The Computational Power of Compiling C++

Martin Böhme* Bodo Manthey†

Abstract

Using a C++ compiler, any partial recursive function can be computed *at compile time*. We show this by using the C++ template mechanism to define functions via primitive recursion, composition, and μ-recursion.

1 Introduction

Any partial recursive function can be computed using the programming language C++. But what about the computational power of C++ compilers? Are there any functions that can be computed just by compiling C++ source code instead of executing a C++ program after compilation? The answer is surprising: Any partial recursive function can be computed by running a C++ compiler. We show this by presenting a way to specify primitive recursion, composition, and μ-recursion [2] by (ab)using the C++ template mechanism and type system [3]. Our strategy for obtaining the output is to provide an error message that contains the result in unary representation. When we run the C++ compiler, after specifying a partial recursive function f and a natural number x as C++ source code, the error message typed out by the compiler will be $f(x)$. We make the reasonable assumption that the compiler outputs helpful error messages that give the names of the types involved in a type conflict, for example. However, even if the only error message that is produced by the compiler is "error", we show that the compiler still has to compute $f(x)$ internally.

The idea of using the C++ template mechanism for computations at compile time goes back to Unruh. He developed a program that printed out prime numbers at compile time [4], and stated that any partial recursive function could be computed in this way [5]. As far as we are aware, though, he has never published a proof. Veldhuizen [6] picked up the idea and applied it to improve the speed of C++ programs. He considers C++ to be a two-level language: static computations performed at compile time and dynamic computations performed at run time [7]. Splitting the computation up in such a way is called "partial evaluation" or "program specialization" (see e.g. Jones [1] or Stroustrup [3, Sec. 13.5]), and today this technique is widely used in template libraries.

*Universität zu Lübeck, boehme@informatik.uni-luebeck.de.

†Universität zu Lübeck, Institut für Theoretische Informatik, manthey@tcs.uni-luebeck.de.

2 Defining Functions Using C++ Templates

We now present the mechanism by which we compute partial recursive functions using the template mechanism. To represent numbers, we choose not to use the built-in integer type `int`; instead, we use types constructed recursively using the C++ template mechanism, which in theory allows us to represent arbitrarily large numbers. We will call a C++ type that represents a natural number a *number type*.

To make this concrete,

```
struct zero { };
```

is the number type that represents the number zero, and, given a number type `T`,

```
template<class T> struct suc
{
   typedef T pre;
};
```

represents the successor of that number. For example, `suc<suc<zero> >` represents the number 2. The `pre` typedef can be used to obtain the predecessor. Thus, for any number type `T` that is not `zero`, `T::pre` represents the predecessor of that number. (For brevity, we will use `struct` instead of `class` throughout this work; both are equivalent except that the default access for `struct` is `public` whereas for `class` it is `private`.)

A function is represented as a C++ class that contains a typedef called `val`. This typedef is equal to the number type of the result. The arguments of the function are represented as template arguments. We will refer to classes that represent functions in this way as *function types*. For example,

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

is a function type that computes the function $f(x) = x + 2$. If we want to compute $f(1)$, we add

```
int main()
{
   plus2<suc<zero > >::val tmp;
   return (int) tmp;
}
```

as the main program. Due to the illegal type cast `(int) tmp;` the compiler will type out the error message

```
‘struct suc<suc<suc<zero> > >’ used where a ‘int’ was expected
```

and thus we obtain the result $f(1) = 3$.

To show that any partial recursive function can be computed in this way, we need to be able to express the base functions as well as primitive recursion, composition and μ-recursion. We quickly review their definitions (see e.g. Smith [2]).

Base Functions: The zero function Z with $Z(x) = 0$, the successor function S with $S(x) = x+1$ and the projection functions U_j^n ($1 \leq j \leq n$) with $U_j^n(x_1, \ldots, x_n) = x_j$ are primitive recursive.

Primitive Recursion: Let g and h be primitive recursive functions of arity n and $n + 2$, respectively. Then the function f with

$$\begin{array}{lcl} f(x_1, \ldots, x_n, 0) & = & g(x_1, \ldots, x_n) \text{ and} \\ f(x_1, \ldots, x_n, y + 1) & = & h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)) \end{array}$$

is also primitive recursive.

Composition: Let g_1, ..., g_m be primitive recursive functions, each of arity n, and h be a primitive recursive function of arity m. Then the function f with

$$f(x_1, \ldots, x_n) \quad = \quad h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)) \, .$$

is also primitive recursive.

μ-Recursion: Let h be a (partial) function of arity $n + 1$. A function f of arity n is defined by μ-recursion from h if

$$f(x_1, \ldots, x_n) = \mu\, y[h(x_1, \ldots, x_n, y) = 0] \, ,$$

where $\mu\, y[h(x_1, \ldots, x_n, y) = 0] = z$ if

- $h(x_1, \ldots, x_n, z) = 0$ and
- for all $y' < z$, $h(x_1, \ldots, x_n, y')$ is defined and $h(x_1, \ldots, x_n, y') \neq 0$.

If no such z exists, $\mu\, y[h(x_1, \ldots, x_n, y) = 0]$ is undefined.

We first demonstrate that primitive recursion can be expressed. We demonstrate this only for the case where f is a binary function, but the extension to the general case is easy. Therefore, let g and h be unary and ternary primitive recursive functions, respectively, and let `G` and `H` be function types that compute g and h. Then the following function type `F` computes the function f as defined by primitive recursion from g and h.

```
template<class X, class Y> struct F;

template<class X> struct F<X, zero>
{
   typedef typename G<X>::val val;
};

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

We omit a demonstration of how the base functions and composition can be expressed since the base functions are fairly easy and composition merely involves assembling the various function types.

With the base functions, primitive recursion and composition, any primitive recursive function can be expressed as a function type. We will now show that μ-recursion can be expressed, thus allowing us to compute any partial recursive function, since any partial recursive function can be expressed with a single μ-operator acting on a primitive recursive function.

In our demonstration of how to express the μ-operator, we again restrict ourselves to unary functions, but again, generalization is easy. Let f be a unary partial recursive function, let h be a binary primitive recursive function such that $f(x) = \mu\, y[h(x, y) = 0]$, and let `H` be a function type that computes h. The idea for computing $\mu\, y[h(x, y) = 0]$ using the template mechanism is to construct a function

$$\mathrm{mu}(h, x, y, p) = \begin{cases} y - 1 & \text{if } p = 0 \text{ and} \\ \mathrm{mu}(h, x, y + 1, h(x, y)) & \text{otherwise.} \end{cases}$$

We have $f(x) = \mu\, y[h(x, y) = 0] = \mathrm{mu}(h, x, 0, 1)$. The way this works is that we always have $p = h(x, y - 1)$ (except for $y = 0$), and so when p is zero for the first time we return $y - 1$ as the result. If $h(x, y) \neq 0$ for all y then the recursion never terminates, and so $\mathrm{mu}(h, x, 0, 1)$ is undefined.

We now define a class `Mu<H, X, Y, HprY>` that computes mu.

```
template<template<class A,class B> class H,
         class X, class Y, class HprY> struct Mu;

template<template<class A,class B> class H,
         class X, class Y> struct Mu<H,X,Y,zero>
{
   typedef typename Y::pre min;
};

template<template<class A,class B> class H,
         class X, class Y, class HprY> struct Mu
{
   typedef typename Mu<
      H, X, suc<Y>, typename H<X,Y>::val>::min min;
};
```

This is a straightforward implementation of the definition of mu. Template specialization is used to select the first definition of `Mu` when `HprY` $\equiv$ `zero`, and the second definition otherwise.

This means that if $f(x) = \mu\, y[h(x, y) = 0] = \mathrm{mu}(h, x, 0, 1)$ is defined, `Mu<H, X, zero, suc<zero> >::min` is the number type that represents $f(x)$. If $f(x)$ is undefined, the type `Mu<H, X, zero, suc<zero> >::min` is likewise undefined; specifically, it is an infinite nesting of template instantiations, which will cause the compiler to go into an infinite loop (or hit an internal limit on the template instantiation depth).

3 Concluding Remarks

We have seen how to compute any partial recursive function f using the C++ template mechanism and type system, outputting the result as an error message from the compiler. But what happens if the compiler does not print out any (helpful) error messages?

In this case, the compiler still has to compute the value $f(x)$ internally. Suppose that `suc<T>` is given the member variables `T dummy1` and `int dummy2`. This means that we have `sizeof(suc<T>)` $>$ `sizeof(T)`, because `suc<T>` contains an object of type `T` as well as an additional integer.[1] Thus, the function that maps a natural number n to the size of the number type representing n is injective. If, in our program, we instantiate (create a variable of) the number type representing $f(x)$, the compiler has to work out its size and thus compute $f(x)$. We note that it should be possible to devise a (compiler-dependent) scheme for extracting $f(x)$ from the executable generated by the compiler, but we will not go into the technicalities of how such a scheme might work.

The way in which we have specified functions is quite similar to the pattern matching used in functional programming languages. Consider for example our implementation of primitive recursion. First we try to match the template arguments with `F<X,zero>` (with arbitrary `X`). If this fails, we try to match them with `F<X, Y>` (which should be successful). In the latter case we know that `Y` $\not\equiv$ `zero`, so we can safely use `Y::pre`.

It is surely interesting to find other programming languages for which compile-time computations are possible. Veldhuizen [8] presented an experimental compiler for Java that performs partial evaluation to improve the performance of numerical code. The worst-case running time of this compiler is quadratic.

As we have seen, C++ allows arbitrary computations at compile time. This is something of a dilemma. We want the power to perform complex program manipulations at compile time, but we would also like to have a guaranteed time bound for the compiler. On the other hand, it could be argued that, since we are completely responsible for the *run-time* complexity of our programs, we should simply get used to being responsible for the *compile-time* complexity, too.

References

[1] Neil D. Jones. An Introduction to Partial Evaluation. *ACM Computing Surveys*, 28(3):480–503, 1996.

[2] Carl H. Smith. *A Recursive Introduction to the Theory of Computation*. Springer, 1994.

[3] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.

[4] Erwin Unruh. Prime number computation. ANSI X3J16-94-0075/ISO WG21-462, 1994.

[1] Strictly speaking, for machines with very restrictive alignment rules, it is conceivable that one additional integer variable would not necessarily increase the size of the type (because of structure packing). In this case, we could use more than one dummy integer to make sure we always have `sizeof(suc<T>)` $>$ `sizeof(T)`.

[5] Erwin Unruh. Template Metaprogrammierung, 2002. URL: http://www.erwin-unruh.de/meta.html (in German).

[6] Todd L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, 1995.

[7] Todd L. Veldhuizen. C++ Templates as Partial Evaluation. In *Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, Technical report BRICS NS-99-1, University of Aarhus, pages 13–18, 1999.

[8] Todd L. Veldhuizen. Just when you thought your little language was safe: "Expression Templates" in Java. In *Proc. of the 2nd Symp. Generative and Component-Based Software Engineering (GCSE)*, volume 2177 of *Lecture Notes in Comput. Sci.*, pages 188–200. Springer, 2001.