Journal of Universal Computer Science, vol. 13, no. 4 (2007), 468-478
submitted: 5/1/07, accepted: 26/4/07, appeared: 28/4/07 © J.UCS

Compile-time Computation of Polytime Functions

Emanuele Covino, Giovanni Pani, and Daniele Scrimieri
(Dipartimento di Informatica, Università di Bari, Italy
{covino,pani,scrimieri}@di.uniba.it)

Abstract: We investigate the computational power of C++ compilers. In particular, it is known that any partial recursive function can be computed at compile time, using the template mechanism to define primitive recursion, composition, and minimalization. We show that polynomial time computable functions can be computed at compile-time using the same mechanism, together with template specialization.

Key Words: Static computation, C++ templates, polytime computable functions.
Category: D.3, F.1.3

1 Introduction

C++ templates were designed to provide generic programming, but they are also capable of performing static computation. In [Veldhuizen, 1999] this ability is described: C++ may be regarded as a 2-level language, in which types are first-class values, and template instantiation mimic off-line partial evaluation (see [Jones, 1996]). The first example of this behaviour was reported in [Unruh, 2001] and [Unruh, 2002], where a program that produces (as error messages) a list of prime numbers was written. Another example is the following:

```
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. This happens because the operator A::B refers to the symbol B in the scope of A; when reading the command pow<3,5>::result, the compiler has to instantiate 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.

In this example some issues should be pointed out: first, the partial specialization of templates, that allows the compiler to compute the base case of the recursive definitions; then, the instruction enum{result=X∗pow<Y-1,X-1>::result;}, that represents the step of the recursive evaluation of pow, and produces the intermediate values. This computation happens at compile time, since enumeration values aren't l-values (that is, they don't have an address); thus, when one pass them to the recursive call of a template, no static memory is used (see [Vandevoorde and Josuttis, 2003], chapter 17). In [Kristiansen and Voda, 2003], some well-known complexity classes have been characterized by means of fragments of C (among them, Polytime), although no distinctive features of the language have been used.

In this paper we show that polynomial-time computable functions can be computed at compile time by using the template mechanism introduced above; to do this, we extend the approach presented in [Böhme and Manthey, 2003] (see next section for a brief description of their result), and we apply it to Bellantoni and Cook's characterization of Polytime, sketched in section 3. Some examples and the proof of our result are in sections 4 and 5.

2 The computational power of C++ compilers

In [Böhme and Manthey, 2003] a way to specify primitive recursion, composition, and μ-recursion by means of the use of C++ template mechanism and type system is presented. The result is interesting: *any partial recursive function can be computed at compile-time*, that is by running a C++ compiler, and returning an error message that contains the result of the function in unary. This result was in some sense anticipated by [Unruh, 2001] and [Unruh, 2002], and by [Veldhuizen, 1995] and [Veldhuizen, 1999]. In this section, we summarize Böhme and Manthey's result.

Number types are constructed recursively and they 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;}`. The number 2 is represented by `suc<suc<zero>>`, and `T::pre` represents the predecessor of any type number `T` which is not `zero`. A function is represented by a C++ class template, in which templates arguments are the arguments of the function.

To express the function type F of a function f defined by primitive recursion from g and h the following template is written (G and H compute g and h):

```
template <class Y, class X_1, ... ,class X_n > struct F
  {typedef typename H<
                    typename Y::pre,
                    X_1,...,X_n,
                    typename F<typename Y::pre, X_1, ... ,X_n >::val
                    >::val val;};

template <class X_1, ... ,class X_n> struct F<zero, X_1,...,X_n >
  {typedef typename G<X_1,...,X_n>::val val; };
```

In a similar manner composition and μ-recursion are represented; thus, the whole class of partial recursive functions can be expressed. Note that the template pow in the Introduction can be written according to Böhme and Manthey definition. Enumeration values and typedef typename mechanism are equivalent, and they both don't assign memory to the recursive calls. Thus, we can write

```
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};
```

where times is an already defined template which computes the product between its two arguments.

3 A recursive-theoretic characterization of Polytime computable functions

In [Bellantoni and Cook, 1992] the class of polynomial time computable functions is characterized as the smallest class of functions containing some initial functions, and closed under *safe* recursion on notation and *safe* composition. This is obtained by imposing a syntactic restriction on variables used in the recursion; they are distinguished in normal or safe, and the latter cannot be used as the principal variable of a function defined 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 $f(\mathbf{x};\mathbf{y})$, where $\mathbf{x}$ is the usual notation for $x_1,\ldots,x_n$; in this case, variables x_i are normal, whereas variables y_j are safe. Following [Bellantoni and Cook, 1992], 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. **Successor:** $s_i(;a) = 2a+i = ai$, for $i \in \{0,1\}$.
4. **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

$$\begin{cases} f(0, \mathbf{x}; \mathbf{a}) = & g(\mathbf{x}; \mathbf{a}) \\ f(yi, \mathbf{x}; \mathbf{a}) = & h_i(y, \mathbf{x}; \mathbf{a}, f(y, \mathbf{x}; \mathbf{a})). \end{cases}$$

   for $i \in \{0, 1\}$, g and h_i in B.

7. **Safe composition:** the function f is defined by safe composition from functions h, $\mathbf{r}$ and $\mathbf{t}$ if $f(\mathbf{x}; \mathbf{a}) = h(\mathbf{r}(\mathbf{x};); \mathbf{t}(\mathbf{x}; \mathbf{a}))$ for h, $\mathbf{r}$ and $\mathbf{t}$ in B.

When defining a function $f(yi, \mathbf{x}; \mathbf{a})$ by recursion from g and h_i, the value $f(y, \mathbf{x}; \mathbf{a})$ is in a safe position of h_i; 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 (via mutual simulation) to the class of polynomial-time computable functions.

4 Some polytime computable functions represented by C++ templates

In this section we introduce a mechanism which allows us to represent polytime functions in B by means of template definitions, using the mechanism introduced in [Böhme and Manthey, 2003], together with a restriction on the role of the template arguments. We analyze the recursive definition of sum, product, and exponent functions, showing how the latter cannot be computed when we force it into our restriction. We recall that functions sum and product can be expressed by safe recursion as follows:

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

Note that, in the previous definition of $\otimes(b+1, a;)$, 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 definitions 6 and 7, previous section); this implies that one cannot use it as the principal variable of a recursion. In other words, principal variables cannot be substituted with being-computed values (recursive calls), but only with totally computed values (numbers). Hence, the following definition of $\otimes$ is not allowed.

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

For the same reason exponentiation (in both following definitions) cannot be defined in B.

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

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

1. each flag is equal to `normal`;
2. flags associated to variables assigned with recursive calls are switched to `safe`;
3. a compiler error must be generated whenever a variable labelled with a `safe` flag is moved into the 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 number types):

```
♯define normal 0; ♯define safe 1;

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

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

template<int Y, int X, int flagx> class sum<Y, safe, X flagx>
  {public: 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 value of 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 <arg's> class bottom <spec-arg's>
  {public: enum {result= bottom <spec-arg's>::result };};
```

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

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

template<int Y, int X, int flagx> class prod<Y, safe, X, flagx>
  {public: 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 compiler 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 the template for 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
  {public: 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 (this is correct, since we tried to assign the recursive call of prod to the principal variable of sum).

We define now the C++ template representation of the exponent function.

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

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

In this case 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 same phenomenon occurs.

Rules 1-3 show us how to define C++ templates for any partial recursive function. If an error message is generated when compiling a function type F, this means that F represents a function f in a way that is not in Bellantoni and Cook's class B. The related theorem and proof are in the following section.

5 Template representation of Polytime

In this section we prove that every function in B can be expressed by C++ templates (with the restrictions on template arguments introduced above). Functions in B have binary numbers as input; thus, we need to introduce a new definition of *binary number types* that represent binary numbers, and that are constructed recursively. The number type representing 0 is class zero { }.

Given a number type T, the number type representing its s_0 successor is template<class T> class suc_0 {typedef T pre;}. Number type representing its s_1 successor is template<class T> class suc_1 {typedef T pre;}.
T::pre represents the predecessor of any type number T which is not zero; for example, number 1101 is represented by suc_1 <suc_0 <suc_1 <suc_1 <zero>>>>; and suc_0 <suc_1 <suc_1 <zero>>>::pre stands for 110. We use the typedef typename mechanism (following [Böhme and Manthey, 2003]) instead of enumerated values because this allows us to write natural definitions of binary successors and predecessor, and of recursion on notation.

Theorem 1. *For each function f in B, there exists a C++ template program P_f such that P_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 two-value flags (normal or safe) with F_X, F_Y, F_A, F_C; we write template<X_1, F_1, ..., X_n, F_n > instead of the sequence of template arguments (and flags) template<class X_1, int F_1, ..., class X_n, int F_n >, when needed.

Base. Templates and template specializations for constant, projection, successor and predecessor are defined as follows (definitions 1-4, section 3):

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

template<X$_1$, F$_1$, ..., X$_n$, F$_n$, X$_{n+1}$, F$_{n+1}$, ..., X$_{n+m}$, F$_{n+m}$ > class Π_j
{ typedef X$_j$ result };

template<X, F$_X$ > class suc$_0$ {typedef X pre;};
template<X> class suc$_0$ <X, normal>
{typedef typename suc$_0$ <X, normal>::pre pre;};

template<X, F$_X$ > class suc$_1$ {typedef X pre;};
template<X> class suc$_1$ <X, normal>
{typedef typename suc$_1$ <X, normal>::pre pre;};

Note that each specialization imitates the safe/normal behaviour of variables. For example, it is mandatory that the s_0 successor operates on a safe argument: thus, we specialize (in a negative way) the related template suc$_0$ by setting to normal the flag associated with number type X, and by producing, in this case, a compiler error. This implies a slight change in the definition of binary number types given above: each number type has a flag attached, whose value is always safe. Templates representing the conditional function are defined as follows (definition 5, section 3):

template<C,X,Y> class myif<suc$_1$ <typename C::pre>, safe, X, safe, Y, safe>
{typedef Y result;};
template<C,X,Y> class myif<suc$_0$ <typename C::pre>, safe, X, safe, Y, safe>
{typedef X result;};
template<C,X,Y> class myif<zero, safe, X, safe, Y, safe> {typedef X result;};
template<C, F$_C$, X, F$_X$, Y, F$_Y$ > class myif
{typedef typename myif<C, F$_C$, X, F$_X$, Y, F$_Y$ >:: result result;};

The first three specializations in the definition of myif are introduced to handle the cases in which the first argument ends with 1 or 0, and the three arguments are safe, simultaneously. The fourth specialization returns an error, meaning that at least one of the arguments is normal.

Step. Case 1. f is defined by safe recursion on notation 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 by the following templates:

template <class Y, int F$_Y$, class X, int F$_X$,class A, int F$_A$ >
class F<suc$_0$ <typename Y::pre>, int F$_Y$, class X, int F$_X$,class A, int F$_A$ >
{typedef typename H$_0$ <typename Y::pre, F$_Y$,
X, F$_X$,
A, F$_A$,
typename F<typename Y::pre, F$_Y$,

```
                                    X, F_X,
                                    A, F_A >::result,
                         safe>::result result };

template <class Y, int F_Y, class X, int F_X,class A, int F_A >
   class F<suc_1 <typename Y::pre>, int F_Y, class X, int F_X,class A, int F_A >
   {typedef typename H_1 <typename Y::pre, F_Y,
                         X, F_X,
                         A, F_A,
                         typename F<typename Y::pre, F_Y,
                                    X, F_X,
                                    A, F_A >::result,
                         safe>::result result };

template <int F_Y, class X, int F_X, class A, int 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 <class Y, class X, int F_X, class A, int 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};
```

This definition can be extended to the general case, when x and a are tuples of variables. Following rules 1-3 in section 4, we set to safe the values of those flags associated with the recursive call of F into H_0 and H_1; and we specialize F to compute the base case of the recursion. The last template is introduced in order to prevent the programmer from assigning a recursive call to the principal variable Y. Note how F is specialized on values suc_0 <typename Y::pre> and suc_1 <typename Y::pre> in order to recursively instantiates H_0 or H_1, according to the value of the last digit of Y (recursion on notation).

Case 2. f is 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. f is represented by the following templates:

```
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 };

Flags associated with R and T into H are switched to normal and safe, respectively; in this case, 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 of R is safe, and R is used into H, simultaneously (R is defined elsewhere, and a safe value can be assigned to some of its variables, harmlessly; but this cannot happen when R is substituted into a normal variable of H). This proof can be extended to the general case, when x and a are tuples of values, and r and t are tuples of functions.

6 Conclusions

Templates written according to our restrictions to C++ are polynomially time-bounded, when evaluated. Base templates (zero, Π_j, suc_0, suc_1, myif) 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 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). The reader can refer to [Bellantoni and Cook, 1992], section 4, for a similar proof.

It is interesting to note that if an algorithm (described following Bellantoni and Cook's approach) computes a non-trivial binary function, then its time-complexity is at least linear in one of the inputs (see [Colson and Fredholm, 1998] for the proof of this result). For example, consider the algorithm for computing the minimum of two natural numbers written in unary:

$\min(0, y) = 0$; $\min(s(x), 0) = 0$; $\min(s(x), s(y)) = s(\min(x, y))$.

The natural computation time of such an algorithm is $O(\min(x, y))$, but no primitive recursive function definable in [Bellantoni and Cook, 1992] can respect this bound. Consider now the following templates, written according to our rules:

template <class X, int F_X, class Y, int F_Y > class min
{typedef typename suc<typename min<typename X::pre, F_X,
typename Y::pre, F_Y >::result>::result
result};

template <int F_X, class Y, int F_Y > class min<zero, F_X, Y, F_Y >
{typedef zero result};

template <class X, int F_X, int F_Y >
    class min<suc<typename X::pre>, F_X, zero, F_Y > {typedef zero result};

template <class X, int F_X, class Y, int F_Y > class min<X, safe, Y, F_Y >
    {typedef typename min<X, safe, Y, F_Y >::result result};

template <class X, int F_X, class Y, int F_Y > class min<X, F_X, Y, safe>
    {typedef typename min<X, F_X, Y, safe>::result result};

Previous templates computes the function min within time $O(\min(x, y))$; this result is achieved by the introduction of simultaneous recursion, plus our safe/normal mechanism (last two templates). Thus, our system seems to be able to express more intensions (that is, more algorithms) than the Bellantoni and Cook's approach.

References

[Bellantoni and Cook, 1992] Bellantoni, S., Cook, S.: "A new recursion-theoretic characterization of the poly-time functions"; Computational Complexity 2(1992)97-110.

[Colson and Fredholm, 1998] Colson, L., Fredholm, D.: "System T, call by value and the minimum problem"; Theoretical Computer Science 206(1998)301-315.

[Böhme and Manthey, 2003] Böhme, M., Manthey, B.: "The computational power of compiling C++"; Bull. of the EATCS, 81, October 2003, 264-270.

[Jones, 1996] Jones, N. D.: "An introduction to partial evaluation"; ACM Computing Surveys 28, 3 (September 1996), 480-503.

[Kristiansen and Voda, 2003] Kristiansen, L., Voda, P. J.: "Complexity classes and fragments of C"; Information Processing Letters 88 (2003), 213-218.

[Myers, 1995] Myers, N.: "A new and useful template technique: "Traits" "; C++ Report 7,5 (June 1995), 32-35.

[Stroustrup, 1997] Stroustrup, B.: "The C++ programming language"; Addison Wesley, 1997.

[Unruh, 2001] Unruh, E.: "Prime number computation"; ANSI X3J16-94-0075/ISO WG21-462.

[Unruh, 2002] Unruh, E.: "Template metaprogrammierung"; http://www.erwin-unruh.de/ meta.html, 2002.

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

[Veldhuizen, 1995] Veldhuizen, T.: "Using C++ templates metaprograms"; C++ Report, 7,4 (1995), 36-43.

[Veldhuizen, 1999] Veldhuizen, T.: "C++ templates as partial evaluation"; Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Semantic-Based Program Manipulation (PEPM), Technical report BRICS NS-99-1, University of Aarhus, 13-18, 1999.