Five compilation models for C++ templates
(Extended Abstract)

Todd L. Veldhuizen
Extreme Computing Laboratory
Indiana University Computer Science Department
Bloomington Indiana 47405, USA
tveldhui@acm.org

Abstract

This paper proposes an alternate structure for C++ compilers. Type analysis is removed from the compiler and replaced with a *type system library* which is treated as source code by the compiler. Type computations are embedded in the intermediate language of the compiler, and partial evaluation is used to drive type analysis and template instantiation. By making simple changes to the behavior of the partial evaluator, a wide range of compilation models is achieved, each with a distinct tradeoff of compile time, code size, and code speed. These models range from purely dynamic typing – ideal for scripting C++ – to profile-directed template instantiation. This approach may solve several serious problems in compiling C++: it achieves separate compilation of templates, allows template code to be distributed in binary form by deferring template instantiation until run time, and reduces the code bloat associated with templates.

1 Introduction

A program in a statically typed language such as C++ describes two computations. There is the usual computation we think of – values being combined to produce other values – but there is also a *type computation*, in which types are combined to produce other types. In this code:

```
float f(int x, float y)
{
  return x + y;
}
```

there is an implicit type computation which determines that *int* + *float* = *float*. The compiler evaluates the type computation at compile time to check types and report errors.

This type computation is normally built into the compiler: the functions which handle arithmetic type promotions, inheritance, and other language features are functions *in the compiler*. In this paper we explore an alternative compiler structure, illustrated by Figure 1: the type system is implemented in a library, and this *type system library* is treated as source code by the compiler. Rather than performing type analysis, the front end inserts calls to this type system library as it translates the source code. Partial evaluation is used to optimize away the type system code at compile time. This approach to compiling C++ uncouples the idea of genericity (functions which operate on arbitrary types) from the idea of specialization (duplicating functions to improve performance).

We illustrate the approach by examining a prototype compiler, called *Lunar*, which implements a modest C++ front end. We start by describing the important features of Lunar's structure and intermediate languages (Section 2). Partial evaluation plays a key role in the compilation process, so we overview the important features of Lunar's partial evaluator (Section 3). Next we describe the process of translating C++ into Lunar's intermediate language (Section 4). The translation is unique because type computations are embedded in the intermediate language. By controlling how the partial evaluator specializes functions, five distinct compilation models may be achieved (Section 6). Finally, we discuss performance issues and related work (Section 7).

2 The intermediate languages

Lunar uses a family of intermediate languages called IL_2, IL_1, and IL_0. The C++ front end generates IL_1, and the partial evaluator operates over IL_0.[1] There are automatic transforms which lower IL_2 to IL_1 to IL_0. IL_0 resembles quadruple-form,[2] with all intermediate results being explicitly named. It is call-by-value, typeless, and single-assignment. It is easier to partially evaluate IL_0 if some higher-level control flow structures are still in place, so IL_0 provides if/else, exceptions, loop/break, function calls and block-scoped variables.

IL_0 exists in the compiler as trees – Lunar does not dump intermediate forms to files – but for rendering IL_0 we use the surface syntax of Figure 2. The only distinct feature of IL_0 is its support for multiple values. Multiple values may be created at any program point by using the square bracket `[]` notation. This statement:

```
return [4,5]
```

returns the pair of values $\langle 4, 5 \rangle$. Multiple values must be immediately bound to variables using an init statement, or the additional values are lost; in this code:

```
[x,y] := [1,2]
foo([1,2],[3,4,5])
```

[1] Parts of the C++ type system library are implemented in IL_2, and Lunar's Java front end generates IL_2. IL_1 is similar to IL_0 but allows variable assignments.

[2] In quadruple form, most instructions are of the form: $r \leftarrow x_1 * x_2$, where r is a name for the result, x_1 and x_2 are names or literals, and $*$ is an operator.

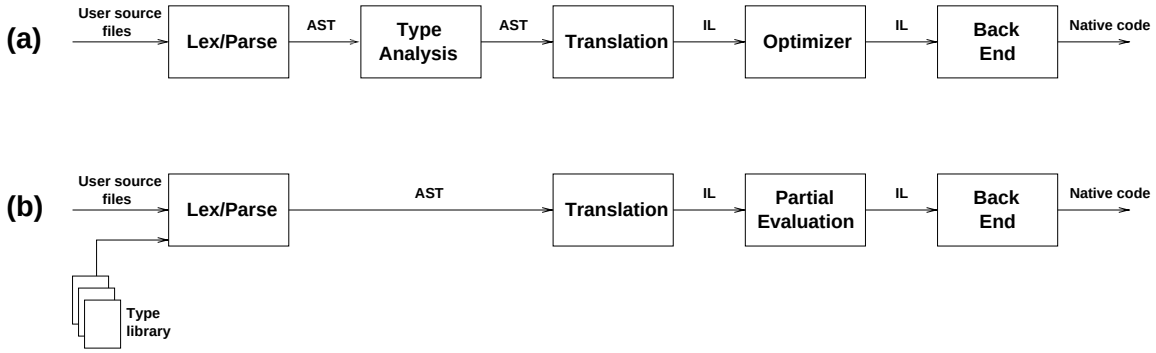

Figure 1: Comparison of two compiler structures: (a) Typical compiler: source files are parsed to abstract syntax trees (AST), and type analysis decorates the trees with type information. The trees are then translated to an intermediate language (IL), optimized, and a back end produces native code. (b) Structure explored in this paper: the type system is implemented by a library, and is treated as just another set of source files. A partial evaluator is used to optimize the user program in the context of the type system, resulting in type analysis being performed at compile time.

```
float pow(float x, int n)
{
  if (n == 0)
    return 1.0;
  else
    return x * pow(x,n-1);
}

float a, b;
a = pow(b,3);
```

(a) Some code

```
// pow has been specialized for n=3
float pow_3(float x)
{
  return x * x * x;
}

float a, b;
a = pow_3(b);
```

(b) After partial evaluation

Figure 3: Partial evaluation example

the variable `x` is initialized with `1` and `y` is initialized with `2`, but the call to function `foo` is equivalent to `foo(1,3)` – the additional values of the arguments are discarded.

In Lunar's C++ front end, multiple values are used to turn C++ values into $\langle value, type \rangle$ pairs; for example, the C++ expression `5.0` is translated to the pair of Lunar values `[5.0,double]` where `double` is a global variable pointing to a data structure representing the C++ type `double`.

3 Partial evaluation

A partial evaluator takes a program, performs the operations which depend only on known values, and outputs a specialized program [6]. The standard example is shown in Figure 3.

Lunar's partial evaluator is a blend of functional-language-style partial evaluation and imperative language optimization techniques. In spirit, it is a partial evaluator – particularly in this application of compiling C++ – because its goal is to evaluate the type system portion of the code at compile time. There are five critical components in the mix:

- **Constant folding and propagation**: the partial evaluator folds primitive operations whose operands are known at compile time, replacing (for example) `3+7` with `10`. It also propagates constants through variables, turning (for example) `x := 3; y := x` into `x := 3; y := 3`.
- **Copy propagation**: if an initialization `x := y` is encountered, later uses of `x` are replaced with `y`.
- **Heap analysis**: constants and copies are propagated through heap (and stack) data structures when possible. Lunar's heap analyzer builds on the tradition of partly-static data structures in the partial evaluation community (e.g. [3]), and of store analyzers in the imperative world (e.g. [9, 11]). Lunar uses a split-store analysis: it distinguishes between store operations which are *mutable* (i.e. someone may later overwrite that region of the heap) from those which are *final*. Use of a final store operation implies an assumption that nothing will ever be written to the same heap location. (In Lunar's C++ front end, this is true of the compiler-generated data structures representing type information – it is not possible for users to modify object layouts, for example – but not true of operations on user-defined classes). Mutable store operations are subject to aliasing; final store operations are not. Propagating constants and copies through the heap is easy for final regions of the heap, and difficult – often impossibly so – for mutable regions.
- **Dead code elimination**: variables, globals, and functions which are no longer needed are stripped from the program. Function calls where the return value is discarded are removed if the function was determined to have no non-local side effects.
- **Specialization**: at each call site, the partial evaluator may choose to specialize the function being called based on some of the argument values. In the imperative world this is known as *procedure cloning*. The front

d	::=	`function` $f(v_1, \ldots, v_n)$ b	function definition
	|	`global` v `=` b	global value definition
b	::=	`blockscope` $\{v_1, \ldots, v_k\}$ $s_1 \ldots s_{n-1}, s_n^*$	basic block
s^*	::=	s	
	|	`return` e	function return
	|	`break` e	break from loop
	|	`throw` e	exception throw
	|	`[]`	void statement
s	::=	e	
	|	`init [` $v_1, \ldots, v_j$ `] =` e	variable initialization
	|	`try` b_1 `catch [` $v_1, \ldots, v_j$ `]` b_2	exception handling
e	::=	| p `(` $t_1, \ldots, t_n$ `)`	primitive
		| t_0 `(` $t_1, \ldots, t_n$ `)`	function call
		| `if` t_0 `then` b_1 `else` b_2	if expression
		| `[` $t_1, \ldots, t_n$ `]`	multiple-value construction
		| `loop` b	loop expression
t	::=	c	constant
	|	v	variable use

Figure 2: The grammar for IL_0 consists of *(d)* top-level definitions; *(b)* basic blocks; *(s*)* terminal statements; *(s)* statements; *(e)* nontrivial expressions; *(t)* trivial expressions.

end may allow or disallow certain specializations; when the Lunar C++ front end is running in "standard-compliant" mode, it only allows functions to be specialized based on arguments which represent types.

In addition to these optimizations, the partial evaluator does liveness analysis, escape analysis, a limited form of alias analysis, stack allocation analysis, and inlining.

The type computation of C++ has some functional aspects: it is single assignment (you cannot change the type of a variable, once declared); data structures representing classes and structures are mostly immutable (you cannot dynamically add a new base class, nor change the type signature of a function). You cannot take the address of an uninstantiated template function, which means the partial evaluator can determine at every call site of a template function exactly which function is being invoked. These functional aspects are critical to optimizing away the type library routines at compile time.

4 The C++ front end

The C++ front end is $\sim$ 10000 lines of source code, not including the parser. It implements a very modest subset of C++, just enough to demonstrate and test the approach. It handles simple functions, classes, and templates. It does not yet handle overloading, non-type template parameters, virtual base classes, namespaces, nor a few hundred other C++ features.

The front end translates C++ to a version of the Lunar intermediate language, called IL_1, which is a little higher-level than IL_0 and allows assignments. The high-level structure of the compiler is shown in Figure 4. The C++ type system is implemented by a type library. Part of this library is implemented directly in IL_2 – a high-level version of the Lunar intermediate language– and part of it is implemented in C++. The type system library is treated as just another set of source files by the compiler.

4.1 The `type_info` hierarchy

All C++ types are represented by pointers to data structures which are subclasses of `type_info`. These classes are declared in a C++ file, part of which is shown in Figure 5.

Instances of the `type_info` classes are created by code written in IL_2. This allows the type library to express that fields of the `type_info` classes are immutable.[3] Without this knowledge, it would be difficult for the partial evaluator to read fields from these data structures: a closed-program alias analysis would be required to exclude the possibility that someone, somewhere, was writing to these fields. Closed-program analyses do not interact well with dynamically-linked shared libraries, which are commonplace now.

4.2 Translation of simple, non-template code

For every variable `x`, the front end creates a variable `x$type` which points to the appropriate `type_info` object representing `x`'s type.

Translation of C++ expressions is straightforward, with each node in the abstract syntax tree often translating directly to a call into the type library. New variables are created to hold intermediate results. For example, this code:

```
void simpleMath()
{
  int x = 5;
  float y = 7.0;
  int z = x + y;
}
```

[3] Immutability is not expressible in C++; `const` can always be cast away.

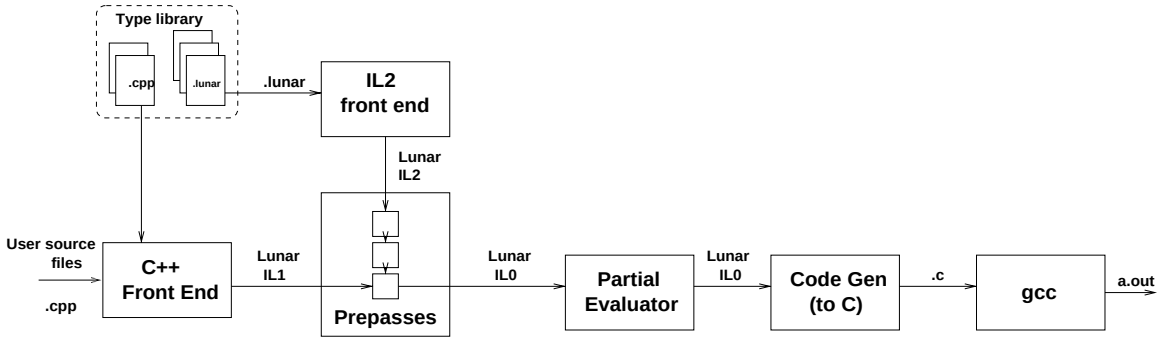

Figure 4: High-level structure of the compiler. No type analysis is done by the C++ front end; the type system is implemented by a *type library*. Type analysis results from partially evaluating the program.

```
// Base class for all types
struct type_info {
 int        type;
 int        size;
 type_info* ptrType;
};

// Primitive types
struct type_info_primitive : type_info {
 char*      name;
 int        precisionRank;
};

// Pointer types
struct type_info_ptr : type_info {
 type_info* derefType;
};

// Function pointers
struct type_info_funcptr : type_info {
 int         arity;
 type_info*  returnType;
 type_info** argTypes;
};

// Class types
struct type_info_class : type_info {
 char*             name;
 int               numBaseClasses;
 type_info_class** baseClasses;
 int*              baseClassOffsets;
 int               numFields;
 type_info_field*  fields;
 int               numMethods;
 type_info_method* methods;
};
```

```
// Class template
struct type_info_template_class
    : type_info {
 char*     name;
 int       numParameters;
 template_class_instance_list* head_instance;
};

// Instance of a class template
struct type_info_template_class_instance
    : type_info_class {
 type_info_template_class* instance_of;
 type_info**               template_parameters;
};

// Field in a class
struct type_info_field {
 char*              name;
 type_info*         type;
 int                offset;
 int                flags;
};

// Method in a class
struct type_info_method {
 char*              name;
 int                flags;
 type_info_funcptr* t;
};

// Virtual methods
struct type_info_virtual_method
    : type_info_method {
 int                vtable_offset;
};
```

Figure 5: The `type_info` hierarchy of classes which represent C++ types.

is translated as:

```
function simpleMath()
 blockscope [return$type, x, x$type, __a8, __a8$type,
    y, y$type, __a9, __a9$type, z, z$type,
    __a10, __a10$type]
  // Translation of the literal 5
  __a8 := 5
  __a8$type := int
  // Initialize x with 5
  [x, x$type] := initializerConversion(__a8,
                        __a8$type, int)
  // Initialize y with 7.0
  __a9 := 7.0
  __a9$type := double   // 7.0 is a double literal
  [y, y$type] := initializerConversion(__a9,
                        __a9$type, float)
  // Add x + y, and initialize z with the result
  [__a10, __a10$type] := plus(x, x$type, y, y$type)
  [z, z$type] := initializerConversion(__a10,
                        __a10$type, int)
```

where `initializerConversion(`*value*`,`*source-type*`,`*dest-type*`)` is a type library routine implementing C++ initializer type conversions, and `plus(..)` implements the `+` operator semantics.

In the translated code, `int` and `float` are no longer keywords representing builtin types. Instead, they are global variables declared by the type library that point to data structures representing the C++ types `int` and `float`. These data structures are instances of `type_info_primitive` (Figure 5).

This translation resembles compiling a dynamically typed language. One of the options of Lunar's C++ front end is to do just this – compile C++ to dynamically typed code – which turns out to have some interesting advantages, discussed later.

When the above code is partially evaluated, the result is:[4]

```
function simpleMath()
 blockscope [z, __a6, x2, x2__195]
  // Promote 5 to floating point
  x2__195 := _itof(5)
  // Add 5 and 7.0 in floating point
  __a6 := _f+(x2__195, 7.0)
  // Convert back to integer
  z := _ftoi(__a6)
```

The calls to the type library routines are partially evaluated and inlined, and all the `x$type` variables disappear because they are no longer needed. The primitives `itof`, `f+`, and `ftoi` turn into single machine operations to convert integer to float, add two floats, and convert a float to integer, respectively.

4.3 Function calls

Now we progress to translating functions which take arguments. We translate each C++ function into two functions:

- a *bind function*, which checks argument types and deduces template parameters for the call site; and
- an implementation function, which is a translation of the function body.

The bind function takes all the arguments and their types, whereas the implementation function takes only the arguments. As an example, consider this `cube()` function:

```
float cube(float x)
{
  return x*x*x;
}
```

The C++ front end produces a bind function `cube$bind(x,x$type)`:

```
function cube$bind(x, x$type)
 blockscope [__a1, __a1$type]
  // Check that x's type is float
  assertIsType(x$type, float)
  // Invoke implementation function
  __a1 := cube(x)
  // Return value/type pair
  return [__a1, float]
```

and an implementation function `cube(x)`:

```
function cube(x)
 blockscope [return$type, x$type, __a2, __a2$type,
    __a3, __a3$type, __a4, __a4$type]
  // Bindings for the function body
  return$type := float
  x$type := float
  // Translation of the function body
  [__a2, __a2$type] := star(x, x$type, x, x$type)
  [__a3, __a3$type] := star(__a2, __a2$type, x, x$type)
  __a4 := assignmentConversion(__a3, __a3$type, return$type)
  return __a4
```

In an implementation which supported overloading, the function `cube$bind` would dispatch to the proper implementation of `cube` based on signature. Also, note that the bind function relies only on the declaration of a function (its return type and signature) so separate compilation is possible.

4.4 Simple template functions

Next we consider translating simple template functions. As an example:

```
template<typename T1, typename T2>
T1 hypot2(T1 x, T2 y)
{
  return x*x + y*y;
}
```

Template functions are also translated into a bind function and an implementation function. The bind function is now responsible for deducing template parameters:

```
function hypot2$bind(x, x$type, y, y$type)
 blockscope [T1, T2, __a1, __a1$type]

  // Deduce the template parameters from the argument
  // types -- easy in this example
  T1 := x$type
  T2 := y$type

  // Call the implementation function; the template
  // parameters are passed as additional arguments
  __a1 := hypot2(x, y, T1, T2)
  return [__a1, T1]
```

[4] More accurately, the partial evaluator turns simpleMath() into an empty function body. To generate this code, the constants `5` and `7.0` were "lifted" out of the partial evaluator's view, and the result `z` was stored in a global variable.

and the implementation function–

```
function hypot2(x, y, T1, T2)
 blockscope [return$type, x$type, y$type, __a2, __a2$type,
    __a3, __a3$type, __a4, __a4$type, __a5, __a5$type]

  // Declare the return type, and each argument type
  return$type := T1
  x$type := T1
  y$type := T2

  // Translation of x * x
  [__a2, __a2$type] := star(x, x$type, x, x$type)

  // Translation of y * y
  [__a3, __a3$type] := star(y, y$type, y, y$type)

  // Add (x * x) + (y * y)
  [__a4, __a4$type] := plus(__a2, __a2$type,
                            __a3, __a3$type)

  // Convert result to the return type and return
  __a5 := initializerConversion(__a4, __a4$type,
                                return$type)
  return __a5
```

Note how the template parameters are passed to the implementation function as extra parameters. When partial evaluation is used to specialize the function with respect to the template parameters, these extra parameters become dead and may be eliminated.[5]

This style of translation – passing types as extra parameters – is called the *type-passing style* of compiling polymorphism [5].

4.5 Template parameter deduction

To deduce template parameters, the C++ front creates a tree representing each declared argument type, and generates code to walk these trees, checking argument types and deducing template parameters. To illustrate, here is a simple pair class:

```
template<class T1, class T2>
class pair {
public:
 T1 first;
 T2 second;
};
```

And a template function which operates on pairs:

```
template<class T1, class T2>
T1 first(pair<T1,T2>* x)
{
  return x->first;
}
```

At compile time, the type `pair<T1,T2>*` is represented by the prefix notation tree: `pointer(template(pair,T1,T2))`. The generated bind function for `first()` walks this tree, checking that `x$type` is a pointer, then checking that it is a pointer to an instance of `pair`, and so on:

```
// Bind function for  first(), illustrating
// tree walk to deduce template parameters
function first$bind(x, x$type)
 blockscope [derefType, tempparmlist, tempparm0, T1,
   tempparm1, T2, __a1, __a1$type]
  // Check that x$type is a pointer
  assertIsPtrType(x$type)
  // Get the type which x points to
  derefType := _readp(x$type, type_info_ptr.derefType)
  // Check that it is an instance of pair<>
  assertIsTemplateInstance(derefType, pair$template)
  // Get its list of template arguments
  tempparmlist := _readp(derefType,
   type_info_template_class_instance.template_parameters)
  // Get the first template argument-- this becomes T1
  tempparm0 := _readp(tempparmlist, 0)
  T1 := tempparm0
  // Get the second template argument-- this becomes T2
  tempparm1 := _readp(tempparmlist, 4)
  T2 := tempparm1
  // Call the implementation and return
  __a1 := first(x, T1, T2)
  return [__a1, T1]
```

4.6 Compiling classes

One inconvenience in compiling classes is that object layout has to be deferred until template instantiation time. A simple example illustrates why:

```
template<class T, class X>
class Foo : public T {
  X x;
  int z;
}
```

The base class of Foo is not known until instantiation time, and the offset of `z` within the object will depend on both the base class and the template type `X`. Clearly, too, `T` might inject field and method names into the environment of class Foo, so even field lookup in expressions like `a->x` must be deferred.

For this reason, `type_info_class` and its subtype `type_info_template_class_instantiation` must contain lists of base classes, fields, and methods; and uses of "`.`" and "`->`" are translated by calling type library functions which look up field names in the `type_info_class` and associated data structures (Figure 5). In other words, even parts of name analysis must be implemented by the type library and resolved by partial evaluation.

Class declarations translate into too much IL code to include here, so we look at a simple example and describe the code which is generated:

```
class B : public A, Q {
 float b;
};
```

For the class B, a global function `B$layout` is created which generates an instance of `type_info_class` (Figure 5). `B$layout()` uses type library routines to initialize the data structure with:

- an array of pointers to the base classes A and Q, and offsets of those base classes within the object layout;
- an array of fields, with names, types, and offsets for each field;
- (not implemented yet) a list of methods and vtable

[5] Lunar does not yet implement dead parameter elimination.

A global variable `B` is created and initialized to the result of calling `B$layout()`. This global is then used to represent the type of an instance of class `B`.

Translation of class templates is similar. For a class template such as

```
template<class T1, class T2>
class pair {
 T1 first;
 T2 second;
};
```

a function `pair$layout(T1,T2)` is created which returns an instance of `type_info_template_class_instance` (Figure 5). Passing the template parameters `T1` and `T2` as arguments gives enough information to calculate the object layout. A `type_info` record for a class template also contains an array of template arguments and a pointer to the uninstantiated template type.

A global function `pair(T1,T2)` is created which checks if the template instance `pair<T1,T2>` already exists, and if not instantiates it by calling `pair$layout(T1,T2)`. In the functional-language world, `pair()` would be regarded as a *type constructor*: given types `T1` and `T2`, it constructs a type `pair(T1,T2)`.

Field accesses are translated into a call to a type library routine. For example, in this code:

```
pair<int,float>* z = new pair<int,float>;
z->first = 3;
```

the expression `z->first` is translated as:

```
  __a5 := 3
  __a5$type := int
  storeFieldPtr(z, z$type, "first", __a5, __a5$type)
```

where `storeFieldPtr(`*x, xtype, name, y, ytype*`)` finds the field *name* in class *xtype*, and stores *y* there.

4.7 Bootstrapping the type system

It would be nice to implement the type system entirely in C++. To see why this is hard, consider this example: the compiler often needs to assert that a type is a pointer type. It does this by inserting a call to a type library routine `assertIsPointer()`.

Suppose this assertion function were written in C++:

```
void assertIsPointer(type_info* t)
{
  if (t->type == type_pointer)
    return;

  lunar_type_error("Expected a pointer type here");
}
```

The C++ front end has to translate this code into intermediate language. In translating the expression `t->type`, the C++ front end will insert a check that `t` is in fact a pointer type:

```
function assertIsPointer(t)
     .
     .
  // Translation of t->type
  assertIsPointer(t)     // Infinite recursion!

  // Get the type_info* for the dereferenced type
  derefType__4 := readp(t, type_info_ptr.derefType)

  // Read the field called "type"
  [__a3, __a3$type] := readField(t, derefType__4, "typeype")
```

So the first time `assertIsPointer()` is invoked, the program will go into an infinite recursive loop. To avoid such problems, this bootstrapping approach is used:

- Layout of `type_info` classes is done in compiler. There are $\sim$ 100 lines of code in the front end which recognize that a class declaration being processed is a `type_info` class, and determine the object layout (for other classes, object layout is implemented by a type library routine). This code handles very restricted objects: only one base class may be specified, and fields may only be builtin types or pointers.
- Layout of other classes is deferred by generating code as described in Section 4.6.
- Primitive types and structure operations (field accesses, etc.) are implemented in type system modules, and written in intermediate language IL_2.
- Higher-level language features may be implemented in C++, as long as they do not use their own language feature in the implementation. Lunar's C++ front end has just reached the point where it has become possible to implement type library routines in C++, and there is nothing to show yet.

4.8 Reporting type errors

One problem which arises with this approach to compilation is reporting errors: when a type error is found during partial evaluation, can it be presented in a sensible way to the user?

Lunar religiously maintains pointers to the front end AST representations throughout partial evaluation. Errors found during partial evaluation are passed to the appropriate front end AST node to be reported. IL_0 includes a primitive `error()` which, if encountered during partial evaluation, forces an error to be reported. This primitive is used in the type library to report type errors. One problem with the current implementation is that type errors are reported at their location in the type system library. It should be straightforward for the partial evaluator to also provide the front end with a specialization stack (analogous to a call stack), so the location of the error in user code can be determined.

4.9 Register selection

Another problem caused by Lunar's approach is that the intermediate language is typeless. Hence the back end does not know whether a variable should be in an integer or floating-point register. The back end uses the source pointers to ask the front end to recommend register types. This works fine if the front end has done type analysis (as is the case for Lunar's Java front end), but Lunar's C++ front end does not do type analysis.

This is likely solvable by having the partial evaluator feed information about eliminated variables to the front end; for example, when the partial evaluator discovers that `x$type` is `float`, it can pass this information to the front end, which holds onto it and uses it to recommend register types for the back end.

5 Freebies

Before discussing the compilation models which Lunar's approach allows for C++ templates, we examine two useful features which result from having a type system library.

5.1 Reflection– for free

One of the obvious benefits of taking this approach to compiling C++ is that an implementation could expose the information in the `type_info` classes, giving reflection "for free". Being able to step through and examine the fields of an arbitrary class is very useful for implementing features like persistence and remote method invocation.

5.2 Virtual function resolution– for free

Although not yet implemented, there is good reason to believe that this style of implementation will resolve some virtual functions at compile time. In Lunar's Java front end, which uses a conventional type system implementation, the partial evaluator is able to propagate the names of virtual functions through the vtable in some circumstances, turning dynamic dispatch into static dispatch. The C++ front end could use the partial evaluator to similar benefit.

6 Five compilation models

So far, we have described the C++ front end as if we wanted to duplicate the usual compilation model for C++ templates, in which all template parameters are determined at compile time, and template functions are specialized based on type.

By making simple changes to the partial evaluator's behavior, we can achieve a range of compilation models, each with a different tradeoff of code size, code speed, and time to compile (Figure 6).

Some of these models are not standard compliant, because type-checking of some template code is deferred until run-time. These non-compliant models are proposed as additional models to be provided by a C++ compiler, rather than as a replacement for the standard template compilation model.

6.1 Dynamic typing (-T0)

An obvious thing to do is not run the partial evaluator at all. The type system library is then part of the application at run-time, and type analysis is done on the fly as the C++ program executes.

This model allows rapid compiles: there is no template instantiation or even type checking done. It may even be possible to compile headers and source files separately, provided a parser symbol table and preprocessor state are dumped for each header file.

With dynamic typing, the program runs slower[6] and all type errors are detected at run time. Even the most egregiously flawed C++ program will compile, so long as it can be parsed. Our current implementation handles run-time type errors by issuing an error message and throwing an exception, which is typically uncaught and aborts the program.

[6] All the type analysis is being done dynamically, and all values are boxed, which slows down execution substantially.

The dynamic typing model is suitable for scripting. Lunar's C++ front end generates IL_0, which is easy to interpret.[7] A scripting front end for C++ would read single statements, convert them to IL_0, and interpret them.

One interesting application of the dynamic typing model is debugging thorny template instantiation errors. Template-heavy libraries are notorious for generating bizarre compile errors several template instantiations deep into the library. Using dynamic typing, one can load up the debugger and run the program until the type error occurs. Figure 7.2 shows a debugger session using the Data Display Debugger [16] to view a template instantiation bug in dynamically typed C++ code compiled by Lunar. When the type error is encountered, the debugger halts execution and the user can browse the backtrace of template functions, examining the template parameter types for each call site. A function in the type library called `displayType()` can be used to show pretty-printed versions of template parameter types, and the brave can use DDD's data structure viewer used to examine the `type_info` structures for template parameters.

6.2 Dynamic typing for template code, static typing otherwise (-Tfast)

This model may be summarized as "genericity without code bloat." Partial evaluation is used to statically type non-template code, but the partial evaluator is not permitted to specialize template functions. This results in static typing being used for non-template code, and dynamic typing being used for template code.[8]

Non-template code will execute quickly, while template code may be substantially slower. Since template functions are not specialized, there is only one dynamically typed version of each template function, resulting in smaller code size.

Separate compilation is possible in this model, since template instantiation is deferred until run time. It is also possible to build binary libraries – even shared libraries – containing uninstantiated templates. This is partway to a solution for the problem of distributing commercial template libraries without revealing source code.

6.3 Monovariant specialization: dynamic typing only to avoid code duplication (-Tmono)

With a closed program assumption and whole-program analysis, it is possible to generate one version of each template function that is specialized *as much as possible* given the ways in which it is used. If at all call sites, a template parameter has the same type, then the template function may be specialized based on that template parameter.

For example, if a program used only `Array<int>`, the `Array<>` methods would be specialized for `int`. However, if the program used both `Array<int>` and `Array<float>`, the methods of `Array` would be dynamically typed.

This model corresponds to *monovariant specialization* in the partial evaluation world, meaning literally: one (mono) variant of each function is allowed.

[7] Lunar has an an interpreter for an older version of IL_0, and updating this for the latest version would be straightforward.

[8] This model is not yet working in Lunar. Of the models described, -T0 and -Tstd work, and the -Tfast, -Tmono and -Tprof models are still being implemented.

Option	Model	Benefits	Templates are instantiated at ...	Non-template code is typed at ...
`-T0`	Dynamic typing	scripting, debugging template instantiation	run-time	run-time
`-Tfast`	Template code is dynamically typed; non-template code is statically typed	fast compile, can build binary libraries containing templates	run-time	compile-time
`-Tstd`	Standard template compilation; everything is statically typed	fast execution, standard compliance	compile-time	compile-time
`-Tmono`	Up to one instance of each template is allowed; otherwise templates are dynamically typed	small code size	mixed	compile-time
`-Tprof`	Profile-guided template instantiation: templates are instantiated only in performance-critical regions	small code size and fast execution	mixed	compile-time

Figure 6: Five compilation models for C++ templates

In this model, there is no code bloat, and there may be some performance benefit from whatever specialization occurs. Doing monovariant specialization requires an iterative closed program analysis, possibly impractical for large C++ applications.

6.4 Standard model: polyvariant specialization (-Tstd)

This model corresponds to the usual C++ template compilation model. Function templates are specialized based on template parameter types. This corresponds to *polyvariant specialization* in the partial evaluation world.

The model is standard-compliant, and suffers from the traditional problems of C++ compilers: separate compilation requires either a disk cache of template instantiations, or discarding duplicate instantiations at link time; or a whole-program analysis is required to do instantiation at link time.

It also shares the convergence problems of typical C++ compilers. Whether a template instantiation chain will converge is undecidable; this is a property of type systems such as C++'s, which allow types to depend on values [1]. Heuristics are required to halt specialization when "too many" specializations have been generated. One advantage of a Lunar-style approach is that rather than halting compilation in such a situation, the compiler can issue a warning and defer instantiating the rest of the templates until run time.

6.5 Profile-guided template instantiation (-Tprof)

This model is a straightforward application of profile feedback; a very similar technique – controlling procedure cloning using profile feedback – is described in [15].[9]

A program is first compiled using another compilation model and executed with a profiler. The profiler is used to generate a list of inclusive time for each function.[10] The program is then compiled again with -Tprof. As the partial evaluator encounters each template function call site, it decides whether to specialize the function or not based on what percentage of time was spent in that function template and its descendents in the call graph.

The effect is to instantiate templates only in performance critical regions – say, the functions that consume more than 10% of the run time – and avoid instantiating templates where it would convey no performance advantage. This provides a controllable tradeoff between efficiency and code growth.

7 Discussion

7.1 Performance of the compiler

There remain serious unanswered questions about the compile times required by this compilation model. As currently implemented, the type system is effectively "interpreted" at compile time by the partial evaluator. A redeeming quality is that the partial evaluator caches specializations; for example, the first time `+` is used to add two numbers, the type library routine `plus` and several others have to be interpreted to handle arithmetic type promotions and instruction selection. However, the next time two numbers of the same type are added, the specialized version of `plus` is retrieved from the cache and inlined (this is called *memoization* in the functional world).

Another aspect of Lunar's C++ front end which hurts performance is that it uses extensional equality to compare types.[11] It is possible that a better alias analysis would allow the use of intensional equality for type comparisons.

In theory, some of the interpretation overhead could be removed by using a partial evaluation trick: the second Futamura projection [4] suggests that the partial evaluator could be specialized with respect to the type system library. Given the difficulties encountered in just getting the partial evaluator to optimize away the type system library (so far ≈ 1000 lines of code) the prospect of achieving this on the entire

[9] Lunar does not yet provide this model, but it appears to be straightforward to implement.

[10] Inclusive time is time spent in a function and all the functions it calls.

[11] In other words, two types are equal if their `type_info` data structures are recursively equal. *Intensional* equality refers to comparing pointers.

compiler system automatically ($\approx$ 70000 lines of code) appears remote.

A more plausible approach is to translate the type library to IL_0, then write a translator to convert IL_0 into code which operates over the partial evaluator's representation of values, heap states, and trees. This code could then (in theory) be linked into the compiler. The type system would then be executing as native code in the compiler, and the main performance disadvantage would be using the partial evaluator's representation of the heaps and values rather than operating on native data representations.

7.2 Related work

The existence of a relationship between C++ templates and partial evaluation was first proposed by Salomon [8], who proposed a dialect of C using partial evaluation to achieve C++ template-like capabilities. The relationship was later explored in detail by [13]. This paper goes well beyond these two by proposing a concrete compilation process which uses partial evaluation to drive type analysis.

Staging [7, 12] is the general notion of splitting a computation into several stages, each of which perform part of the computation and produce a residual to be executed in the next stage. Dynamic typing of statically typed code, as used in some of Lunar's template compilation models, may be thought of as staged type analysis [10]. Lunar handles template parameters using a type-passing style similar to that of [5].

Lunar initially converts C++ to dynamically typed code, so there is some resemblance between this approach and soft typing [2], which is used to turn dynamically typed code into statically typed code. Soft typing is a specialized analysis to recover types, and does not use partial evaluation to drive type analysis as Lunar does.

In a tenuous way, Lunar's approach to compiling C++ can be viewed as a flavour of attribute grammar computations. Lunar only worries about a single attribute – types – but one can imagine embedding other attributes in the intermediate language as well, by inserting `x$attr` variables and appropriate calls into an attribute computation library. Partial evaluation can then be viewed as staging the attribute computation so that only values are computed at run-time.

Lunar's approach to compiling C++ may also be thought of as an embedded type system [14], in the sense that the type semantics of C++ are embedded in the semantics of Lunar's intermediate language.

References

[1] Augustsson, L. Cayenne – a language with dependent types. *ACM SIGPLAN Notices 34*, 1 (Jan. 1999), 239–250.

[2] Cartwright, R., and Fagan, M. Soft typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* (Toronto, ON, Canada, June 1991), B. Hailpern, Ed., ACM Press, pp. 278–292.

[3] Consel, C. Binding time analysis for higher order untyped functional languages. In *1990 ACM Conference on Lisp and Functional Programming* (June 1990), ACM, ACM Press, pp. 264–272.

[4] Futamura, Y. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls 2*, 5 (1971), 45–50.

[5] Harper, R., and Morrisett, G. Compiling polymorphism using intensional type analysis. In *Principles of Programming Languages* (San Francisco, Jan. 1995).

[6] Jones, N. D. An introduction to partial evaluation. *ACM Computing Surveys 28*, 3 (Sept. 1996), 480–503.

[7] Jorring, U., and Scherlis, W. L. Compilers and staging transformations. In *POPL'86* (1986), pp. 86–96.

[8] Salomon, D. J. Using partial evaluation in support of portability, reusability, and maintainability. In *Compiler Construction '96* (Linköping, Sweden, 24–26 Apr. 1996), pp. 208–222.

[9] Sarkar, V., and Knobe, K. Enabling sparse constant propagation of array elements via array SSA form. *Lecture Notes in Computer Science 1503* (1998), 33–??

[10] Shields, M., Sheard, T., and Jones, S. P. Dynamic typing as staged type inference. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, 19–21 Jan. 1998), pp. 289–302.

[11] Steensgaard, B. Sparse functional stores for imperative programs. In *ACM SIGPLAN Workshop on Intermediate Representations (IR'95)* (Jan. 1995), vol. 30 (3) of *SIGPLAN Notices*, ACM Press, pp. 62–70.

[12] Taha, W., and Sheard, T. Multi-stage programming with explicit annotations. *ACM SIGPLAN Notices 32*, 12 (1997), 203–217.

[13] Veldhuizen, T. L. C++ templates as partial evaluation. In *Proceedings of PEPM'99, The ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, ed. O. Danvy, San Antonio, January 1999.* (Jan. 1999), University of Aarhus, Dept. of Computer Science, pp. 13–18.

[14] Wand, M. Embedding type structure in semantics. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages* (New Orleans, LS, Jan. 1985), B. K. Reid, Ed., ACM Press, pp. 1–6.

[15] Way, T., and Pollock, L. Using path spectra to direct function cloning. In *Workshop on Profile and Feedback-Directed Compilation* (1998).

[16] Zeller, A., and Lütkehaus, D. DDD - A free graphical front-end for UNIX debuggers. *ACM SIGPLAN Notices 30*, 12 (Dec. 1995).

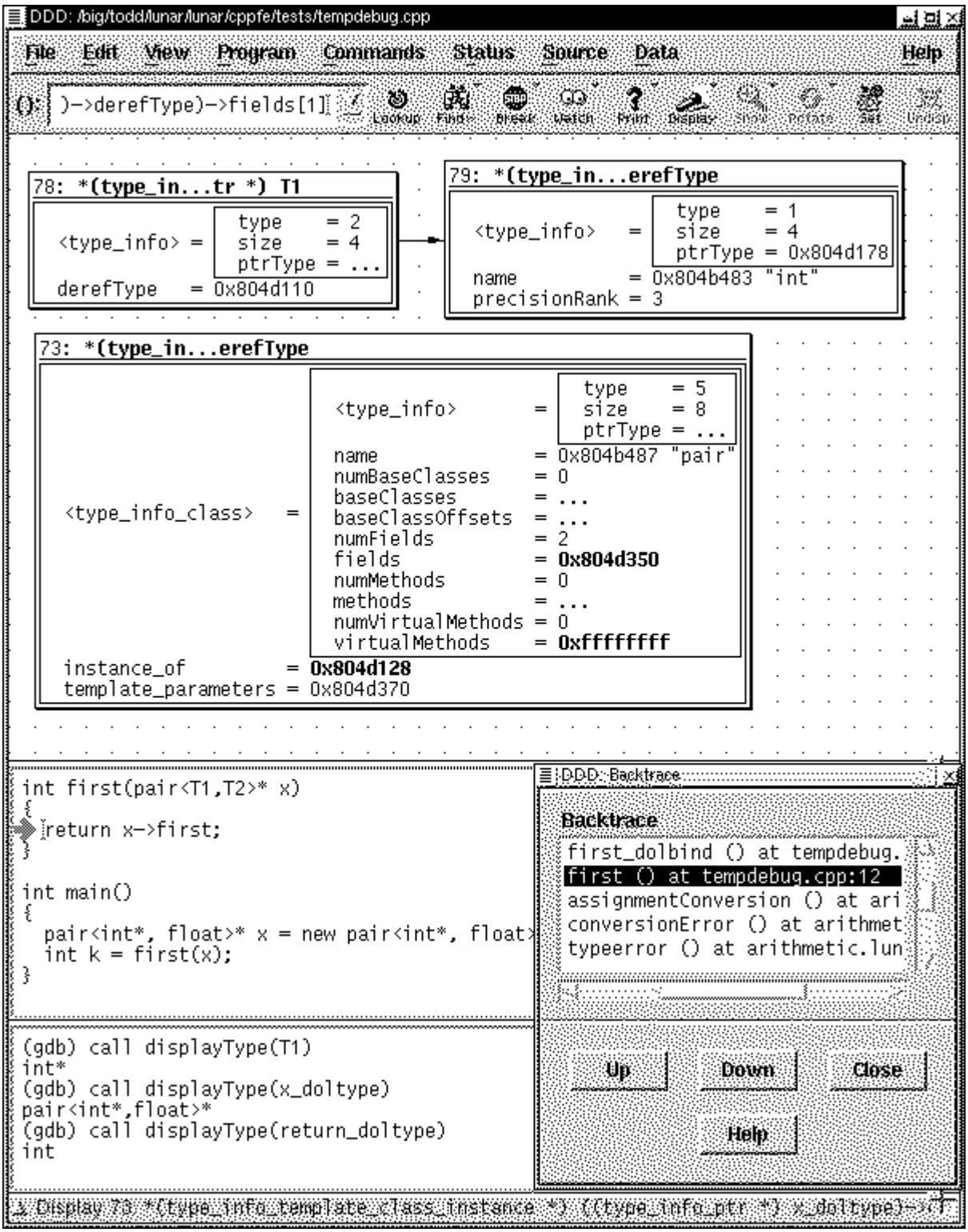

Figure 7: Using the Data Display Debugger [16] to debug template instantiation. The data window (top) is showing the `type_info` structures for: (78) `T1=int*` (79) `int` (73) `pair<int*,float>`. The data display expands and collapses by clicking to reveal fields, base classes, and other information about types. In the GDB console (bottom), the user can call the lunar routine `displayType()` to show "pretty-printed" versions of types.