General Constant Expressions for System Programming Languages

Gabriel Dos Reis
Texas A&M University
gdr@cse.tamu.edu

Bjarne Stroustrup
Texas A&M University
bs@cse.tamu.edu

Abstract

Most mainstream system programming languages provide support for builtin types, and extension mechanisms through user-defined types. They also come with a notion of constant expressions whereby some expressions (such as array bounds) can be evaluated at compile time. However, they require constant expressions to be written in an impoverished language with minimal support from the type system; this is tedious and error-prone. This paper presents a framework for generalizing the notion of constant expressions in modern system programming languages. It extends compile time evaluation to functions and variables of user-defined types, thereby including formerly ad hoc notions of Read Only Memory (ROM) objects into a general and type safe framework. It allows a programmer to specify that an operation must be evaluated at compile time. Furthermore, it provides more direct support for key meta programming and generative programming techniques. The framework is formalized as an extension of underlying type system with a binding time analysis. It was designed to meet real-world requirements. In particular, key design decisions relate to balancing experssive power to implementability in industrial compilers and teachability. It has been implemented for C++ in the GNU Compiler Collection, and is part of the next ISO C++ standard.

Categories and Subject Descriptors D.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Standardization

1. Introduction

Modern high level programming languages typically feature a variety of abstraction mechanisms to support popular programming methodologies: object-based programming, object-oriented programming, functional programming, generic programming, etc. For system programming, however, the abstraction support is incomplete. A distinctive trait of system programming is the ability to describe data known at program translation time. Such data are usually denoted by so called *constant expressions* (C, C++, Java, etc.) or *static expressions* (e.g. Ada). Furthermore, for embedded systems we typically need to describe data that should reside in Read Only Memory (ROM). It is also common to need tables with non-trivial structure that ideally are computed at compile time or link time to minimize startup costs and memory consumption. Mainstream system programming languages, such as C and C++ do not have standard, reliable, and systematic language features for expressing that. At best, values that must be known before run time can only be expressed in an impoverished subset of a language and must rely on non-standard conventions selectively implemented by compilers.

We developed a general and formal model for compile time evaluation that can be applied to provide compile-time evaluation for a wide variety of programming languages. For space constraints, we will limit this paper to an informal presentation of that framework. Our examples will be in C++ and some of our specific language-technical design decisions reflect our experience with the C++ language, implementation, use, and the C++ standards process.

Consider a simple a jump table:

```
typedef void (*handler_t)();
extern void handler1();
extern void handler2();
const handler_t jumpTable[] = {
  &handler1, &hanlder2
};
```

Obviously we could place `jumpTable` in ROM — all the information to do so is available. Indeed, some (but not all) production compilers recognize such constructs and generate appropriate static initialization. However, relying on conventions leads to brittle and non-portable code, pushing programmers towards proprietary languages and language extensions.
The usual meaning of `const` object is an unchanging datum, a value. However, that interpretation is insufficient to ensure that the object is initialized in such a way it is placed in ROM. Consider:

```
double mileToKm(double x) { return 1.609344 * x; }

const double marks[] = {
   mileToKm(2.3), mileToKm(0.76)
};
```

This defines the array object `marks` as unchanging, but there is no guarantee that the values are computed at compile time. Indeed since `mileToKm()` is a function, this will generate (redundant) dynamic (run-time) initialization, that unfortunately ensures that `marks` never ends up in ROM. ISO C++ [1] does not provide a systematic and reliable way for the programmer to require that a particular object must be evaluated at compile time or link time.
A workaround for this “permissive” `const` semantics is for programmers to use preprocessor macros, designed for token substitutions, rather than general and type safe language mechanisms. For example the function `mileToKm` may be replaced with the macro

```
#define MILE_TO_KM(M) ((M) * 1.609344)
```

SAC '10 March 22–26, 2010, Sierre, Switzerland
Copyright © 2010 ACM 978-1-60558-638-0/10/03...$5.00

However, preprocessor macros are brittle abstraction tools, fraught with type, namespace, and scope problems. For example, assume that two collaborating teams want to make sure that units of measurement are not confused (potentially leading to catastrophic crashes). An obvious solution would be to adopt a "typeful programming" discipline using distinct classes/types to represent values of differing units. For instance, they may provide `LengthInKM` and `LengthInMile` types so that the `marks` initialization could be written:

```
const LengthInKM marks[ ] = {
   LengthInKM(MILE_TO_KM(2.3)),
   LengthInKM(MILE_TO_KM(0.76))
};
```

That looks good, but the use of the class `LengthInKM` with a constructor brings back the problem with ROMability avoided through the use of the macro `MILE_TO_KM` instead of the function `mileToKm`. Again, this may go unnoticed since the resulting (dynamic initialization) code is correct from the C++ language point of view. What we see here is a failure to support user-defined type abstractions at the same level as builtin types. In reality, the problem is worse because it forces programmers to avoid key facilities of the C++ standard library and of embedded systems support libraries.

The problems illustrated above are not specific to C or C++. They are shared by all mainstream languages used for system programming. Almost invariably, they define the notion of *constant expression* or *static expression* as an expression of builtin types (e.g. integers only) using only a restricted list of builtin operators (i.e. no functions) and builtin constants. This paper proposes a methodology to develop the notion of *general constant expression* applicable to most modern system programming languages. The methodology has been instantiated for C++, implemented in a version of the GNU Compiler Collection, and has been accepted for the C++0x draft standard. It introduces two notions:

- *literal types*: A literal type is one which is "sufficiently simple" so that the compiler knows its layout at compile time (see §2.2).
- *constexpr functions*: A constexpr function is one which is "sufficiently simple" so that it delivers a constant expression when called with arguments that are constant values (see §2.1).

The "length example" can be written more clearly, type safe, and without added run-time or space overheads in C++0x. We start by defining a type `LenghtInKM`:

```
struct LengthInKM {
  constexpr explicit LengthInKM(double d) : val(d) { }
  constexpr double getValue() { return val; }
private:
  double val;
};
```

The function with the same name as the class, `LengthInKM` is a constructor; it specifies that a `LengthInKM` must be initialized by a double-precision floating-point value.

The `LengthInMile` type is defined similarly, but has an additional function, `operator LengthInKM`, that specifies a conversion from `LengthInMile` to `operator LengthInKM`:

```
struct LengthInMile {
  constexpr explicit LengthInMile(double d) : val(d) { }
  constexpr double getValue() { return val; }
  constexpr operator LengthInKM() {
     return LengthInKM(1.609344 * val);
  }
private:
  double val;
};
```

Given those two types our example shrinks to:

```
constexpr LengthInKM marks[] = {
  LengthInMile(2.3), LengthInMile(0.76)
};
```

Obviously the representations of both classes `LengthInKM` and `LengthInMile` are "sufficiently simple" for the compiler to "understand" — each is represented by a single value of the built-in type `double`. All member functions are defined with the keyword `constexpr` to ensure that the compiler checks that it can evaluate their bodies at compile time. Obviously, these functions are "sufficiently simple" for compile time evaluation.

We use two objects of type `LengthInMile` to initialize an array `marks` of objects of type `LengthInKM`. Because `marks` is defined with the keyword `constexpr` the construction and conversions are all performed at compile time. The compiler will issue an error if the initialization of a constexpr object is dynamic. The notion of constant expression is no longer limited to a handful builtin types and operations, but also applies to user-defined types and functions. Critically, the rules for compilation and evaluation are simple for both compilers and users.

The rest of this paper is structured as follows. Section 2 introduces the notion of literal types, constexpr functions, and discusses some of the pitfalls that need to be avoided. Then §3 considers recursive constexpr function. Section 4 consider extensions to functions taking parameters by reference; then §5 considers interaction with object-oriented facilities (e.g. inheritance, virtual functions). Finally we highlight related works in §6 and conclude in §8. Fine points and technical aspects of the design are made precise in a detailed static semantics following a natural semantics style [3] presentation contained in a much longer technical report.

2. Constant Expressions

We start with the notion of a constant expression as it has appeared in C++ from its inception. From that, we proceed through a sequence of generalizations.

In some contexts — such as array bounds — an expression is required to evaluate to a constant. A constant expression is either a literal, a relational expression the sub-expressions of which are constant expressions, an arithmetic expression of type `int` whose sub-expressions are constant expressions, or a conditional expression whose sub-expressions are all constant expressions. The name of global variables defined with the `const` keyword and initialized with constant expressions also evaluates to a constant expression. For example

```
const int bufsz = 256 * 4;
int main() {
  int buffer[bufz] = { 0 };   // bufsz is constant 1024
  // ...
}
```

This definition of constant expression in C++ [1] is quite conventional. In particular, a call to a (user-defined) function with constant expression arguments is never considered a constant expression. For example:

```
int round_up(double x) { return int((x+x)/2); }

int buf[round_up(0.76)];  // error
```

The call `round_up(0.76)` is not considered a constant expression, even though recent extensions to C allow floating-point values to participate in constant expression evaluation.

2.1 Constexpr functions

Our first generalization modifies the notion of constant expression so that a call to a "sufficiently simple" function with constant expression is a constant expression, e.g. one that can be evaluated at compile time. A *constexpr function* is a function the definition of which has the following characteristics:

- its return type, and the types of its parameters (if any), are *literal types* (see §2.2). For concreteness, literal types include `bool`, `int`, or `double`;
- its body is a compound statement of the form

  ```
  { return expr; }
  ```

  where *expr* is such that if arbitrary constant expressions of appropriate types are substituted for the parameters in expr, then the resulting expression is a constant expression as defined in introductory paragraph of §2. The expression *expr* is called a *potential constant expression.*

For example, the function `mileToKm` from §1 is a constexpr function but the following is not

```
int next(int x) { return x = x + 1; }
```

because it uses the the assignment operator.
We now extend the notion of constant expression as follows. A constant expression is an expression of scalar type and of the following form:

1. a literal,
2. a name that denotes a global variable of scalar type defined with `const` and initialized with a constant expression,
3. a relational expression involving only constant sub-expressions,
4. an arithmetic expression of scalar type involving only constant sub-expressions,
5. a conditional expression of scalar type involving only constant sub-expressions,
6. a call to a constexpr function with constant expression arguments.

In principle, a compiler can infer from a function definition whether it is constexpr or not. However, for ease of use and ease of implementation, we require that a constexpr function definition be preceded by the keyword `constexpr`. The intent is that at the point of definition, the compiler should validate that indeed a function definition satisfies all conditions to be eligible, as expected by the programmer. The early checking offered by this requirement is especially useful in large scale programming where programs are assembled from several libraries and third party components. Here are some examples of constexpr functions

```
constexpr int square(int x) { return x * x; } // OK

constexpr int int_max() { return 2147483647; } // OK

constexpr int isqrt_helper(int sq, int d, int a) { // OK
   return sq <= a ? isqrt_helper(sq+d,d+2,a) : d;
}

constexpr int isqrt(int x) {         // OK
  return isqrt_helper(1,3,x)/2 - 1;
}
```

The handling of recursive function is discussed in §3.
It is not an error to call a constexpr function call with non-constant arguments in a context where a constant expression is not required; the expression is to be evaluated at run time. The alternative would be to double the number of functions (one for constant expression evaluation and one for run-time evaluation). That would not be viable in real-world code. Another practical concern addressed by this design is that a compiler need only store the body of a constexpr function for potential evaluation. This is important for the compilation of large programs. Basically, constexpr functions define a functional sub-language. For simplicity, we exclude loops and everything else in C++ that require more than an expression. This reflects our design goal of a simple, yet powerful type system extension.
The strictness of the constexpr function rule, just happens to be valuable to optimizers even if a call to a constexpr function does not involve constant expressions. In particular, that a constexpr function is side-effect free ("pure").

2.2 Literal Types

Limiting constexpr functions to receive and return only values of builtin types is useful, but still restricts the programmer to expressions of (builtin) scalar types. This goes against the principle that a high level language should support user-defined types just as well as builtin types, so as to allow general use of the type system. A *literal type* is

- a scalar type, or
- a class with all data members of literal types, and a constexpr constructor.

A *constexpr constructor* is just like a constexpr function, except that it must initialize the data members in the member-initializer part and those initializations must involve only potential constant expressions, and its body is empty. The restriction on the body is quite natural for the kind of types people most often want in ROM. Here is an example

```
struct complex {
   constexpr complex(double x = 0, double y = 0)
                : re(x), im(y) { }
   constexpr double real() { return re; }
   constexpr double imag() { return im; }
private:
   double re;
   double im;
};
```

It is often stated that a datatype like `complex` should be built into the language just like `int` for reasons of efficiency. We contend that once we modify our notion of constant expression to include objects of literal types initialized with constant expressions, then the class `complex` provides a datatype just as efficient as if it were made builtin. In particular, we can define:

```
struct imaginary {
  constexpr explicit imaginary(double z) : val(z) { }
  constexpr operator complex() { return complex(0.0,val); }
private:
  double val;
};
constexpr imaginary I = imaginary(1.0);
```

This provides the classic imaginary unit `I` just as efficiently and notationally cleanly as if it were hardwired into the language.
To turn `complex` into a full blown user-defined arithmetic type with the usual arithmetic operations, we need to extend the notion of constexpr function to member functions. Member functions have a hidden (pointer) parameter: In C++, this parameter is called "the `this` pointer." For each call, `this` points to the object on which the member function is invoked. That is, the definitions of `real()` and `imag()` are equivalent to:

```
constexpr double real() { return (*this).re; }
constexpr double imag() { return (*this).im; }
```

We don't propose to handle pointers in general at compile time. However, the restricted form of the `this` pointer means that we can deal with it as long as we can cope with member selection. Handling class object member selection is easy: if the object is of

literal type and created with constant expressions arguments to the constructor, then it is clear that all its components are constant values. Also, all fields correspond to offsets known at compile time. Consequently, the compiler can evaluate a field member selection of such an object at compile time. Next, if a member function of a class χ is defined with the `constexpr` keyword, and would have been a constexpr function (as defined in §2.1) if the expression `*this` is replaced by an arbitrary constant expression of type χ, then that member function is considered a member function. For example `complex::real` and `complex::imag` are constexpr member functions.
We can now summarize our notion of general constant expression and illustrate its effectiveness. A constant expression is an expression of literal type of the following form:

1. a literal,
2. a name that denotes a global variable of literal type defined with the keyword `constexpr` or `const` and initialized with a constant expression,
3. an object created by a constexpr constructor with constant expression arguments,
4. a member selection of a constant expression of literal class type or array of constant expressions,
5. a relational expression involving only constant expression sub-expressions,
6. an arithmetic expression of scalar type involving only constant expression sub-expressions,
7. a conditional expression of scalar type involving only constant sub-expressions,
8. a call to a constexpr function or constexpr member function with constant expression arguments.

This general notion of constant expression is simple. It is merely a recursive definition based on composition of built-in types and built-in operators. However it is powerful enough to turn a user-defined datatype such as `complex` into one that delivers the same efficiency as a built-in type. In particular, the usual arithmetic operations may be defined as follows:

```
constexpr complex operator+(complex x, complex y) {
  return complex(x.real()+y.real(), x.imag()+y.imag());
}
constexpr complex operator*(complex x, complex y) {
  return complex(x.real()*y.real() - x.imag()*y.imag(),
            x.real()*y.imag() + x.imag()*y.imag());
}
```

This allows us to handle the types that are most frequently considered for placement in ROM and for which the highest number of objects are needed. Our aim is not to support arbitrary object creation and object manipulation at compile-time, but to provide simple support for objects and operations for which the distinction between run-time and compile-time evaluation matters. If you want more, we observe that what we provide is obviously Turing complete.

2.3 Static Initialization and Phase Distinction

General evaluation of expressions at compile time is tricky; especially for system programming languages. Indeed, one has to be careful and distinguish between entities that are fully defined or known only at different phases: compile time, link time, and run time. For example, the addresses of global variables are not known until link time. That limits the kind of static initialization that can be performed and used by the compiler before link time. For example C and C++ do not allow the following (assume that the variables are defined at top level)

```
const int n = 42;
const int p = (int)&n;  // dynamic initialization
int array[n] = { };     // OK, n is constant
int ary[p] = { };       // error: p is not constant
```

This is because the address of the variable `n` is not known until link time, therefore cannot be considered a constant expression. Examples, such as these makes programmers ask for a guarantee that certain initializations are compile-time evaluated. Using `constexpr` as part of a variable definition achieves that. For example:

```
constexpr int n = 42;             // OK
constexpr intptr_t p = (intptr_t)&n; // error
const intptr_t q = (intptr_t)&n; // OK; dynamic init.

constexpr complex z = I;          // OK
constexpr double d = z.real();    // OK
constexpr double e = sqrt(3.14); // error
const double f = sqrt(3.14);     // OK; dynamic init.
```

The standard library square root function, `sqrt`, is not a constexpr function so the rather innocent-looking initialization of `e` is an error.
It has been argued that no such compile-time initialization guarantee is necessary and we can rely on the programmers and the compilers to "do the right thing." Experience shows that for large number of programmers dealing with large programs using multiple compilers that argument simply isn't true. Plain C++ `const` is not enough, exactly because it does not offer that guarantee — it is too flexible for some important uses.

3. Handling Recursion

By allowing recursive constexpr functions we open the possibility for a compiler to enter an infinite loop. In particular, the type system is not decidable anymore. However, there are several ways to admit recursion while preventing infinite loops. One way is to restrict the definition is such a way that termination is always decided by the syntactic structure of the function. However, for C++ that would just add complexity to the syntax without providing significant benefits. Recursion at compile time is already common in C++ programs and recursive function calls can be handled using existing techniques.
C++ programmers are comfortable with the idea that a compiler may reject their programs not because its capacities are exceeded. In C++ [1], one can already write

```
template<int n>
  struct Fact {
    enum { Value = n * Fact<n-1>::Value };
  };

template<>
  struct Fact<0> {
    enum { Value = 1 };
  };

constexpr int f5 = Fact<5>::Value;
```

Such constructs are popular, but they are indirect expressions of ideas so they constitute a barrier for understanding and maintenance. As a data point, we proposed an early version of constexpr (for C++0x) without recursion, but that led to many requests for that feature, complete with real-world constant expression use cases. Consequently, we directly support recursive constexpr functions. They bring clarity, and impose no additional burden on programmers. For example, we can simplify the example above to:

```
constexpr int fac(int n) {
  return n == 1 ? 1 : n*fac(n-1);
}

constexpr int f5 = fac(5);
```

We stress that our framework still beneficial for a language that elects to restrict allow recursions at compile time.

4. Supporting References Parameters

In this section we examine what it takes to allow a constexpr function with reference parameters. This issue is of importance to a system programming language that also supports object oriented programming or generic programming.

Reference parameters are conventionally implemented as pointers to objects, so their evaluation at compile time is quite tricky. In general, it's impossible without using a full interpreter. In particular, we need to maintain phase distinction, avoid the problems mentioned in §2.3, and keep the compiler small and fast.

The C++ programming language has an eager dynamic semantics with two modes of parameter passing: pass-by-value, and pass-by-reference. The pass-by-value mode makes a copy of the argument into a temporary variable (the parameter), so that evaluation of expressions are insensitive to object location. The pass-by-reference mode works by binding the address of the argument object to the parameter. Since the address of an object is not known until link time or run time, we have to be cautious when attempting to evaluate an expression at compile-time.

Consider a simple example:

```
template<class T>
  constexpr T max(const T& a, const T& b) {
    return (a > b) ? a : b;
  }

  constexpr double d = max(one_val,another_val);
```

First, note that as in the case of constant expression of literal type, we don't really need to know the address of the object. Rather, all we need to know is the value the reference (or dereference) evaluates to. If that value is a constant expression, and no other part of the expression is sensitive to the addresses, then it is safe to do a substitution at compile time. That is precisely how we extend the notion of constexpr function to include functions with reference parameters. So, here is our revised definition of constexpr function: a function is said *constexpr* if

- it is defined with the `constexpr` keyword,
- its return type is a literal type,
- each parameter in the parameter list (if non-empty) is of literal type, or a reference to a literal type,
- its body is a compound statement of the form

  `{ return` *expr*`; }`

  where *expr* is a potential constant expression.

This definition is sufficient to guarantee that there is no difference between the value produced by a runtime evaluation and a compile time evaluation.

This technique for handling references applies for far larger subset of C++ (including some uses of the `new` operator) than what we needed for the practical problems that motivated this work.

In what circumstances would a programmer ever want to define a constexpr function with parameters of reference types? First, in C++, for most generic (template) functions (such as `max` above), people declare function parameters with `const` reference type as there is no general reliable mechanism to select generically the argument passing mode based on the type of the argument. Second, some languages such as Java rely extensively on pass-by-reference semantics. It is therefore important to know that with sufficiently natural restriction, the framework applies. Third, most C++ standard library functions (e.g. `std::abs`, `std::main`, `std::max`, etc.) are defined generally as taking `const` reference parameters. If we don't handle them, then programmers would invent brittle workarounds, leading to code duplications, fraught with all sorts of traps. Fourth, in C++, there are some situations where there is no choice but to use a `const` reference for the parameter in order to preserve as much type information as possible that would participate in the computation of the constant value. A concrete example is when one attempts to pass an array to a function. In C and C++, an array object as argument in pass-by-value mode automatically decays to the address of the first object, which loses the array length information. A classic way to preserve the array length information is to declare that the array is to be passed by reference, as exemplified by the following classic program fragment:

```
template<typename T, int N>
  constexpr int length(const T(&ary)[N]) { return N; }
```

Here the parameter `ary` is declared to be a reference to an array of type `const T[N]` where `T` and `N` are the template parameters. When called as in

```
const int grades[] = { 20, 10, 13, 3 };

int main() {
   return length(grades);  // OK: returns 4
}
```

the template parameter `T` is bound to `int`, and the parameter `N` is bound to 4 (the length of `grades`). Note that this function is very convenient for retrieving length of statically initialized array.

5. Object Orientation

In this section we discuss interactions between object oriented features and constant expressions of user-defined types that may involve inheritance and virtual function (or dynamic dispatch).

Since our primary goal is to support, at the language level, compile-time evaluation of constant expressions of user-defined types, why bother with object-oriented features which typically involve run-time semantics? Well, inheritance and dynamic dispatch are important in all languages deemed "object oriented." Thus, it is an advantage that under appropriate appropriate conditions, our methodology still applies so that a choice of an object-oriented techniques does not automatically implies poor support for constant expression evaluation. In fact, some simple object-oriented techniques can be used to simplify constant expression code.

Support for object-oriented programming tends to depend on the programming language's object model. For concreteness, we will use the C++ object model, which supports combinations of single inheritance, multiple inheritance, and virtual inheritance. In both the single inheritance and multiple inheritance case, the offsets of data members are known at compile time so the discussion in §2.2 carries verbatim. The only natural restriction is that all base classes should be literal types. For example, in C++0x we can factor out the common parts of `LengthInKM` and `LengthInMile` into a base class:

```
struct Length {        // no unit
  constexpr explicit Length(double d) : val(d) { }
  constexpr double getValue() { return val; }
private:
  double val;
};
```

This base class can be used in the obvious way:

```
struct LengthInKM: Length {
  constexpr explicit LengthInKM(double d)
                     : Length(d) { }
};

struct LengthInMile: Length {
  constexpr explicit LengthInMile(double d)
                     : Length(d) { }
  constexpr operator LengthInKM() {
     return LengthInKM(1.609344 * getValue());
  }
};
```

Our user code is unchanged:

```
constexpr LengthInKM marks[] = {
  LengthInMile(2.3), LengthInMile(0.76)
};
```

We will not discuss virtual bases because we have never seen an example where they would benefit from compile time evaluation.

6. Related Work

Most of the related work is found in the definition of real programming languages for system programming such as Ada [7] and C++ [1]. Our design for general constant expressions can be understood as a carefully developed instance of partial evaluation [9, 2, 4], built into the type system of an existing programming language that has been under intense industrial use for nearly three decades. There is a huge body of work done by the partial evaluation community, with a dedicated conference. Our work aims at simple and general types rules — easily understood by programmers and compiler implementors — that guarantee a specific level of effective partial evaluation of programs across all compilers. This contrasts to relying on un-annotated programs or on the quality of compilers and optimizer. Compile-time evaluation of user-defined functions has been part of the Lisp macro system. Its usage requires careful explicit annotation for the three-binding times found in Lisp: translation-time, load-time, and run-time. Our work aims at simplicity, ease of use, and ease of implementation. It guarantees function inlining and constant folding [6] in a wide range of contexts. Our design presents a simple coherent set of rules, as opposed to building seperate metalanguage on top of the existing programming language [5]. It requires only a careful semantics extension of existing types.

7. Acknowledgment

This work was partly supported by NSF grant CCF-0702765.

8. Conclusion

We have presented an effective general methodology to extend the notion of constant expressions found in major system programming languages to include user-defined types and user-defined (possibly recursive) functions. The result offers more reliable semantics for formerly ad hoc notions of ROM. This form of constant expressions can also be seen as explicit programmer-supplied annotations to guide offline partial evaluation, therefore opening up more opportunities for constant propagations, easing data dependence analysis and possible parallelization. The framework presented in this paper can be extended to include a streamlined effect type and system [8] to allow imperative constructs are long as the effects are local and not reflected in the outcome of the evaluation. This paper has focused on the vast majority of practical needs of compile time evaluations.

References

[1] International Organization for Standards. *International Standard ISO/IEC 14882. Programming Languages — C++*, 2nd edition, 2003.

[2] Neil Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, June 1993.

[3] Gilles Kahn. Natural Semantics. In *STACS '87: Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, pages 22–39, London, UK, 1987. Springer-Verlag.

[4] J. Koop and O. Rüthing. Constant propagation on predicated code. 9(8):829–850, 2003.

[5] Sean Seefried, Manuel Chakravarty, and Gabriele Keller. Optimising Embedded DSLs Using Template Haskell. In *In Proceedings of the 3rd International Conference on Generative Programming and Component Engineering*, pages 186–205, Vancouver, BC Canada, 2004. Springer-Verlag.

[6] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[7] S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, and Erhard Ploederer, editors. *Consolidated Ada Reference Manual*, volume 2219 of *Lecture Notes in Computer Science*. Springer, 2000.

[8] Jean-Pierre Talpin and Pierre Jouvelot. The Type and Effect Discipline. *Inf. Comput.*, 111(2):245–296, 1994.

[9] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.