C++ Template Lisp Interpreter

Spencer Tipping

July 6, 2010

Contents

1 Introduction to Template Patterns

Template expansion provides a pure untyped lambda calculus. All equality is extensional and the calculus supports higher-order functions (templates) with annotations at invocation, but not declaration, time.[1] This section goes over the encoding of lambda calculus in the template system.

1.1 Encoding constants

Constants are simply `structs`, `classes`, or other types that don't take template parameters. It isn't a problem if they do take template parameters, of course;

[1] This makes it untyped. C++ templates also support function types using nested template syntax – see section 1.4.

those will simply be specified later once the constant has propagated through the lambda expansion.

Listing 1 examples/constants.h

```
// Defining a constant term
struct foo {
  enum {value = 10};
};

// Defining a global constant also_foo = foo
typedef foo also_foo;

// Defining two templated terms that act as constants
template<class t> struct has_a_field {
  t field;
};

template<int n> struct has_a_number {
  enum {number = n};
};
```

1.2 First-order function encoding

The idea is to have structs that represent terms of the calculus. If they are templates, then they represent functions (which are also terms). For example:

Listing 2 examples/first-order-functions.cc

```
#include <iostream>
#include "constants.h"

// Defining the identity function, where the result
// can be retrieved by specifying bar<T>::type
template<class t> struct bar {
  typedef t type;
};

// Defining a global constant identity_result = bar(also_foo)
// This is analogous to the code 'let identity_result = bar also_foo'
// in Haskell.
typedef bar<also_foo>::type identity_result;

int main () {
  std::cout << "foo::value            = " <<
               foo::value               << std::endl <<
              "also_foo::value       = " <<
```

```
                also_foo::value           << std::endl <<
               "identity_result::value = " <<
                identity_result::value    << std::endl;
}
```

In practice there is some difficulty already. Notice the use of `::type` to retrieve the value of a function application. This slot had to be assumed by the caller; it is analogous to JavaScript code like this:

Listing 3 `examples/unfriendly-identity.js`

```
// An unfriendly identity function.
var identity = function (x) {
  return {type: x};
};

// Invocations must now look like this:
var y = identity(x).type;
```

Having issues like this percolating through the design can be a real problem. Unless the slot is passed to every invocation site,[2] invocations will be divergent and will create errors. This means that return values should be unified to a single slot, in this library (and the Boost MPL) called `::type`.[3]

So we establish some conventions up front. Whenever you define a constant, it is used as-is without a contained `typedef` that we have to know about. This is OK because we shouldn't ever make assumptions about the members of types that are used as template parameters.

1.3 Higher-order function encoding

Higher-order functions are possible by encoding slots for invocations.[4] We do this by declaring another template inside the first:

Listing 4 `examples/higher-order-functions.cc`

```
#include <iostream>
#include "constants.h"

// Defining the K combinator
template<class t>
```

[2] It also must be forwarded, which isn't possible in C++ to the best of my knowledge.

[3] This may seem counter-intuitive, since the types here encode values in lambda-calculus. However, it does serve a mnemonic purpose later when value types are used as template parameters, and dependent value-type relations are established. Once this happens it becomes useful to explicitly distinguish between type template parameters and value template parameters.

[4] This is equivalent to the distinction between pure, extensional object-oriented programming and pure, extensional functional programming. In the latter, term juxtaposition (e.g. `f x`) constitutes invocation of the default slot, generally referred to as *apply*. In the former, slots are explicitly named, as would be the case in a language such as Java – thus juxtaposition has no meaning on its own.

```
struct k {
  template<class u>
  struct apply {
    typedef t type;
  };
};

// Using that on two types
typedef has_a_number<5> t1;
typedef has_a_number<6> t2;
typedef k<t1>::apply<t2>::type should_be_t1;

int main () {
  std::cout << "t1::number           = " << t1::number           << std::endl <<
               "should_be_t1::number = " << should_be_t1::number << std::endl;
}
```

In this example, `k` has a call slot `apply` that ultimately provides the value. So, for example, `k<x>::apply<y>::type` is equivalent to the more concise `k x y` in Haskell, or `((k x) y)` in Scheme.

At this point it should be clear that nothing is standardized here. Top-level functions are invoked directly, whereas returned functions use `::apply<x>`. Type results from template invocations are accessed as `::type`. One way to go about fixing it is to make a rule that a function gets encoded a bit less directly:

Listing 5 `examples/indirect-functions-broken.cc`

```
#warning This example is deliberately broken, so a compilation error is normal.

// Encoding the K combinator uniformly, but with compile errors
struct k {
  template<class t>
  struct apply {
    template<class u>
    struct apply {
      typedef t type;
    };
  };
};
```

However, if you compile it you get an error stating that you can't define a nested `struct` with the same name as the outer one. The solution is to use an intermediate `::type` dereference to wrap the inner `::apply<x>`.

Listing 6 `examples/indirect-functions.cc`

```
#include <iostream>
#include "constants.h"

```

```
// Encoding the K combinator uniformly
struct k {
  template<class t>
  struct apply {
    struct type {
      template<class u>
      struct apply {
        typedef t type;
      };
    };
  };
};

typedef has_a_number<5> t1;
typedef has_a_number<6> t2;
typedef k::apply<t1>::type::apply<t2>::type should_be_t1;

int main () {
  std::cout << "t1::number           = " << t1::number           << std::endl <<
               "should_be_t1::number = " << should_be_t1::number << std::endl;
}
```

At this point a nice pattern emerges. Whenever we apply a function to something, we get its `::type` as well. So constants map to themselves, and function invocations are all of the form `f::apply<...>::type`.

1.4 Higher-order function type signatures

C++ lets you specify type signatures for higher-order templates. This can be useful to ensure that a function possesses at least a certain Church arity[5] or takes at least so many arguments. It also provides some notational convenience at invocation-time.

Here is the Haskell function that we will model in C++ templates:

Listing 7 `examples/apply-two-function.hs`

```
apply_two :: ((a, a) -> b) -> a -> b
apply_two f x = f (x, x)
```

In template metaprogramming it isn't possible to express the constraints about values, but we can express constraints about arity and function status:

Listing 8 `examples/apply-two-function.cc`

```
#include <iostream>
#include "constants.h"
```

[5] I use this term to refer to the arity of the uncurried form of the function. For example, the Church arity of $\lambda x.\lambda y.x$ is 2, since uncurrying yields $\lambda(x, y).x$.

```
3
4  // Encoding the type signature as a template parameter specification
5  struct apply_two {
6    template<template<class arg1, class arg2> class f>
7    struct apply {
8      struct type {
9        template<class x>
10       struct apply {
11         typedef f<x, x> type;
12       };
13     };
14   };
15 };
16
17 // An example value for f
18 template<class x, class y>
19 struct sample_f {
20   typedef x x_type;
21   typedef y y_type;
22 };
23
24 typedef has_a_number<10> t1;
25 typedef has_a_number<12> t2;
26 typedef apply_two::apply<sample_f>::type::apply<t1>::type two_of_t1;
27
28 int main () {
29   std::cout << "t1::number                = " << t1::number << std::endl <<
30                "two_of_t1::x_type::number = " << two_of_t1::x_type::number << std::endl <<
31                "two_of_t1::y_type::number = " << two_of_t1::y_type::number << std::endl;
32 }
```

The parameter definition `<template<class arg1, class arg2> class f>` is equivalent to the Haskell type signature `f :: (a, b) -> c`; none of the individual types are specified, but the template must be invoked on two parameters or not invoked at all.[6] The other thing of note is that you can arbitrarily refine the left-hand side; for example:

```
template<template<template<class x> class f,
                  template<class y> class g> class compose>
struct composer {...};
```

This is equivalent to `composer :: ((a -> b), (c -> d)) -> e`. As far as I know there is no way to specify anything about the return type of a function using template syntax.

[6] Note that at this point I'm not referring to invocation using the `::apply` convention established earlier. This invocation is just regular template expansion.

I'm not using template types in this project for a couple of reasons. First, declaring formals uses names (I'm actually not sure whether those names are considered reserved by C++, but I assume so). Second, it isn't possible to encode slot types, and all invocation in the lambda-calculus encoding is done with the `::apply` slot.

1.5 Conditionals

Templates don't model conditionals *per se*. Rather, you can create conditionals by using pattern matching and explicit specialization. There are some weird limitations about this, but here is the basic idea:

Listing 9 examples/specialization.cc

```
#include <iostream>
#include "constants.h"

// General case
template<class t>
struct piecewise {
  typedef t type;
};

// When t = has_a_number<50>, do this instead
template<>
struct piecewise<has_a_number<50>> {
  typedef has_a_number<100> type;
};

typedef piecewise<has_a_number<3>>::type  general;
typedef piecewise<has_a_number<50>>::type specialized;

int main () {
  std::cout << "general::number     = " << general::number     << std::endl <<
               "specialized::number = " << specialized::number << std::endl;
}
```

1.5.1 Limitations of inner specialization

Because terminal (i.e. non-expanding) types are extensionally equivalent, pattern matching can be used to reliably specialize template expansions. The only case where this doesn't work is inside a class:

Listing 10 examples/inner-specialization-broken.cc

```
#warning This example is deliberately broken, so a compilation error is normal.

```

```
template<class t>
struct container {
  template<class u>
  struct piecewise {
    typedef u type;
  };

  // Compiler complains about this:
  template<>
  struct piecewise<int> {
    typedef t type;
  };
};

typedef container<int>::piecewise<int>::type foo;
```

The solution to this problem is to break the inner class outside of the outer one and uncurry its arguments:

Listing 11 examples/inner-specialization.cc

```
#include <iostream>
#include "constants.h"

namespace inside_container {
  template<class t, class u>
  struct piecewise {
    typedef u type;
  };

  template<class t>
  struct piecewise<t, int> {
    typedef t type;
  };
}

template<class t>
struct container {
  template<class u>
  struct piecewise {
    typedef typename inside_container::piecewise<t, u>::type type;
  };
};

typedef has_a_number<1> t1;
typedef has_a_number<2> t2;
typedef container<t1>::piecewise<t2>::type  general;
```

```
typedef container<t1>::piecewise<int>::type specialized;

int main () {
  std::cout << "general::number     = " << general::number     << std::endl <<
               "specialized::number = " << specialized::number << std::endl;
}
```

The namespace here isn't necessary, but it hides what normally gets hidden when you create an anonymous closure. Also notice that it must be declared before `container` due to C++'s forward-reference semantics.[7]

1.6 Makefile for examples

This makefile will build all of the examples listed in section 1. It is available in the source distribution as `src/makefile`, and should be run from inside `src`. Note that it requires your version of `gcc` to support a fairly recent C++0x draft, as specified by `-std=gnu++0x`.

Listing 12 `examples/makefile`

```
WORKING = first-order-functions higher-order-functions indirect-functions \
          apply-two-function specialization inner-specialization
ERRORS  = indirect-functions-broken inner-specialization-broken

CC      = g++
CC_OPTS = -g -Wall -std=gnu++0x

all: $(WORKING)
broken: $(ERRORS)

.PHONY: clean
clean:
        rm -f $(WORKING) $(ERRORS)

%: %.cc
        $(CC) $(CC_OPTS) $< -o $@ || true
```

2 Preprocessor Definitions

The goal of this section is to define a small domain-specific language for building functions and defining variables. The basic operations are these:

1. Defining a global constant
2. Defining a local variable

[7] Despite this, template expansion is generally lazy in other ways.

3. Applying a function to an expression
4. Returning a value from a function
5. Defining a function
6. Defining a closure

Each of these operations has a roughly standard form.[8] The definition of a shorthand for each is given in the following subsections.

2.1 Preprocessor limitations

One notable limitation of the C preprocessor when working with templates is that it doesn't treat < and > as nested parentheticals.[9] This has the unfortunate effect of splitting on commas that separate template parameters, e.g.:

```
#define f(x) ...

template<class x, class y>
struct foo {...};

f(foo<bar, bif>);   // Error here; too many arguments
```

In value-space you can deal with this by explicitly parenthesizing expressions, but parentheses aren't a transparent construct in type-space. The way I'm getting around this for now is to make macros variadic and always put any template invocations at the end:

```
#define f(x...) typedef x bar;
```

This will preserve the commas in the original expression.

Another solution is to define a macro that preserves commas in its arguments, and then wrap any comma-laden expressions with it:

```
#define commas(args...) args
f(commas(foo<bar, bif>));
```

[8] There are two for item 3; one form is used outside of a function body and the other is used inside. The difference has to do with disambiguating template expansions and is covered in section ??.

[9] It wouldn't be possible for it to do the right thing here anyway, since the preprocessor sees code before types have been defined. However, template punctuation was an unfortunate choice considering its ambiguity with relational operators, and the fact that relational operators are always ungrouped while template parameter delimiters are always grouped.

2.2 Defining a global constant

This is the simplest operation. It involves creating a `typedef` without a scoping label, for example `typedef foo bar`:

Listing 13 `preprocessor/global.h`

```
#define global(name, value...) typedef value name;
```

Note that the order of arguments is reversed from a `typedef`. This is both necessary and convenient; it is convenient because you generally define variables by specifying the name first, and necessary because the variadic parameter should come last.

2.3 Defining a local variable

Inside a `struct` the visibility of `typedef`s can be changed by using `public:`, `private:`, etc. Because local variables have no external visibility, they are defined with the `private:` modifier.

Listing 14 `preprocessor/local.h`

```
#define local(name, value...) private: typedef typename value name;
```

2.4 Applying a function to an expression

The syntax for this varies depending on whether you are inside or outside a template. If you are inside a template, some disambiguation needs to happen regarding template expansion and namespace lookups. For example, this global `typedef`:

```
typedef foo<x, y>::bar<y>::type bif;
```

would be translated into this local `typedef`:

```
typedef typename foo<x, y>::template bar<y>::type bif;
```

As such, there are two different versions of this macro. One, `global_apply`, should be used at the top level, while the other, `apply`, should be used inside classes. The insertion of `typename` is handled by local variable definition and the return construct.

Listing 15 `preprocessor/apply.h`

```
#define global_apply(lhs, args...) lhs::apply<args>::type
#define apply(lhs, args...) lhs::template apply<args>::type
```

2.5 Defining a function

Encoded functions generally look like this:

```
template<parameters>
struct apply {
  ...
  typedef ... type;
};
```

There is a challenge here, however. As mentioned in section 1.5.1, closures can't be encoded in a purely nested way. Rather, inner functions must be explicitly uncurried and declared beforehand. So the full form of expansion looks like this:

```
namespace _internals {
  // dependent closures go here
}

struct function_name {
  template<parameters>
  struct apply {
    ...
    typedef ... type;
  };
};
```

The way I'm going to deal with this for now is to require closures to both be named, and to specify the variables they close over. In other words, the uncurrying is done manually and not by the macro layer.[10]

Listing 16 preprocessor/function.h

```
1  // Protect from argument separation in the preprocessor.
2  #define v(xs...) xs
3
4  // Define a named expanding slot.
5  #define slot(name, parameters) \
6    template<parameters> struct name
7
8  // Define a function that can have specialized closures.
9  #define function(name, cases...) \
10   namespace name##_internals { cases } \
11   struct name { typedef name##_internals::apply apply; }
12
13 // Define either a general or specialized case.
14 #define when(parameters) slot(apply, parameters)
```

[10] It may be possible to define a smarter set of macros to do this properly, so this section may change in the future.

2.6 Defining a closure

Using the definitions above, closure definitions