A Tool for the Evaluation of the Complexity of Programs Using C++ Templates

Nicola Corriero, Emanuele Covino and Giovanni Pani
Dipartimento di Informatica
Università di Bari
Italy
Email: (corriero|covino|pani)@di.uniba.it

***Abstract*—We investigate the relationship between C++ template metaprogramming and computational complexity, showing how templates characterize the class of polynomial-time computable functions, by means of template recursion and specialization. Hence, standard C++ compilers can be used as a tool to certify polytime-bounded programs.**

***Keywords*-Partial evaluation; C++ template metaprogramming; polynomial-time programs.**

I. Introduction

According to [7], template metaprograms consist of classes of templates operating on numbers and types as a data. Algorithms are expressed using template recursion as a looping construct and template specialization as a conditional construct. Template recursion involves the use of class templates in the construction of its own member type or member constant. Templates were introduced to C++ to support generic programming and code reuse through parameterization. This is done by defining generic functions and objects whose behaviour is customized by means of parameters that must be known at compile time, entirely. For example, a generic vector class can be declared in C++ as follows:

```
template <class T, int N> class vector
  { T data[N]; };
```

The class has two parameters: T, the type of the vector's elements, and N, the length of the vector. The command line vector<int,5> instantiates the template by replacing all occurrences of T and N in the definition of vector with int and 5, respectively.

Templates are also able to perform static computation. The first example of this behaviour was reported in [22] and [23], where a program that forces the compiler to calculate (at compile time) a list of prime numbers is written; this ability is largely described by [7], [25], [26] and [9]: C++ may be regarded as a 2-level language, in which types are first-class values, and template instantiation mimics off-line partial evaluation. For instance, the following templates compute the function $pow(y, x) = x^y$;

```
template <int Y, int X> class pow
  {public: enum {result=X*pow<Y-1,X>::result };};
template <int X> class pow<0,X>
  {public: enum {result=1};};
```

The command line int z=pow<3,5>::result, produces at compile time the value 125, since the operator A::B refers to the symbol B in the scope of A; when reading the command pow<3,5>::result, the compiler triggers recursively the template for the values <2,5>, <1,5>, until it eventually hits <0,5>. This final case is handled by the partially specialized template pow<0,X>, that returns 1. Instructions like enum{result = function<args>::result;} represent the step of the recursive evaluation of function, and produce the intermediate values. This computation happens at compile time, since enumeration values are not l-values, and when one pass them to the recursive call of a template, no static memory is used (see [24], chapter 17). Thus, the compiler is used to compute *metafunctions*, that is as an interpreter for metaprogramming.

In [28], the following definition is given: a restricted metalanguage *captures* a property when every program written in the restricted metalanguage has the property and, conversely, for every unrestricted program with the property, there exists a functionally equivalent program written in the restricted metalanguage. An example of capturing a property by means of a restricted language is given in [3]: *any partial recursive function can be computed at compile-time* returning an error message that contains the result of the function. This is achieved specifying primitive recursion, composition, and μ-recursion by means of C++ template metaprogramming. A sketch of this result is in Section II-B.

On the other hand, the problem of defining syntactical characterizations of complexity classes of functions has been faced during the 90's; this approach has been dubbed *Implicit Computational Complexity* (ICC), and it aims at studying the complexity of programs without referring to a particular machine model and explicit bounds on time or memory. Several approaches have been explored for that purpose, like linear logic, rewriting systems, types and lambda-calculus, restrictions on primitive recursion. Two objectives of ICC are to find natural implicit characterizations of functions of various complexity classes, and to design systems suitable for static verification of programs complexity. In particular,

[2], [8], [11], [13], [14] studied how restricted functional languages can capture complexity classes.

In this paper we investigate the relationship between template metaprogramming and ICC, by defining a recursive metalanguage by means of C++ templates; we show that it captures the set of polynomial-time computable functions, that is, functions computable by a Turing machine in which the number of moves — the time complexity — is bounded by a polynomial. We also show that our approach can be extended to recursion schemes that are more general than those used in ICC. This result makes two contributions. First, it represents an approach to the automatic certification of upper bounds for time consumption of metaprograms. The compilation process certifies the complexity of the program, returning a specific error when the complexity is not polynomial. Second, there are few, if not any, characterizations of complexity classes made by metaprogramming; in particular, the result is achieved with a real, industrial template language, one that was constructed for doing real programming. Moreover, we do not define any extension of the language; we simply use the existing C++ type system to perform the computation.

The paper is organized as follows: in Section II we discuss some works related with our approach, and we recall some known results that we will use later; in Section III we show how to represent some polytime computable functions by means of template metaprogramming and how to rule out those functions that are not polytime (this is done by imposing some restrictions on the role of the template arguments); in Section IV we define the *Poly-Temp* language; in Section V we show that *Poly-Temp* is equivalent to the class of polynomial-time computable functions; finally, conclusions and further work are in Section VI.

II. Related works

A. C++ metaprogramming and functional programming

The prevailing style of programming in C++ is imperative. However, the mechanics of C++ metaprogramming shows a clear resemblance to dynamically-typed functional language, where all metaprograms are evaluated at compile time. This is clearly stated in [19]: they extended the purely functional language Haskell with compile-time metaprogramming (i.e., with a template system *à la* C++), with the main purpose to support the algorithmic construction of programs at compile-time.

In [12], it is recalled how function closure can be modelled in C++ by enclosing a function inside an object such that the local environment is captured by data members of the object; this idiom can be generalized to a type-safe framework of C++ class templates for higher-order functions that supports composition and partial application, showing that object-oriented and functional idioms can coexist productively. In [15] and [16], a rich library supporting functional programming in C++ is described, in which templates and C++ type inference are used to represent polymorphic functions. Another similar approach is in [20], where a functional language inside C++ is provided by means of templates and operator overloading. Both approaches provides functional-like libraries in run-time, while computations made by means of our metalanguage are performed at compile-time, totally. Coevally, [10] developed the template-implemented Lambda Library, which adds a form of lambda functions to C++. All these approaches lead to the introduction of lambda expression in C++ standard.

B. The computational power of C++ compilers

The first attempt to use C++ metaprogramming to capture a significant class of functions has been made by [3]; they presented a way to specify primitive recursion, composition, and μ-recursion by means of C++ templates. The result is not astonishing, provided that C++ templates are Turing complete (see [27]), but the reader should note that the technique used in the paper is based on the partial evaluation process performed by C++ compilers.

Number types are used to represent numbers; the number type representing zero is `class zero { }`. Given a number type `T`, the number type representing its unary successor is `template<class T> class suc { typedef T pre;}`.

A function is represented by a C++ class template, in which templates arguments are the arguments of the function. For example,

```
template<class T> class plus2
  {typedef suc<suc<T>> result;};
```

is a function type that computes the function $f(x) = x + 2$. The instructions

```
plus2<suc<zero>>::result tmp;
return (int) tmp;
```

returns an error message including `suc<suc<suc<zero>>>`, that is the value of $f(1)$. In particular, given a two-variable function f defined by primitive recursion from g and h,

$$\begin{cases} f(0,x) = & g(x) \\ f(y+1,x) = & h(y,x,f(y,x)) \end{cases}$$

function type `F` of f is expressed by the following templates, where `G` and `H` are the class templates computing g and h.

```
template <class Y, class X> class F
  {typedef typename
      H<typename Y::pre, X,
      typename F<typename Y::pre, X>::result >
                    ::result result;};

template <class X> class F<zero,X>
  {typedef typename G<X>::result result; };
```

Similar templates can be written to represent composition and μ-recursion, and to extend the definition to the general case on n variables; thus, the whole class of partial recursive functions can be expressed by template metaprogramming. For instance, templates times and pow can be defined as follows, where add returns the sum of its two arguments and is defined in the same way.

```
template<class Y, class X> class times
  {typedef typename
     add<X, typename times<typename Y::pre, X>
        ::result>::result result;};

template <class X> class times<zero,X>
  {typedef zero result};

template<class Y, class X> class pow
  {typedef typename
     times<X, typename pow<typename Y::pre,X>
                              ::result>::result result;};

template <class X> class pow<zero,X>
  {typedef zero result}.
```

C. Capturing complexity classes by function-theoretic characterization

Syntactical characterizations of relevant classes of functions have been introduced by the Implicit Computational Complexity approach, that studies how restricted functional languages can capture complexity classes; in general, several restricted recursion schemes have been introduced, all sharing the same feature: no explicit bounds (as in [4]) are imposed in the definition of functions by recursion.

In order to show the relation between template metaprogramming and polynomial-time computable functions we need to recall that this class is defined in [2] as the smallest class B containing some initial functions, and closed under *safe recursion on notation* and *safe composition*. This result is obtained by imposing a syntactic restriction on variables used in the recursion and composition; they are distinguished in normal or safe, and the latter cannot be used as the principal variable of a function defined by recursion. In other words, one does not allow (safe) recursive terms to be substituted into a (normal) position which was used for an earlier definition by recursion. Normal inputs are written to the left, and they are separated from the safe inputs by means af a semicolon. A function in B can be written as $f(\vec{x};\vec{y})$; in this case, variables x_i are normal, whereas variables y_j are safe.

B is the smallest class of functions containing the initial functions 1-5 and closed under 6 and 7.

1) **Constant:** 0 (it is 0-ary function).
2) **Projection:** $\pi_j^{n,m}(x_1\ldots,x_n;x_{n+1}\ldots,x_{n+m}) = x_j$, for $1 \le j \le m+n$.
3) **Binary successor:** $s_i(;a) = ai$, $i \in \{0,1\}$.
4) **Binary predecessor:** $p(;0) = 0$, $p(;ai) = a$.
5) **Conditional:**
$$C(;a,b,c) = \begin{cases} b & \text{if } a \bmod 2 = 0 \\ c & \text{otherwise.} \end{cases}$$
6) **Safe recursion on notation:** the function f is defined by safe recursion on notation from functions g and h_i if
$$\begin{cases} f(0,\vec{x};\vec{a}) = & g(\vec{x};\vec{a}) \\ f(yi,\vec{x};\vec{a}) = & h_i(y,\vec{x};\vec{a},f(y,\vec{x};\vec{a})). \end{cases}$$
for $i \in \{0,1\}$, g and h_i in B; y is called the *principal variable* of the recursion.
7) **Safe composition:** the function f is defined by safe composition from functions h, $\vec{r}$ and $\vec{t}$ if
$$f(\vec{x};\vec{a}) = h(\vec{r}(\vec{x};);\vec{t}(\vec{x};\vec{a}))$$
for h, $\vec{r}$ and $\vec{t}$ in B.

When defining a function $f(yi,\vec{x};\vec{a})$ by safe recursion on notation from g and h_i, the value $f(y,\vec{x};\vec{a})$ is in a safe position of h_i (right-side of the semicolon); and a function having safe variables cannot be substituted into a normal position of any other function, according to the definition of safe composition. Moreover, normal variables can be moved into a safe position, but not viceversa. By constraining recursion and composition in such a way, class B results to be equivalent to the class of polynomial time computable functions.

III. Template representations of some polytime functions

We show how to represent some polytime computable functions by means of template metaprogramming, imposing some restrictions on the role of the template arguments, following the mechanism introduced in [2]. Functions $\oplus$ and $\otimes$ can be expressed by safe recursion as follows:

$$\begin{cases} \oplus(0;x) = & x \\ \oplus(y+1;x) = & succ(;\oplus(y;x)). \end{cases}$$

$$\begin{cases} \otimes(0;a) = & 0 \\ \otimes(b+1;a) = & \oplus(a;\otimes(b;a)). \end{cases}$$

The recursive call $\otimes(b;a)$ is assigned to the safe variable x of $\oplus$, and one cannot re-assign this value to a normal variable of $\oplus$ (by definition 7, previous section, one cannot assign a function with safe variables to a normal position). For this reason, the following definition of $\otimes$ and both definitions of $\uparrow$ are not allowed in B.

$$\begin{cases} \otimes(0;a) = & 0 \\ \otimes(b+1;a) = & \oplus(\otimes(b;a);a). \end{cases}$$

$$\begin{cases} \uparrow(;0,x) = & 1 \\ \uparrow(;y+1,x) = & \otimes(x;\uparrow(;y,x)). \end{cases}$$

$$\begin{cases} \uparrow(;0,x) = & 1 \\ \uparrow(;y+1,x) = & \otimes(\uparrow(;y,x);x). \end{cases}$$

When defining the C++ templates that represent the previous three functions (or, in general, functions in B), we have to mimic the normal/safe behaviour by putting beside each variable a two-value flag; flags' values are defined according to the following rules:

1) each flag is equal to normal, initially;
2) flags beside variables assigned with recursive calls are changed to safe;
3) a compiler error must be generated whenever a variable labelled with a safe flag is used as principal variable of a recursion; this is done by adding a *negative specialization* (see below for its definition).

The template representation of $\oplus$ is the following (for sake of conciseness, we use enumeration values instead of typedef typename definitions, and integers instead of their class template representation):

```
♯define normal 0;
♯define safe 1;

template<int Y, int flagy, int X, int flagx> class sum
  {enum {result= 1+sum<Y-1, flagy, X, flagx>::result };};

template<int flagy, int X, int flagx>
  class sum<0, flagy, X, flagx> {enum {result= X };};

template<int Y, int X, int flagx>
  class sum<Y, safe, X, flagx>
  {enum {result= sum<Y, safe, X, flagx>::result };};
```

The instruction sum<2,normal,3,normal>::result returns the expected value, by recursively instantiating the first template sum for the values <2,3> and <1,3>, until <0,3> is reached (we omit here the flags); this value matches the second specialized template, which returns 3. The third template is introduced to avoid the substitution of other recursive calls or functions into variable Y, according to previous rule 3. This specialization is in the general form

```
template <args> class error <spec-args>
  {enum {result= error<spec-args>::result };};
```

and in this case the compiler stops, producing the error 'result' is not a member of type 'error<spec-args>'. The template representation of $\otimes$ is the following:

```
template<int Y, int flagy, int X, int flagx> class prod
  {enum {result=
  sum<X,flagx, prod<Y-1, flagy, X, flagx>::result, safe
          >::result}};

template<int flagy, int X, int flagx>
  class prod<0, flagy, X, flagx> {enum {result= 0};};

template<int Y, int X, int flagx>
  class prod<Y, safe, X, flagx>
  {enum {result= prod<Y, safe, X, flagx>::result}};
```

By rule 2, the flag associated with the recursive call of prod which occurs into sum is switched to safe, and by rule 3, the last template specialization is introduced to prevent the programmer from assigning another recursive call or function to Y. The instruction prod<2,normal,3,normal>::result instantiates the first template sum for values 3, normal, prod<1,normal,3,normal>::result, and safe, respectively; thus, the product is recursively evaluated. As shown above, one can also define prod by exchanging the arguments of sum, that is by assigning the recursive call of prod to the safe variable of sum, as follows:

```
template<int Y, int flagy, int X, int flagx> class prod
  {enum {result=
  sum< prod<Y-1, flagy, X, flagx>::result, safe, X, flagx
          >::result}};
```

The instruction prod<2,normal,3,normal>::result instantiates the template sum for values prod<1,normal, 3,normal>::result, safe, 3, and normal, respectively; this instantiation matches the values of the third template of sum's definition, and a compile-time error is produced (as expected, since we are trying to assign the recursive call of prod to the principal variable of sum). The template representation of the exponential function is

```
template<int Y, int flagy, int X, int flagx> class esp
  {enum {result=
  prod<esp<Y-1, flagy, X, flagx>::result, safe, X, flagx
          >::result}};

template<int flagy, int X, int flagx>
  class esp<0, flagy, X, flagx>
  {enum {result=1 };};

template<int Y, int X, int flagx>
  class esp<Y, safe, X, flagx>
  {enum {result= esp<Y, safe, X, flagx>::result}};
```

The instruction esp<2,normal,3,normal>::result instantiates the template prod for the values esp<1, normal, 3, normal>::result, safe, 3, and normal, respectively; this matches the third template of prod's definition, and a compiler error 'result' is not a member of type 'prod<1, safe, 3, normal>' is produced. If one exchanges the roles of prod's variables, the following template is written:

```
template<int Y, int flagy, int X, int flagx> class esp
  {enum {result=
  prod<X, flagx, esp<Y-1, flagy, X, flagx>::result, safe
          >::result}};
```

In this case the error occurs in the third template of sum's definition. In what follows, we will show that every partial recursive function in B can be represented by C++

templates that follow rules 1-3. If the specific error message is generated when compiling a function type F, this means that F represents a function f in a way that is not class B.

IV. Template representation of Polytime

In this section we define the Polytime language. Let normal and safe be notations for constants 0 and 1, respectively (this means, in C++ code, ♯define normal 0 and ♯define safe 1). *Binary number types* represent binary numbers, and are constructed recursively. We use the typedef typename mechanism (following [3]) instead of enumerated values; this allows us to write natural definitions of binary successors and predecessor and, in what follows, of composition and recursion on notation.

Number types representing the constant function 0 and binary successors of any number type T are in Figure 1; those representing the binary predecessor of any number type T are in Figure 2. According to these definitions, and intuitively using the composition template defined in Figure 4, the number 1101 can be represented by suc_1 <suc_0 <suc_1 < suc_1 <zero,safe>,safe>,safe>,safe>. The predecessor of any type number T is represented by pre<T, safe>::result. Each specialization has to implement the safe/normal behaviour on templates arguments (that is, on functions' variables). For example, it is mandatory in our system that the binary successors and the predecessor operate on safe arguments: thus, we add negative specializations to templates suc_0, suc_1 and pre, forcing them to produce a significant compiler error when the flag associated with the argument T is normal.

```
template<> class zero { typedef zero result;}

template<class T> class suc0 <T, safe>
      {typedef suc0 <T, safe> result;};

template<class T> class suc1 <T, safe>
      {typedef suc1 <T, safe> result;};

template<class T> class suc0 <T, normal>
      {typedef typename suc0 <T, normal>::result result;}

template<class T> class suc1 <T, normal>
      {typedef typename suc1 <X, normal>::result result;}
```

Figure 1: Templates for zero and binary successors

Templates for projection and conditional are defined in Figure 3. The first three specializations in myif definition are introduced to handle the cases in which the first argument C ends with 1 or 0, and the three arguments are safe, simultaneously. The fourth specialization returns an error when one or more arguments are normal.

The class template F that represents the *safe composition* of templates H, R and T is defined in Figure 4. Flags associated with R and T into H have values normal and safe, respectively; this implies that the value of T cannot be used by H as a principal variable of a recursion. The last specialization produces a compiler error if the variable X in R is safe, and R is used into H, simultaneously (X can be assigned with a safe value into R, harmlessly; but this cannot be done when R is substituted into a normal variable of H). This definition matches the definition of safe composition given in section II-C, and can be extended to the general case, when X and A are tuples of values, and R and T are tuples of templates.

```
template<> class pre<zero,safe> {typedef zero result;};

template<class T> class pre<suc0 <T, safe>, safe>
      {typedef T result;}

template<class T> class pre<suc1 <T, safe>, safe>
      {typedef T result;}

template<class T> class pre<suc0 <T, safe>, normal>
      {typedef typename pre<suc0 <T, safe>, normal>
                ::result result;}

template<class T> class pre<suc1 <T, safe>, normal>
      {typedef typename pre<suc1 <T, safe>, normal>
                ::result result;}
```

Figure 2: Templates for binary predecessor

```
template< class X1, int F1, ..., class Xn, int Fn > class Πj
   {typedef Xj result }

template<class C, class X, class Y> class myif
   <suc1 <typename pre<C,safe>::result,safe>, safe,
            X, safe, Y, safe>
   {typedef Y result;};

template<class C, class X, class Y> class myif
   <suc0 <typename pre<C,safe>::result, safe>, safe,
            X, safe, Y, safe>
   {typedef X result;};

template<class C, class X, class Y>
   class myif <zero, safe, X, safe, Y, safe> {typedef X result;};

template<class C, int FC, class X, int FX, class Y, int FY >
   class myif
   {typedef typename if<C, FC, X, FX, Y, FY >
            :: result result;};
```

Figure 3: Templates for projections and conditional

We introduce now an extended definition of recursion, w.r.t. the definition given in [2]. A function f is defined by *n-ple safe recursion on notation* from functions h, $g_1 \dots g_n$, $m_1 \dots m_n$ if

$$\begin{cases} f(\vec{y}, \vec{x}; \vec{a}) = g_i(\vec{x}; \vec{a}) & \text{if one of } y_i \text{ is } 0 \\ f(\vec{y}, \vec{x}; \vec{a}) = h(m(\vec{y}), \vec{x}; f(m(\vec{y}), \vec{x}; \vec{a})) & \text{otherwise} \end{cases}$$

```
template<template<class X, int F_X > class R,
        class X, int F_X, class A, int F_A > class F
        {typedef typename
        H< typename R<X, F_X >::result, normal,
             typename T<X, F_X, A, F_A >::result, safe
             >::result result; };

template<template <class X> class R, class X, int F_X,
                              class A, int F_A >
  class F <R<class X, safe>, X, F_X, A, F_A >
  {typedef typename F<R<class X, safe>, X, F_X, A, F_A >
                              ::result result; };
```

Figure 4: Templates for safe composition

where $\vec{m(y)}$ stands for the sequence $m_1(y_1), \ldots, m_n(y_n)$, and each m_i is a sequence of binary predecessors.

Similarly, the class template F that represents the *n-ple safe recursion on notations* from templates H, G_1, ..., G_n and M_1, ..., M_n is defined in Figure 5, where each M_i $(i = 1 \ldots n)$ is a sequence of predecessors applied to a binary number type which is not zero, and where we write template<X_1, F_1, ..., X_n, F_n > instead of template<class X_1, int F_1, ..., class X_n, int F_n >, for sake of simplicity. This definition can be extended to the general case, when X and A are tuples of variables.

```
template <Y_1, F_1, ..., Y_n, F_n, X, F_X, A, F_A > class F
  {typedef typename
  H<typename M_1 <Y_1 >::result, F_1,
      ...
      typename M_n <Y_n >::result, F_n,
      X, F_X, A, F_A,
      typename F<typename M_1 <Y_1 >::result, F_1,
                    ...
                    typename M_n <Y_n >::result, F_n,
                    X, F_X, A, F_A >::result,
      safe>::result result;};

template <Y_1, F_1, ..., Y_{i-1}, F_{i-1}, F_i,
              Y_{i+1}, F_{i+1}, ... Y_n, F_n, X, F_X, A, F_A >
  class F<Y_1, F_1, ..., Y_{i-1}, F_{i-1}, zero, F_i,
              Y_{i+1}, F_{i+1}, ... Y_n, F_n, X, F_X, A, F_A >
  {typedef typename G_i <X, F_X, A, F_A >::result result;};

template <Y_1, F_1, ..., Y_{i-1}, F_{i-1}, Y_i,
              Y_{i+1}, F_{i+1}, ... Y_n, F_n, X, F_X, A, F_A >
  class F<Y_1, F_1, ..., Y_{i-1}, F_{i-1}, Y_i, safe,
              Y_{i+1}, F_{i+1}, ... Y_n, F_n, X, F_X, A, F_A >
  {typedef typename F<Y_1,F_1,...,Y_{i-1},F_{i-1},Y_i,safe,
              Y_{i+1},F_{i+1},... Y_n,F_n,X,F_X,A,F_A >::result result;};
```

Figure 5: Templates for n-ple safe recursion

We set to safe the value of the flag associated with the recursive call of F into H (rule 2, Section III); we specialize F to compute the base cases of the recursion (where one of the templates G_i has to be computed); and we introduce the last n templates because the programmer is not allowed to assign a recursive call to one of the principal variables Y_1, ..., Y_n (rule 3).

We define the language *Poly-Temp* as the smallest class of templates containing zero, suc_0, suc_1, pre, myif, Π_j and closed under safe composition and n-ple safe recursion on notations. The polynomial-time functions will be represented exactly by those templates in *Poly-Temp* with all normal flags.

V. *Poly-Temp* CAPTURES POLYTIME

In this section, we state that every function computable within polynomial time by a Turing machine can be expressed in *Poly-Temp*; in order to do this, we recall that Polytime is captured by class B [2], and we prove that B is represented by templates in *Poly-Temp* (Theorem 5.1). Conversely, we show that any template in *Poly-Temp* is polynomial-space bounded (Theorem 5.2) and hence polynomial-time bounded (Theorem 5.3).

Theorem 5.1: For each function f in B, there exists a C++ template program F such that F computes f (at compile time).

Proof: (by induction on the construction of f). We denote binary number types with the capital letters X, Y, A, C, and the related flags with F_X, F_Y, F_A, F_C; we write (1) template<X, F_X, Y, F_Y > instead of template<class X, int F_X, class Y, int F_Y >; and (2) p<X> instead of typename pre<X,safe>::result, for sake of simplicity.

Base. Templates defined in section IV (constant, binary successors, predecessor, conditional and projections) trivially compute the basic functions of B.

Step. Case 1. Let f be defined by safe recursion on notations from functions $g(x; a)$, $h_0(y, x; a, s)$ and $h_1(y, x; a, s)$, that are computed, by the inductive hypotheses, by templates G, H_0 and H_1, respectively. f is represented in *Poly-Temp* by the following template F:

```
template <Y, F_Y, X, F_X, A, F_A > class F
{typedef typename myif<Y, safe
    typename H_0 <p<Y>, F_Y, X, F_X, A, F_A,
                 typename
                 F<p<Y>, F_Y, X, F_X, A, F_A >::result,
                 safe>::result, safe
    typename H_1 <p<Y>, F_Y, X, F_X, A, F_A,
                 typename
                 F<p<Y>, F_Y, X, F_X, A, F_A >::result,
                 safe>::result, safe
                     >::result result };

template <F_Y, X, F_X, A, F_A >
     class F<zero, F_Y, X, F_X, A, F_A >
     {typedef typename G<X, F_X, A, F_A >::result result};
```

```
template <Y, X, F_X, A, F_A >
    class F<Y, safe, X, F_X, A, F_A >
    {typedef typename F<Y, safe, X, F_X, A, F_A >
                                ::result result};
```

F is obtained by n-ple safe recursion (Figure 5) and safe composition (Figure 4) from templates if, H_0 and H_1.

Case 2. Let f be defined by safe composition from functions $h(p;q)$, $r(x;)$ and $t(x;a)$, that are computed, by the inductive hypotheses, by templates H, R and T, respectively. The template F computing f is defined in Figure 4. ■

To prove that any template in our language is polynomial-time bounded, we find a polynomial-space bound for the length of any template belonging to *Poly-Temp*. For sake of brevity, we omit the flags and we write safe inputs to the right of a semicolon, and normal ones to the left, following the notation used by Bellantoni and Cook. We also use "$(\ldots)$" instead of "$<\ldots>$". This implies that if a template F is defined by n-ple safe recursion from templates H, G_1, …, G_n and M_1, …, M_n, we write

$$\begin{aligned} F(\overline{Y},\overline{X};\overline{A}) &= G_i(\overline{X};\overline{A}) \quad \text{if one of } Y_i \text{ is zero} \\ &= H(\overline{M(Y)},\overline{X};\overline{A},F(\overline{M(Y)},\overline{X};\overline{A})) \quad \text{otherwise} \end{aligned}$$

where $\overline{M(Y)}$ stands for $M_1(Y_1),\ldots,M_n(Y_n)$, and each M_i is a sequence of binary predecessors.

Theorem 5.2: For each template F in *Poly-Temp*, there exists a polynomial q_F such that

$$|F(\overline{X};\overline{A})| \le q_F(\overline{|X|}) + \max_i |A_i|$$

where $\overline{X}$ and $\overline{A}$ are the variables labelled with normal and safe, respectively, and $q_F(\overline{|X|})$ stands for $q_F(|X_1|,\ldots,|X_n|)$.

Proof: (by induction on the construction of F).

Base. If F is a constant, binary successors, predecessor, conditional or projection template, then we have $|F(\overline{X};\overline{A})| \le 1 + \sum_i |X_i| + \max_i |A_i|$.

Step. Case 1. If F is defined by n-ple safe recursion we have, by induction hypotheses, the polynomials $q_{G_1},\ldots,q_{G_n}$ and q_H bounding $G_1,\ldots,G_n$ and H, respectively; that is,

$$|F(\ldots,\text{zero},\ldots,\overline{X};\overline{A})| \le q_{G_j}(\overline{|X|}) + \max_i |A_i|, \text{ and}$$
$$|F(\overline{Y},\overline{X};\overline{A})| \le q_H(\overline{|M(Y)|},\overline{|X|}) + \max(\max_i |A_i|, |F(\overline{M(Y)},\overline{X};\overline{A})|).$$

Define q_F such that

$$q_F(\overline{|Y|},\overline{|X|}) = \overline{|Y|}\cdot q_H(\overline{|Y|},\overline{|X|}) + \textstyle\sum_j q_{G_j}(\overline{|X|}).$$

We have that $|F(\ldots,\text{zero},\ldots,\overline{X};\overline{A})| \le q_F(\overline{|\text{zero}|},\overline{|X|}) + \max_i(A_i)$. We also have

$$\begin{aligned} |F(\overline{Y},\overline{X};\overline{A})| &= |H(\overline{M(Y)},\overline{X};\overline{A},F(\overline{M(Y)},\overline{X};\overline{A}))| \\ &\le q_H(\overline{|M(Y)|},\overline{|X|}) + \\ &\qquad \max(\max_i |A_i|, |F(\overline{M(Y)},\overline{X};\overline{A})|) \\ &\le q_H(\overline{|M(Y)|},\overline{|X|}) + \\ &\qquad \max(\max_i |A_i|, q_F(\overline{|M(Y)|},\overline{|X|}) + \max_i |A_i|) \\ &\le q_H(\overline{|M(Y)|},\overline{|X|}) + \\ &\qquad q_F(\overline{|M(Y)|},\overline{|X|}) + \max_i |A_i| \\ &\le q_H(\overline{|M(Y)|},\overline{|X|}) + \\ &\qquad \overline{|M(Y)|}\cdot q_H(\overline{|M(Y)|},\overline{|X|}) + \textstyle\sum_j q_{G_j}(\overline{|X|}) + \\ &\qquad + \max_i |A_i| \\ &\le (\overline{|M(Y)|}+1)\cdot q_H(\overline{|M(Y)|},\overline{|X|}) + \\ &\qquad \textstyle\sum_j q_{G_j}(\overline{|X|}) + \max_i |A_i| \\ &\le \overline{|Y|}\cdot q_H(\overline{|M(Y)|},\overline{|X|}) + \\ &\qquad \textstyle\sum_j q_{G_j}(\overline{|X|}) + \max_i |A_i| \\ &\le \overline{|Y|}\cdot q_H(\overline{|Y|},\overline{|X|}) + \textstyle\sum_j q_{G_j}(\overline{|X|}) + \max_i |A_i| \\ &\le q_F(\overline{|Y|},\overline{|X|}) + \max_i |A_i| \end{aligned}$$

Case 2. If f is defined by safe composition we have, by induction hypotheses, q_H, q_R and q_T bounding H, R and T, respectively; we have

$$\begin{aligned} |F(\overline{X};\overline{Y})| &= |H(R(\overline{X};);T(\overline{X};\overline{Y}))| \\ &\le q_H(|R(\overline{X};)|) + |T(\overline{X};\overline{Y})| \\ &\le q_H(q_R(\overline{|X|})) + |T(\overline{X};\overline{Y})| \\ &\le q_H(q_R(\overline{|X|})) + q_T(\overline{|X|}) + \max_j |Y_j| \end{aligned}$$

Let $q_F(\overline{|X|},\overline{|Y|})$ be $q_H(q_R(\overline{|X|}))+q_T(\overline{|X|})$. We have the result. ■

Note that templates in *Poly-Temp* are polynomially time-bounded too, when evaluated. Indeed, base templates (zero, Π_j, suc_0, suc_1, if) are bounded by the length of their arguments; for composition templates, observe that the composition of two polynomial-time templates is still a polynomial time template; for recursion templates, it is well known that recursion on notation can be executed in polynomial time if the result of the recursion is polynomially length-bounded and the step and base functions are polytime, as in our case. Thus, we have

Theorem 5.3: Each template F in *Poly-Temp* is evaluated in polynomial time.

VI. Conclusions and further work

In summary, we have defined a restricted metalanguage by means of C++ templates, and we have shown that it captures at compile time the set of polynomial-time computable functions. As we mentioned in the Introduction, a contribution of this result is that it could provide the theoretical base for the construction of tools for the formal certification of upper bounds for metaprogramming time consumption. As an anonymous referee says, "normal" meta-programs do not follow the restriction imposed by our metalanguage; thus, a sensible prosecution of this work could be the analysis of transformation methods from "normal" metaprograms to "restricted" ones. Nevertheless, even if our template language is admissible C++, there is no doubt that programming in it should be hard, due to the extra annotations encoded as templates parameters; one may think to hide them into *traits* [18] containing representation of numbers and of related flags; in this way we'd be able to obtain a neater language. However, obscure error messages from C++ compilers could inhibit this as a workable approach. The programmer is not able to understand why and where in the program he used a recursive variable in the wrong way; *static interfaces* techniques [17] could help us to provide a clearer meaning to error messages.

Even if this is a clumsy characterization of a complexity class, it is worth noting that the three rules introduced above can produce polynomial time bounded templates when applied to all kinds of recursions, not only to primitive recursion; it seems that our approach improves the understanding of polynomial-time computation's nature, allowing us to use more expressive recursive schemes.

References

[1] D. Abrahams and A. Gurtovoy, *C++ template metaprogramming: concepts, tools and techniques from Boost and beyond*, Addison Wesley, 2004.

[2] S. Bellantoni and S. Cook, “A new recursion-theoretic characterization of the poly-time functions”, *Computational Complexity*, vol. 2, pp. 97–110, 1992.

[3] M. Böhme and B. Manthey, “The computational power of compiling C++”, *Bull. of the EATCS*, vol. 81, pp. 264–270, 2003.

[4] A. Cobham, “The intrinsic computational difficulty of functions”, in *Y. Bar-Hillel (ed), Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science, North-Holland, Amsterdam*, 1962, pp. 24–30.

[5] L. Colson, “ About primitive recursive algorithms”, *Theoretical Computer Science*, vol. 83, pp. 57–69, 1991.

[6] L. Colson and D. Fredholm, “ System T, call by value and the minimum problem”, *Theoretical Computer Science*, vol. 206, pp. 301–315, 1998.

[7] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[8] M. Hofmann, “ Linear type and non-size-increasing polynomial time computation”, in *Proceedings of the 14th IEEE Symposium on Logic in Computer Science*, 1999, pp. 464–473.

[9] J. Järvi, “ Compile Time Recursive Objects in C++”, in *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS '98), Washington DC, USA*, 1998.

[10] J. Järvi, G. Powell, and A. Lumsdaine, “ The Lambda library: unnamed functions in C++”, *Softw. Pract. Exper.*, vol. 33, pp. 259–291, 2003.

[11] N.D. Jones, “ LOGSPACE and PTIME characterized by programming languages”, *Theoretical Computer Science*, vol. 228, pp. 151–174, 1999.

[12] K. Läufer, “ A Framework for Higher-Order Functions in C++”, in *Proceedings of the Conf. on Object-Oriented Technologies (COOTS), Monterey, CA*, 1995.

[13] D. Leivant, ”Ramified recurrence and computational complexity I: word recurrence and polytime”, in P. Clote, J. Remmel (eds), Feasible Mathematics II, Birkauser, 1994.

[14] D. Leivant and J.-Y. Marion, ”Ramified recurrence and computational complexity II: substitution and polyspace”, in J. Tiuryn, L. Pocholsky (eds), Computer Science Logic, LNCS, vol. 933, pp. 486–500, 1995.

[15] B. McNamara and Y. Smaragdakis, “ Functional programming in C++”, in *Proceedings of the 5th ACM SIGPLAN International conference on Functional programming, ICFP '00, New York NY, USA*, 2000, pp. 118–129.

[16] B. McNamara and Y. Smaragdakis, “Functional programming with the FC++ library”, *J. Funct. Program.*, vol. 14(4), pp. 429–472, 2004.

[17] B. McNamara and Y. Smaragdakis, “Static Interfaces in C++”, *First Workshop on C++ Template Programming, Erfurt, Germany*, 2000.

[18] N. Myers, “ A new and useful template technique: "Traits" ”, *C++ Report*, vol. 7(5), pp. 32–35, 1995.

[19] T. Sheard and S. Peyton Jones, “ Template meta-programming for Haskell”, in *Proceedings of the 2002 Haskell Workshop (Haskell '02), Pittsburgh, PA*, 2002, pp. 1–16.

[20] J. Striegnitz, “ The FACT! library homepage”, 2000. Available (September 2011): http://www.fz-juelich.de/zam/FACT

[21] B. Stroustrup, *The C++ programming language*, Addison-Wesley, 1997.

[22] E. Unruh, “Prime number computation, ANSI X3J16-94-0075/ISO WG21-462”.

[23] E. Unruh, “ Template metaprogrammierung”, 2002. Available (September 2011): http://www.erwin-unruh.de/ meta.html

[24] D. Vandevoorde and N.M. Josuttis, *C++ templates: the complete guide*, Addison-Wesley, 2003.

[25] T. Veldhuizen, “ Using C++ templates metaprograms”, *C++ Report*, vol. 7(4), pp. 36–43,1995 .

[26] T. Veldhuizen, “ C++ templates as partial evaluation”, in *Proceedings of the 1999 ACM SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation*, 1999, pp. 13–18.

[27] T. Veldhuizen, “ C++ templates are Turing complete”, *unpublished.*

[28] T. Veldhuizen, “ Tradeoffs in metaprogramming”, in *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation*, 2006, pp. 150–159.