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 `struct`s, `class`es, 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`

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

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

```

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

```
// 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>
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 () {
```

[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.

```
  std::cout << "t1::number           = " << t1::number           << std::endl <<
               "should_be_t1::number = " << should_be_t1::number << std::endl;
}
```

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`

```cpp
#include <iostream>
#include "constants.h"
// Encoding the K combinator uniformly
struct k {
  template<class t>
  struct apply {
    template<class u>
    struct invoke_foobar {
      typedef t type;
    };

    typedef invoke_foobar apply;
  };
};
```

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.

1.4 Makefile for examples

This makefile will build all of the examples listed in the introduction.

Listing 6: `examples/makefile`

```make
all: first-order-functions higher-order-functions
run: all
	./first-order-functions
	./higher-order-functions
	./indirect-functions-broken

%: %.cc
	g++ -g -Wall $< -o $@
```