, we provide AST meta-classes. Where a standard compiler builds an AST expressed as objects, we build an AST expressed as a type (the TAT). A standard compiler provides classes for operational analysis, we provide meta-classes to do this work.

It has been shown that a Turing machine could be constructed with template constructs (Veldhuizen (2002)). Any work traditionally done by a standard compiler can theoretically be done with C++ meta-programs. The method that we present should thus be adaptable to any other language. The only restrictions are the C++ compilation times and memory use.

4 Translation into TAT

Let us return to the Expression Templates technique with the following Tiger program:

```
(5 * 10 + 1)
```

Since the Expression Templates technique was originally used to describe and evaluate simple expressions (literals, variables, unary, binary and potentially n-ary operations), such examples can easily be constructed with it. Here is an example of TAT corresponding to the previous example:

Listing 1.1. A simple TAT

```
typedef BinOp< BinOp< ConstInt<5>, ConstInt<10>, Times >,
                                   ConstInt<1>, Plus >
               program_t;
```

However this covers a very small part of the whole programming language. Important features such as type declarations, function declarations and calls, or flow control cannot be expressed. Moreover, Tiger expressions are typed: we want our compiler to be able to evaluate and work on typed-expressions. When trying to translate more complex examples into TATs, different problems arise such as the list problem, or the reference problem.

4.1 The list problem

Let us consider this Tiger example:

Listing 1.2. Two functions

```
let
  function double(x : int) : int = 2 * x
  function sum(a : int, b : int, c : int): int = a + b + c
in
  double(30) - sum(6, 1, 2)
end
```

When building this program's TAT, we need to express lists: declaration lists, function formals lists, and function call arguments lists. The usual way to do this is to use recursive lists. A recursive list is defined as empty or as a head element followed by a tail list.

This can be transposed into C++ with the static list technique described by Veldhuizen (2002). We use a template class List, which parameters are the first element (a type), and the remaining list. A class EmptyList is used to mark the end of the list. With this notation, we can express lists as types. For example, in `sum (6, 1, 2)`, the argument list can be expressed with the following TAT:

```
List < ConstInt<6>,
  List < ConstInt<1>,
    List < ConstInt<2>,
      EmptyList
        >
      >
    >
```

The full TAT conversion of a similar sample is given in the next section. Static lists, which are a particular case of trees, will be used extensively in the remaining of this paper: this is our first addition to the Expression Templates technique.

4.2 The reference problem

The following simple example illustrate the reference problem:

Listing 1.3. Two variables addition

```
let
  var i : int := 80
  var j : int := 6
in
  i + j
end
```

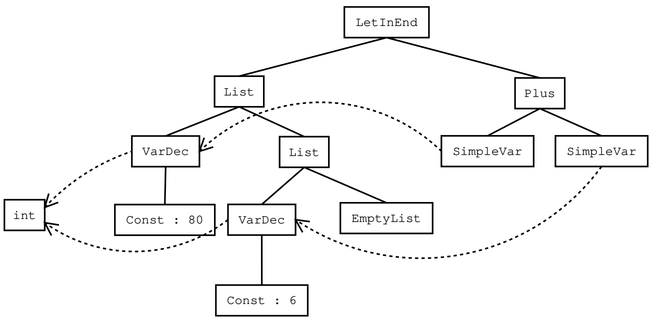

Fig. 2. AST of example 1.3

The expression `i` refers to the variable declaration `var i : int := 80`. The same way, the declaration `var i : int := 80` refers to the builtin type `int`. This example demonstrates that we cannot consider programs as simple trees. The main structure acts as a tree, but the implicit relations by reference transforms this tree into a DAG (direct acyclic graph).

The TAT has to describe a tree plus some graph relations between a declaration and its uses. This is the main difficulty compared to the Expression Templates technique. Without a reference mechanism, we cannot express concepts such as types or functions.

Each time a declaration is referred, we need a pointer to it. The following part shows how to solve this: each declaration will have a location in an evaluation environment.

5 Evaluation Environment

At every point in the program, there is a set of active declarations which can be used. An expression such as `i + j` (listing 1.3), or `double(30) - sum(6, 1, 2)` (listing 1.2) cannot be evaluated without the declaration context: we need to maintain an environment at evaluation time.

Tiger defines some builtin types and functions. These declarations, visible at every point in every Tiger program, will be the initial state of our environment. Declarations that have the same visibility are grouped into scopes. In the remainder of this paper, the list of declarations of the same scope is called a *chunk*.

The main operations we need on this environment are pushing and popping chunks. Moreover, we need a way to extract a declaration, given its chunk and its location in the chunk.

New declarations are introduced with the `let-in-end` structure, which is composed of two parts. A first declarative part, located between `let` and `in`, allows declaring a chunk. The second part, is an expression, in which we can use previous declarations. Evaluating the whole structure is done by pushing the chunk into environment, evaluating the expression and finally popping the chunk.

The environment can also be modified by a function call: when this occurs the evaluation point is changed. This implies that the set of active declarations changes.

Listing 1.4. A function call

```
let
  function double(x : int) : int = 2 * x
in
  let
    var i : int := 17
  in
    double(i) + i
  end
```

```
end
```

In the above example, the function call is evaluated the following way:

1. Evaluate function parameters: here i = 17.
2. Initialize formal values: x $\leftarrow$ i
3. Pop declarations introduced between the function declaration and the function call: this restores the environment of the function implementation. In our case: pop the chunk containing `var i : int := 17`, as the function double does not know this declaration.
4. Push formals declarations. Here: push a chunk containing `x : int`.
5. Evaluate the function body: `(2 * x)`
6. Restore callers environment: `x` does not exist any more, `i` is reintroduced.

At this point, a stack seems to be appropriate for our needs. This stack will be filled with declaration chunks. A declaration chunk simply contains the corresponding part of the TAT. At a given evaluation point, each visible declaration is located with a pair of indexes: the index of the chunk, and the index of the declaration in the chunk. So a simple pair of indexes is enough to refer to a declaration.

The example 1.3 can now be translated into the following TAT:

```
LetInEnd<
  List < Var< ConstInt < 80 >, builtin_types , int_type >,
    List < Var< ConstInt < 6 >, builtin_types , int_type >,
      EmptyList > >,
  BinOp< SimpleVar < 0, 0 >, SimpleVar < 0, 1 >, Plus >
>
```

The pair $< 0, 0 >$ refers to the first declaration of the first chunk, which corresponds to `var i:= 80`. The pair $< 0, 1 >$ refers to `var j:= 6`. `builtin_types` and `int_type` are predefined integer values, which identify the builtin `int` Tiger type. This mechanism of environment and location pair is a form of static pointers.

We are also able to translate example 1.4:

```
LetInEnd<                                          let
  List < Function< List <                           function double(
      TypeLnk< builtin_types , 1 > >,                          x : int) =
    BinOp< ConstInt < 2 >,                                2
        SimpleVar <1, 0 >, Times >,                         * x
     0 > >,
    LetInEnd<                                      in
      List < Variable < ConstInt < 17 >,             let
                 builtin_types , 1 > >,                var i : int := 17
                                                     in
        BinOp< FuncCall < 0, 0,                        double(
            List < SimpleVar < 1, 0 > > >,                    i)
        SimpleVar < 1, 0 >, Plus >                        + i
```

```
   >                                      end
>                                      end
```

Let's remember the goal: translating an AST into a C++ type (the TAT), so that the compiler can work on this type. In the proposed implementation, the environment related computations are done at compile-time. Meta-programming techniques will allow us to reduce the execution-time work considerably.

6 Implementation

The basis of the Expression Templates technique is to write a template class per kind of node available in the AST. The parameters of this template are the node subtrees. Each of these template classes correspond to a node of the AST.

These template classes fulfill two roles: first they express the AST information. This is implicitly done with class organization into the TAT. Second, our classes must provide evaluation code.

In the case of expressions, this consists on two tasks: the type calculation, and the value calculation. The declaration classes provide some other services such as common operations for types.

Apart from AST meta-classes, we also need to provide meta-code to perform some static processing. This corresponds to the set of operations related to the evaluation environment.

6.1 Global organization

Two kinds of classes have to be written: expression classes and declaration classes. Declarations will be further distinguished via classes specialized for type, variable and function declarations. Moreover, the implementation also includes the environment mechanism, and tools for its manipulation.

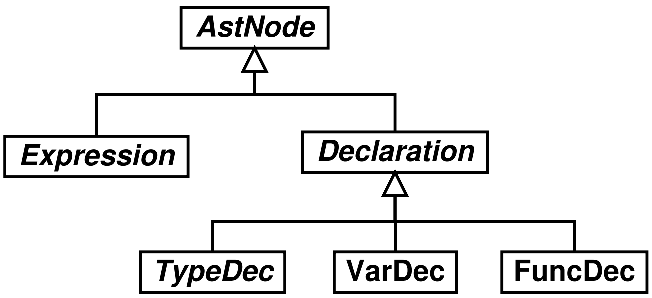

Fig. 3. Main kind of classes

Note that the base classes `AstNode`, `Expression`, `Declaration`, `TypeDec`... are only used for some static checking. These classes are not very interesting, and will not be detailed in this paper.

6.2 Expression classes

As in the Expression Templates technique, each Expression class will implement an evaluation method. These methods are inlined, so that the C++ compiler can build efficient evaluation code.

The main difference with Expression Templates is due to the evaluation environment: The evaluation method depends on the current environment. Another striking difference is that expressions are typed. Evaluating an expression consists in computing both its type and value. We want expression types to be evaluated statically: this work will be done through typedefs. All the typed values that we manipulate are represented with four bytes. In order to simplify, we decided to represent all variables with the `void*` type. This lead us to the following model adopted by all expression classes:

```
// var_t represent a non-typed value.
typedef void* var_t;

// Here comes the template parameters: the TAT subtrees.
template< ... >
struct AnExpression: public Expression
{
  // Evaluation is dependent of current environment.
  template<class T_env>
  struct eval
  {
    // statically compute the expression type
    typedef ... T;

    // inline method that evaluates the expression value
    inline var_t doit ()     { ... }
  };
};
```

```
template<signed Value>
struct ConstInt: public Expression
{
  template<class T_env>
  struct eval
  {
    typedef IntType          T;
    inline var_t doit ()     {return (var_t)Value;}
  };
};
```

ConstInt $< 123 >$ is a TAT: its value and type can be evaluated:

```
typedef ConstInt<123>  program_t;

var_t value = program_t::eval< initial_env_t >::doit();
typedef program_t::eval< initial_env_t >::T  type;
```

Notice that "`::`" is C++ for Java "`.`"

Two types are predefined:

- `var_t` represents all Tiger variables. For example, an `int` can directly be casted into a `var_t` (these two types have the same size: four bytes). Most of time a `var_t` corresponds to a record pointer or an array pointer.
- `initial_env_t` corresponds to the Tiger builtin environment: builtin types such as `IntType` or `StringType`, and builtin functions (`print`, `ord`, `concat`...).

The TAT given in listing 1.1 can now be evaluated. Here is the template expansion chain that leaded to the result: 51.

```
1. program_t::eval< initial_env_t >::doit()
2. BinOP< ConstInt<5>, ConstInt<10>, Times >
                ::eval< initial_env_t >::doit() +
     ConstInt<1>::eval< initial_env_t >::doit();
3. ConstInt<5>::eval< initial_env_t >::doit() *
   ConstInt<10>::eval< initial_env_t >::doit() + 1;
4. 5 * 10 + 1
5. 51
```

6.3 Declaration classes

The first role of declaration classes is to store information relative to the declaration. For example a variable declaration must store its type and initial value. This is done with template parameters exactly as above. The second role of declaration classes depends on the kind of declaration. For variables and functions, only some utility functions are implemented. The type classes do more things: their second role is to implement all operations related to the type: assignment, comparison, creation and destruction. These operations can depend on the environment. This is for example the case for an array, which refers to the type of its elements.

Here is the model of type declaration classes:

```
struct AType: public TypeDec
{
  // Type eval depends on environment
  template<class T_env>
  struct eval
  {
    // Common operations are implemented here
    void create(var_t& v);
    void destroy(var_t v);
```

```
    void assign(var_t& left, var_t right);
    int compare(var_t left, var_t right);
  };
};
```

Such classes are implemented for `VoidType`, `IntType`, `StringType`, `ArrayType` and `RecordType`.

Each new type definition in a Tiger program, will result in new type operations. Our Tiger compiler generates evaluation code, but also operations code. In order to emphasis on this contribution, we chose to implement assignment and comparison as structural. At the contrary to the Tiger specifications, when two records are compared, this is done member by member. When an array is assigned, the all content is copied.

6.4 Program Environment

We have seen that type and expression evaluations depend on an environment, through the type identified by `T_env` in the previous code samples. We want the environment to be computed statically: we need an implementation which allows to push, pop, and retrieve declarations at compile-time. Therefore we use again static lists: an environment is implemented as a static list of declaration chunks. A declaration chunk is a part the TAT which is also a static list. This construction allows us to manipulate the environment:

Pushing and popping declaration chunks is done with typedefs:

```
// Push T_new_chunk on T_env, yielding T_new_env.
typedef List< T_new_chunk, T_env >        T_new_env;

// Pop an element of T_env, yielding T_new_env.
typedef T_env::tail                       T_new_env;
```

Environment access is done with a template class and a specialization:

```
template<class T_env, unsigned N>
struct ListGet
{
  typedef ListGet<T_env::tail, N - 1>::T  T;
};

template<class T_env>
struct ListGet<T_env, 0>
{
  typedef T_env::head      T;
};

// Access to the chunk number 3.
typedef typename ListGet<T_env, 3>::T          T_chunk_3;
// Access to the declaration number 1 of this chunk.
typedef typename ListGet<T_chunk_3, 1>::T      T_decl_3_1;
```

We are now able to write a simplified version of the `LetInEnd` template class.

```
template<class T_decl, class T_exp>
struct LetInEnd
{
  template<class T_env>
  struct eval
  {
    typedef List<T_decl, T_env>             T_new_env;
    typedef T_exp::eval<T_new_env>::T       T;
    var_t doit()
    {
      // Create new variables declared in T_decl
      // (not detailed here).

      // Evaluate the expression in the new environment.
      var_t res = T_exp::eval<T_new_env>::doit();

      // Destroy the variables declared in T_decl
      // (not detailed here).

      return res;
    }
  };
};
```

All the needed operations on the environment can be done with type operations: we are able to fully compute the environment at compile-time for each evaluation point. Function calls are not detailed here, but they use the same operations. Note that all functions are evaluated each time they are called (as inline functions). This implies that if we want a recursive function to be translated as a C++ recursive function, we need the environment to be exactly the same at each recursive call.

6.5 The dynamic part

Not everything can be done at compile-time. The Tiger language allows some constructions which cannot be resolved statically.

The main dynamic stuff is the variable declaration and use. When a variable is declared, we need to store its value somewhere. At each evaluation point of the program, there is a set of variables which are accessible.

A variable can be of any supported Tiger type: it can be an array, a string or a record. There is no static representation of such values: we need to store this into memory at the program execution. Therefore we use the C++ stack: variables declared in a `let-in-end` construction are declared as local variables in the `LetInEnd` evaluation method.

The Tiger has a nested let declaration. At a given evaluation point, there can be several visible scopes. This obliges us to maintain a stack of scope pointers during the whole execution process. Accessing a variable is performed with two indirections: a first one to get the right scope and another one to reach the variable into this scope. We could have chose to implement variables access with a static link mechanism. This would corresponded to the adaptable closure present in the phoenix library, part of the spirit project (group (2002)).

These indirections are our main limitation to really perform a static resolution of programs. Conversely, here is a program that is *entirely* statically evaluated:

Listing 1.5. A program solved statically

```
let
  function foo() = 20 * 20
  function bar() = 30 / 2
  function smousse() = if (80 > 6) then 1 else 0
in
   (foo() + bar() + smousse()) * 4
end
```

There is no variable used so, after our transform towards C++, we can expect that a C++ compiler can statically solve this program. In this particular case, using a C++ compiler which has good optimization capacities, we directly obtain one assembler instruction which gives the integer result.

6.6 The C++ program

The C++ program always have the same structure:

```
// Include all template classes needed to express and evaluate
   the AST.
#include "all.h"

// Generate the TAT.
typedef ... program_t;

int main()
{
  // instantiate program evaluation
  return (int)program_t::eval< initial_env_t >::doit();
}
```

The line of the main() launch the doit() instantiation, which results in the generation of the program evaluation code. This work is done by the C++ compiler.

7 Results

Our compiler covers all of the Tiger language. Lot of Tiger programs have been tested, and work successfully. Our process has been tested with como, g++ 3.2 and icc which gives slightly faster programs.

To experiment the performance of generated code, some Tiger programs compiled with our process have been compared to their C hard-coded equivalent. In average, the C program goes two to three times faster than the (C++) Tiger one. This performance lack is mostly explained by the variable access cost: each access needs two indirections. But viewed as an evaluation process, this can be considered as good results.

This performance highly depends on the aptitude of the C++ compiler to optimize code. These optimizations are essentially obtained by the inlining mechanism. This optimization has been tested using the g++ option called `-finline-limit`. This option influences the quantity of functions inlined. This experience showed us the importance of good inlining at compile-time. Optimizations are done until approximatively `-finline-limit-1000`, which is much more than for usual C++ programs. This can be explained by the amount of functions that are instantiated. Indeed for each node of the AST, there is at least one function which will be used.

8 Conclusion

We have seen that a program can be expressed as an Abstract Syntax Tree (AST) given the language grammar. Using a technique based on Expression Templates, we are able to build a C++ type which describes this AST. This representation is called the TAT (Tree As Type).

Building and evaluating the TAT poses various problems. We need to express lists (for declarations, arguments, etc.). This problem is solved using the Static list technique. In the TAT, some elements refer to others. The reference problem implies the use of an environment which is implemented using a stack. We have seen that this container allows the required operations: pushing, popping and accessing. This stack is directly filled with parts of the TAT: this is a form of static pointers, which solves the reference problem.

An implementation based on the Tiger language has been proposed. This implementation intensively uses meta-programming techniques, therefore, the C++ compiler is able to do lot of work at compile-time: expression types and element references are solved statically. The limits of static resolution is the use of variables which can only be manipulated dynamically.

Our Tiger compiler is originally inspired by the Expression Templates technique. However, the evaluated constructions are not restricted to basic ones, such as unary or binary operators, but includes the common flow control constructions, structured types, variables, and nested functions. Moreover, thanks to the use of a static environment, such advanced operations can be evaluated by jumping from one point of the program to another. This happens for example each time a function is called. That characteristic is a noticeable difference with the Expression Templates which are evaluated in a simple bottom-up fashion.

This original technique shows how we used C++ meta-programming in order to work on abstract syntax trees of a mostly functional programming language.

Indeed the C++ generative power allowed us to implement compiler parts and translation into C++ equivalent code.

Bibliography

A. Appel. *Modern Compiler Implementation in C / Java / ML*. Cambridge University Press, 1997.

J. Crotinger, J. Cummings, S. Haney, W. Humphrey, S. Karmesin, J. Reynders, S. Smith, and T. Williams. Generic programming in POOMA and PETE. In *Generic Programming, Proceedings of the International Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 218–. Springer-Verlag, 2000. URL http://www.acl.lanl.gov/pete/.

K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques and Applications*. Addison-Wesley, 2000.

S. group. Spirit parser framework, 2002. URL http://spirit.sourceforge.net.

S. Haney and J. Crotinger. How templates enable high-performance scientific computing in C++. *Computing in Science and Engineering*, 1(4), 1999. URL http://www.acl.lanl.gov/pooma/papers.html.

G. P. J. Jarvi. The boost lambda library, 2002. URL http://www.boost.org/libs/lambda/doc/.

J. Järvi. Compile time recursive objects in C++. In *Technology of Object-Oriented Languages and Systems*, pages 66–77. IEEE Computer Society Press, 1998.

B. McNamara and Y. Smaragdakis. Functional programming in C++ using the FC++ library. *SIGPLAN Notices*, April 2001.

J. Striegnitz and S. A. Smith. An expression template aware lambda function. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000. URL http://oonumerics.org/tmpw00/.

C. van Reeuwijk. Rapid and robust compiler construction using template-based metacompilation. In *12th International Conference on Compiler Construction*, Lecture Notes in Computer Science, pages 247–, Warsaw, Poland, April 2003. Springer-Verlag.

T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

T. Veldhuizen. Techniques for scientific C++. Technical report, Computer Science Department, Indiana University, Bloomington, USA, 2002. URL http://osl.iu.edi/ tveldhui/papers/techniques/.

A A simple C++ meta-program and its evaluation

```
template<unsigned i>
struct factorial
{
  enum {res = i * factorial< i - 1 >::res};
};
```

```
template<>
struct factorial<0>
{
  enum {res = 1};
};

enum { fact4 = factorial<4>::res };
```

Thanks to the template expansion mechanism, this C++ meta-function allows to compute a factorial at compile-time:

```
factorial<4>::res
4 * factorial<3>::res
4 * 3 * factorial<2>::res
4 * 3 * 2 * factorial<1>::res
4 * 3 * 2 * 1 * factorial<0>::res
4 * 3 * 2 * 1 * 1
24
```

B A full tiger program

```
let
 type any = {any : int}
 var buffer := getchar()

 function printint(i: int) =
  let function f(i:int) = if i>0
             then (f(i/10); print(chr(i-i/10*10+ord("0"))))
   in if i<0 then (print("-"); f(-i))
      else if i>0 then f(i)
      else print("0")
  end


function readint(any: any) : int =
 let var i := 0
     function isdigit(s : string) : int =
                  ord(buffer)>=ord("0") & ord(buffer)<=ord
                    ("9")
     function skipto() =
       while buffer=" " | buffer="\n"
         do buffer := getchar()
  in skipto();
     any.any := isdigit(buffer);
     while isdigit(buffer)
       do (i := i*10+ord(buffer)-ord("0"); buffer := getchar()
         );
     i
```

```
end

type list = {first: int, rest: list}

function readlist() : list =
   let var any := any{any=0}
       var i := readint(any)
    in if any.any
        then list{first=i,rest=readlist()}
        else nil
   end

function merge(a: list, b: list) : list =
  if a=nil then b
  else if b=nil then a
  else if a.first < b.first
     then list{first=a.first,rest=merge(a.rest,b)}
     else list{first=b.first,rest=merge(a,b.rest)}

function printlist(l: list) =
  if l=nil then print("\n")
  else (printint(l.first); print(" "); printlist(l.rest))

  var list1 := readlist()
  var list2 := (buffer:=getchar(); readlist())

in
        print ("list 1 : \n");
        printlist (list1);
        print ("list 2 : \n");
        printlist (list2);
        print ("merged list : \n");
        printlist (merge (list1, list2) )
end
```

C TAT of the previous program

The following program compiles in less than two minutes with g++ 3.2 on a 350Mhz processor.

```
#include "all.h"

typedef LetInEnd< List< RecordType< List< TypeLnk< builtin_types
  , 1 > > > >,
LetInEnd< List< Variable< FuncCall< builtin_funcs, 9, List< > >,
  builtin_types, 2 > >,
LetInEnd< List<
Function< List< TypeLnk< builtin_types, 1 > >, LetInEnd< List<
Function< List< TypeLnk< builtin_types, 1 > >, If< BinOp< SimpleVar
  < 5, 0 >, ConstInt< 0 >, GreatThan >, ExpList< FuncCall< 4, 0, List<
  BinOp< SimpleVar< 5, 0 >, ConstInt< 10 >, Divide > > >, ExpList<
```

```
  FuncCall< builtin_funcs , 0, List< FuncCall< builtin_funcs , 4, List<
  BinOp< BinOp< SimpleVar< 5, 0 >, BinOp< BinOp< SimpleVar< 5, 0 >,
  ConstInt< 10 >, Divide >, ConstInt< 10 >, Times >, Minus >, FuncCall<
  builtin_funcs , 3, List< ConstString< 0 > > >, Plus
   > > > > > > > > >, 4 >
 >,
ExpList< If< BinOp< SimpleVar< 3, 0 >, ConstInt< 0 >, LessThan >,
  ExpList< FuncCall< builtin_funcs , 0, List< ConstString< 1 > > >,
  ExpList< FuncCall< 4, 0, List< BinOp< ConstInt< 0 >, SimpleVar
  < 3, 0 >, Minus > > > > >, If< BinOp< SimpleVar< 3, 0 >, ConstInt
  < 0 >, GreatThan >, FuncCall< 4, 0, List< SimpleVar< 3, 0 > > >,
  FuncCall< builtin_funcs , 0, List< ConstString< 2 > > > > > > > >
, 2 >
, List<
Function< List< TypeLnk< 0, 0 > >, LetInEnd< List< Variable< ConstInt
  < 0 >, builtin_types , 1 > >,
LetInEnd< List<
Function< List< TypeLnk< builtin_types , 2 > >, If< BinOp< FuncCall<
  builtin_funcs , 3, List< SimpleVar< 1, 0 > > >, FuncCall< builtin_funcs
  , 3, List< ConstString< 3 > > >, GreatEq >, BinOp< FuncCall<
  builtin_funcs , 3, List< SimpleVar< 1, 0 > > >, FuncCall< builtin_funcs
  , 3, List< ConstString< 4 > > >, LessEq >, ConstInt< 0 > >, 5 >
, List<
Function< List< >, While< If< BinOp< SimpleVar< 1, 0 >, ConstString
  < 5 >, Equal >, ConstInt< 1 >, BinOp< SimpleVar< 1, 0 >, ConstString
  < 6 >, Equal > >, Assign< SimpleVar< 1, 0 >, FuncCall< builtin_funcs
  , 9, List< > > > >, 5 >
 > >,
ExpList< FuncCall< 5, 1, List< > >, ExpList< Assign< FieldVar<
  SimpleVar< 3, 0 >, 0 >, FuncCall< 5, 0, List< SimpleVar
  < 1, 0 > > > >, ExpList< While< FuncCall< 5, 0, List< SimpleVar
  < 1, 0 > > >, ExpList< Assign< SimpleVar< 4, 0 >, BinOp< BinOp< BinOp
  < SimpleVar< 4, 0 >, ConstInt< 10 >, Times >, FuncCall< builtin_funcs
  , 3, List< SimpleVar< 1, 0 > > >, Plus >, FuncCall< builtin_funcs , 3,
  List< ConstString< 7 > > >, Minus > >, ExpList< Assign< SimpleVar
  < 1, 0 >, FuncCall< builtin_funcs , 9, List< > > > > > > >, ExpList<
  SimpleVar< 4, 0 > > > > > > >
, 2 >
 > >,
LetInEnd< List< RecordType< List< TypeLnk< builtin_types , 1 >, List<
  TypeLnk< 3, 0 > > > > >,
LetInEnd< List<
Function< List< >, LetInEnd< List< Variable< Record< 0, 0, List<
  ConstInt< 0 > > >, 0, 0 >, List< Variable< FuncCall< 2, 1, List<
  SimpleVar< 6, 0 > > >, builtin_types , 1 > > >,
ExpList< If< FieldVar< SimpleVar< 6, 0 >, 0 >, Record< 3, 0, List<
  SimpleVar< 6, 1 >, List< FuncCall< 4, 0, List< > > > > >, Nil > > >
, 4 >
, List<
Function< List< TypeLnk< 3, 0 >, List< TypeLnk< 3, 0 > > >, If< BinOp<
  SimpleVar< 5, 0 >, Nil , Equal >, SimpleVar< 5, 1 >, If< BinOp<
  SimpleVar< 5, 1 >, Nil , Equal >, SimpleVar< 5, 0 >, If< BinOp<
  FieldVar< SimpleVar< 5, 0 >, 0 >, FieldVar< SimpleVar< 5, 1 >, 0 >,
  LessThan >, Record< 3, 0, List< FieldVar< SimpleVar< 5, 0 >, 0 >, List
  < FuncCall< 4, 1, List< FieldVar< SimpleVar< 5, 0 >, 1 >, List<
  SimpleVar< 5, 1 > > > > > > > >, Record< 3, 0, List< FieldVar< SimpleVar
  < 5, 1 >, 0 >, List< FuncCall< 4, 1, List< SimpleVar< 5, 0 >, List<
  FieldVar< SimpleVar< 5, 1 >, 1 > > > > > > > > > > > >, 4 >
, List<
Function< List< TypeLnk< 3, 0 > >, If< BinOp< SimpleVar< 5, 0 >, Nil ,
  Equal >, FuncCall< builtin_funcs , 0, List< ConstString< 8 > > >,
  ExpList< FuncCall< 2, 0, List< FieldVar< SimpleVar< 5, 0 >, 0 > > >,
  ExpList< FuncCall< builtin_funcs , 0, List< ConstString< 9 > > >,
  ExpList< FuncCall< 4, 2, List< FieldVar< SimpleVar
  < 5, 0 >, 1 > > > > > > > > >, 4 >
 > > >,
LetInEnd< List< Variable< FuncCall< 4, 0, List< > >, builtin_types
  , 0 >, List< Variable< ExpList< Assign< SimpleVar< 1, 0 >, FuncCall<
```

```
  builtin_funcs , 9 , List <  > > >, ExpList < FuncCall < 4 , 0 , List <
   > > > >, builtin_types , 0 > > >,
ExpList < FuncCall < builtin_funcs , 0 , List < ConstString < 10 > > >,
  ExpList < FuncCall < 4 , 2 , List < SimpleVar < 5, 0 > > >, ExpList <
  FuncCall < builtin_funcs , 0 , List < ConstString < 11 > > >, ExpList <
  FuncCall < 4 , 2 , List < SimpleVar < 5, 1 > > >, ExpList < FuncCall <
  builtin_funcs , 0 , List < ConstString < 12 > > >, ExpList < FuncCall
  < 4 , 2 , List < FuncCall < 4 , 1 , List < SimpleVar < 5, 0 >, List < SimpleVar
  < 5, 1 > > > > > > > > > > > > > > > > > >

        program_t ;

const char*     metasmousse::const_string[] = {"0", "-", "0", "0", "9",
  " ", "\012", "0", "\012", " ", "list 1 : \012", "list 2 : \012", "
  merged list : \012", NULL};

int main()
{
  return (int)program_t::eval< initial_env_t >::doit();
}
```