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

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;
```

---

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

```
 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 {
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
```

---

[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.

```
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, foo2 has a call slot apply that ultimately provides the value. So, for example, foo2<x>::apply<y>::type is equivalent to the more concise foo2 x y in Haskell, or ((foo2 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  };
```

4

```
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 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`

```
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 {
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   };
```

_____

[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 $\lambda(x, y).x$.

```
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 `(a, b) -> c`; none of the individual types are specified, but the template must be invoked on two parameters or not invoked at all. The other thing of note is that you can arbitrarily refine the left-hand side; for example:

```
template<template<template<class x> f, template<class y> class g> class compose>
struct ...
```

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

### 1.5  Makefile for examples

This makefile will build all of the examples listed in . It is available in the source distribution as `src/makefile`, and should be run from inside `src`.

Listing 9:  examples/makefile

```
1  WORKING = first-order-functions higher-order-functions \
2           indirect-functions apply-two-function
3  ERRORS  = indirect-functions-broken
4
5  CC      = g++
6  CC_OPTS = -g -Wall
7
8  all: $(WORKING)
9  broken: $(ERRORS)
10
```

```
11  .PHONY: run
12  run: all
13          ./first-order-functions
14          ./higher-order-functions
15          ./indirect-functions
16          ./apply-two-function
17
18  .PHONY: clean
19  clean:
20          rm -f $(WORKING) $(ERRORS)
21
22  %: %.cc
23          $(CC) $(CC_OPTS) $< -o $@
```