

C++ Template Lisp Interpreter

Spencer Tipping

July 6, 2010

Contents

1	Introduction to Template Patterns	1
1.1	Encoding constants	1
1.2	First-order function encoding	2
1.3	Higher-order function encoding	3
1.4	Higher-order function type signatures	5
1.5	Conditionals	7
1.5.1	Limitations of inner specialization	7
1.6	Makefile for examples	9
2	Representation of Lambda Calculus Constructs	9
3	Preprocessor Definitions	10
3.1	Preprocessor limitations	10

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

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

Listing 1 examples/constants.h

```
1 // Defining a constant term
2 struct foo {
3     enum {value = 10};
4 };
5
6 // Defining a global constant also_foo = foo
7 typedef foo also_foo;
8
9 // Defining two templated terms that act as constants
10 template<class t> struct has_a_field {
11     t field;
12 };
13
14 template<int n> struct has_a_number {
15     enum {number = n};
16 };
```

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

```
1 #include <iostream>
2 #include "constants.h"
3
4 // Defining the identity function, where the result
5 // can be retrieved by specifying bar<T>::type
6 template<class t> struct bar {
7     typedef t type;
8 };
9
10 // Defining a global constant identity_result = bar(also_foo)
11 // This is analogous to the code 'let identity_result = bar also_foo'
12 // in Haskell.
13 typedef bar<also_foo>::type identity_result;
14
15 int main () {
16     std::cout << "foo::value          = " <<
17                 foo::value          << std::endl <<
18                 "also_foo::value     = " <<
19                 also_foo::value     << std::endl <<
20                 "identity_result::value = " <<
21                 identity_result::value << std::endl;
```

22 }

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`

```
1 // An unfriendly identity function.
2 var identity = function (x) {
3   return {type: x};
4 };
5
6 // Invocations must now look like this:
7 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`

```
1 #include <iostream>
2 #include "constants.h"
3
4 // Defining the K combinator
5 template<class t>
6 struct k {
7   template<class u>
8   struct apply {
```

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

³This 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.

⁴This 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.

```

9     typedef t type;
10 };
11 };
12
13 // Using that on two types
14 typedef has_a_number<5> t1;
15 typedef has_a_number<6> t2;
16 typedef k<t1>::apply<t2>::type should_be_t1;
17
18 int main () {
19     std::cout << "t1::number          = " << t1::number          << std::endl <<
20                 "should_be_t1::number = " << should_be_t1::number << std::endl;
21 }

```

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

```

1 #warning This example is deliberately broken, so a compilation error is normal.
2
3 // Encoding the K combinator uniformly, but with compile errors
4 struct k {
5     template<class t>
6     struct apply {
7         template<class u>
8         struct apply {
9             typedef t type;
10         };
11     };
12 };

```

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

```

1 #include <iostream>
2 #include "constants.h"
3
4 // Encoding the K combinator uniformly
5 struct k {
6     template<class t>

```

```

7   struct apply {
8       struct type {
9           template<class u>
10              struct apply {
11                  typedef t type;
12              };
13      };
14  };
15  };
16
17  typedef has_a_number<5> t1;
18  typedef has_a_number<6> t2;
19  typedef k::apply<t1>::type::apply<t2>::type should_be_t1;
20
21  int main () {
22      std::cout << "t1::number          = " << t1::number          << std::endl <<
23                  "should_be_t1::number = " << should_be_t1::number << std::endl;
24  }

```

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 arity⁵ 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

```

1  apply_two :: ((a, a) -> b) -> a -> b
2  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

```

1  #include <iostream>
2  #include "constants.h"
3
4  // Encoding the type signature as a template parameter specification
5  struct apply_two {

```

⁵I 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 $\lambda(x,y).x$.

```

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.

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

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

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

```
1 #include <iostream>
2 #include "constants.h"
3
4 // General case
5 template<class t>
6 struct piecewise {
7     typedef t type;
8 };
9
10 // When t = has_a_number<50>, do this instead
11 template<>
12 struct piecewise<has_a_number<50>> {
13     typedef has_a_number<100> type;
14 };
15
16 typedef piecewise<has_a_number<3>>::type general;
17 typedef piecewise<has_a_number<50>>::type specialized;
18
19 int main () {
20     std::cout << "general::number    = " << general::number    << std::endl <<
21               "specialized::number = " << specialized::number << std::endl;
22 }
```

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

```
1 #warning This example is deliberately broken, so a compilation error is normal.
2
3 template<class t>
4 struct container {
5     template<class u>
```

```

6     struct piecewise {
7         typedef u type;
8     };
9
10    // Compiler complains about this:
11    template<>
12    struct piecewise<int> {
13        typedef t type;
14    };
15 };
16
17 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

```

1 #include <iostream>
2 #include "constants.h"
3
4 namespace inside_container {
5     template<class t, class u>
6     struct piecewise {
7         typedef u type;
8     };
9
10    template<class t>
11    struct piecewise<t, int> {
12        typedef t type;
13    };
14 }
15
16 template<class t>
17 struct container {
18     template<class u>
19     struct piecewise {
20         typedef typename inside_container::piecewise<t, u>::type type;
21     };
22 };
23
24 typedef has_a_number<1> t1;
25 typedef has_a_number<2> t2;
26 typedef container<t1>::piecewise<t2>::type general;
27 typedef container<t1>::piecewise<int>::type specialized;
28
29 int main () {

```



```

30     std::cout << "general::number    = " << general::number    << std::endl <<
31         "specialized::number = " << specialized::number << std::endl;
32 }

```

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`

```

1 WORKING = first-order-functions higher-order-functions indirect-functions \
2         apply-two-function specialization inner-specialization preprocessor/k
3 ERRORS  = indirect-functions-broken inner-specialization-broken
4
5 CC      = g++
6 CC_OPTS = -g -Wall -std=gnu++0x
7
8 all: $(WORKING)
9 broken: $(ERRORS)
10
11 .PHONY: clean
12 clean:
13     rm -f $(WORKING) $(ERRORS)
14
15 %: %.cc
16     $(CC) $(CC_OPTS) $< -o $@ || true

```

2 Representation of Lambda Calculus Constructs

This section outlines the conversion process from the untyped lambda calculus to templates. It starts with just the context from [section 1](#), and ends up with a Turing-complete mini-language that is then implemented in [section 3](#). This language then forms the basis for the metacircular interpreter defined in [section ??](#).

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

3 Preprocessor Definitions

The goal of this section is to implement the language defined in [section ??](#). Roughly, the main operations are:

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

3.1 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, although its use may be limited to the GNU preprocessor.¹⁰

⁸There 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 ??](#). Also a great article about the role of typename in C++, its uses, and its limitations: <http://pages.cs.wisc.edu/~driscoll/typename.html>.

⁹It 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.

¹⁰A more standard way to do this is to use the `__VA_ARGS__` builtin and an anonymous ellipsis.