C++ Template Lisp Interpreter

Spencer Tipping

July 7, 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]This makes it untyped. C++ templates also support function types using nested template syntax – see section 1.4.

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.hh

```
// 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.hh"

// 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.hh"
```

[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.

```

// 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 () {
  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

```
// 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.hh"
```

```
// 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>
```

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

```
#include "constants.hh"

// Encoding the type signature as a template parameter specification
struct apply_two {
  template<template<class arg1, class arg2> class f>
  struct apply {
    struct type {
      template<class x>
      struct apply {
        typedef f<x, x> type;
      };
    };
  };
};

// An example value for f
template<class x, class y>
struct sample_f {
  typedef x x_type;
  typedef y y_type;
};

typedef has_a_number<10> t1;
typedef has_a_number<12> t2;
typedef apply_two::apply<sample_f>::type::apply<t1>::type two_of_t1;

int main () {
  std::cout << "t1::number                = " << t1::number << std::endl <<
               "two_of_t1::x_type::number = " << two_of_t1::x_type::number << std::endl <<
               "two_of_t1::y_type::number = " << two_of_t1::y_type::number << std::endl;
}
```

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.hh"

// 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`

```
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.hh"

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]

2 Metaprogramming in Value Space

Without using any particular encoding, this section provides a few examples of template metaprogramming in action. Each example has been factored down by preprocessor macros to show the structure of the code. Also, these examples are not necessarily representative of best practices, nor do they scale well.

2.1 Fibonacci numbers (explicit specialization)

This is some metaprogramming in value-space. Constants are evaluated and folded at compile-time, so the `expression` term below can be used to perform arithmetic evaluation. This example defines one general form of `fibonacci`, and two specific forms for an inductive process with two base cases. It also shows the lack of generality of piecewise definitions; while it is possible to specify a template for given values, implementing the classic compact definition of the function:

$$f(n) = \begin{cases} f(n-1) + f(n-2) & n \geq 2 \\ n & n < 2 \end{cases}$$

is not possible through specialization without enumerating every value of n below or above 2. In this section we implement a more straightforward definition:

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & \text{otherwise} \end{cases}$$

The compact definition is possible, however. The key is to encode that condition separately and specialize on a boolean parameter. That is implemented in section 2.2.

Listing 12 `examples/fibonacci.cc`

```
#include <iostream>

```

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

```
#define letrec(name, params, defaults, expression) \
  template params \
  struct name defaults { \
    enum { value = expression }; \
  };

#define call(name, params...) (name<params>::value)

letrec(fibonacci, <int n>, , call(fibonacci, n - 1) + call(fibonacci, n - 2))
letrec(fibonacci, <>, <0>, 0)
letrec(fibonacci, <>, <1>, 1)

int main () {
  std::cout << "fibonacci<10>::value = " << fibonacci<10>::value << std::endl;
}
```

2.2 Fibonacci numbers (piecewise definition)

As mentioned in section 2.1, it is possible to encode the Fibonacci function using a general conditional instead of explicitly specializing the two base values of 0 and 1. This example doesn't motivate its full utility (better would be a densely piecewise function such as the absolute value), but you can easily extrapolate from the design pattern.

The first step is to isolate the cases. For the piecewise Fibonacci function:

$$f(n) = \begin{cases} n & n < 2 \\ f(n-1) + f(n-2) & n \geq 2 \end{cases}$$

A boolean contains enough information to encode which branch should be taken. I'll use $n < 2$ as the predicate, so a value of `true` should result in n while a value of `false` should result in $f(n-1) + f(n-2)$.

Here is the obvious way to go about it, though it doesn't work because of mutual dependencies in the definitions:

Listing 13 examples/fibonacci-piecewise-broken.cc

```
template <bool n_lt_2, int n>
struct fibonacci_case {
  enum { value = fibonacci<n - 1>::value + fibonacci<n - 2>::value };
};

template <int n>
struct fibonacci_case <true, n> {
  enum { value = n };
};

```

```
template <int n>
struct fibonacci {
  enum { value = fibonacci_case<(n < 2), n>::value };
};
```

C++ needs to know about template types before you hit them. Because there isn't a way to forward-define templates, we have to get a little bit more creative:

Listing 14 examples/fibonacci-piecewise.cc

```
#include <iostream>

template <class f, bool n_lt_2, int n>
struct fibonacci_case {
  enum { value = f::template recursive<n - 1>::value +
                 f::template recursive<n - 2>::value };
};

template <class f, int n>
struct fibonacci_case <f, true, n> {
  enum { value = n };
};

struct fibonacci_piecewise {
  template <int n>
  struct recursive {
    enum { value = fibonacci_case<fibonacci_piecewise, (n < 2), n>::value };
  };
};

template <int n>
struct fibonacci {
  enum { value = fibonacci_piecewise::template recursive<n>::value };
};

int main () {
  std::cout << "fibonacci<10>::value = " << fibonacci<10>::value << std::endl;
}
```

Note the wrapping of `fibonacci_piecewise::recursive`. This is necessary because within the scope of a template, the name of the templated construct refers to the specialized form, not the template itself. For example:

Listing 15 examples/self-reference-broken.cc

```
template <int n>
struct foo {
  typedef foo bar;
```

```
};

typedef foo<10>::bar should_be_a_template;
typedef should_be_a_template<10> problem;
```

In this example, `should_be_a_template` isn't a template at all; it's the expanded form of `foo`, which has already been specialized to `foo<10>`. The compiler complains when `typedef`ing `problem` because we are trying to expand something that isn't a template.

The solution is to wrap the template inside a non-template outer `struct`, as the working Fibonacci example demonstrates. This enables you to refer unambiguously to the outer `struct` and then explicitly dereference the inner template, which will not be specialized at that point.

2.3 Linked list

This is a simple model of a data structure in value-space. For scalability it would be better to use types for the head and tail, but for simplicity the example is restricted to using values.

The `q` function in this example works around a limitation of the preprocessor. Section 3.1 covers this in more detail.

Listing 16 `examples/linked-list.cc`

```
#include <iostream>

#define function(name, params, values...) \
  template <params> \
  struct name { \
    enum { values }; \
  }

// Prevents commas from separating the arguments:
#define q(args...) args

function(cons, q(class X, class Y), car = X::value, cdr = Y::value);
function(car, class T, value = T::car);
function(cdr, class T, value = T::cdr);
function(int_wrapper, int n, value = n);

int main () {
  std::cout << "car(cons(5, 6)) = " <<
               car<cons<int_wrapper<5>, int_wrapper<6>>>::value << std::endl;
}
```

2.4 Church-encoded lists, `map`, and `filter`

This is a more involved example that uses curried templates and a Church encoding. Note the transition between value-space and type-space achieved by using `select`. `select` effectively serves as a map from `bool` to either `head` or `tail`.

Listing 17 examples/church-lists.cc

```
#include <iostream>

#define fn(args...)        template <args> struct
#define lambda(args...)    template <args> struct apply

#define lift(name, xs...)  template <xs> class name
#define call(v, xs...)     typename v::template apply<xs>::type

#define let(var, value...) private: typedef value var
#define ret(value...)      public:  typedef value type
#define val(e)             public:  struct type { enum { value = e }; }

struct head {lambda(class h, class t) {ret(h);};};
struct tail {lambda(class h, class t) {ret(t);};};

namespace _select {
  lambda(bool b)   {ret(head);};
  lambda() <false> {ret(tail);};
}

struct select {
  lambda(class t) {ret(call(_select, t::type::value));};
};

fn(class h, class t) cons {
  lambda(class f) {
    ret(call(f, h, t));
  };
};

struct nil {
  typedef int type;
};

struct map {
  lambda(class f, class cell) {
    let(mapped_head, call(f, call(cell, head)));
    let(mapped_tail, call(map, f, call(cell, tail)));
```

```
    ret(cons<mapped_head, mapped_tail>);
  };

  lambda(class f) <f, nil> {
    ret(nil);
  };
};

struct filter {
  lambda(class f, class cell) {
    let(h,             call(cell, head));
    let(filtered_tail, call(filter, f, call(cell, tail)));
    let(choices,       cons<cons<h, filtered_tail>, filtered_tail>);
    let(selector,      call(select, call(f, h)));

    ret(call(choices, selector));
  };

  lambda(class f) <f, nil> {
    ret(nil);
  };
};

fn(int n) int_wrap {
  val(n);
};

fn(class x) plus {
  lambda(class y) {
    val(x::type::value + y::type::value);
  };
};

fn(int n) is_divisible_by {
  lambda(class x) {
    val(x::type::value % n == 0);
  };
};

int main () {
  typedef cons<int_wrap<5>, cons<int_wrap<6>, nil>>      the_list;
  typedef plus<int_wrap<6>>                              the_function;
  typedef is_divisible_by<3>                             the_criterion;
  typedef filter::apply<the_criterion, the_list>::type   the_short_list;
  typedef map::apply<the_function, the_short_list>::type the_result;
  typedef the_result::apply<head>                        should_be_seven;
```

```

  std::cout << "head(map(x -> x + 6, filter(x % 3 == 0, list(5, 6)))) = " <<
               should_be_seven::type::value << std::endl;
}
```

The use of cons to represent the possible outcomes of a decisional isn't a new idea. It's key to the use of select, which will return either head or tail. We then apply the cons cell to that outcome to obtain the conditional result. The only problem with this approach is that both possibilities end up getting evaluated regardless of the condition. So in general, church-encodings of booleans and other conditionals has the caveat that it may cause infinite recursion. Most conditionals should probably be implemented using specialization (see section 2.2) to avoid this problem.

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.:

[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 **??**. Also a great article about the role of `typename` in C++, its uses, and its limitations: http://pages.cs.wisc.edu/~driscoll/typename.html.

[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.

```
#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]

4 β-Rewrite Representation

This section outlines the conversion process from a simple β-rewrite calculus to template metaprogramming constructs. This rewrite system then is combined with an `eval` function to form the basis for the metacircular interpreter defined in section ??.

4.1 List construct

The β rewrite system assumes the existence of two data types. One is the *term*, which corresponds to a value that might get replaced, and the other is a cons cell of two values. The first step to modeling these things is defining a cons cell, which for this system is Church-encoded:

Listing 18 core/cons.hh

```
#ifndef CORE_CONS_HH
#define CORE_CONS_HH

#include "core.hh"
#include LISP_BEGIN_CORE_MODULE()
struct cons {
  template <class h, class t>
  struct apply {
    struct type {
      template <class f>
      struct apply {
        typedef f<h, t> type;
```

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

```
      };
    };
  };
};
#include LISP_END_CORE_MODULE()

#endif
```

5 Resources

This section contains extra files that are helpful for running the examples.

5.1 Makefile for examples

This makefile will build all of the examples. It is available in the source distribution as `src/examples/makefile`, and should be run from inside `src/examples`. Note that it requires your version of `gcc` to support a fairly recent C++0x draft, as specified by `-std=gnu++0x`. At some point in the future I'll go back and revise the code for compatibility with the C++ 98 standard to remove this restriction.

Listing 19 `examples/makefile`

```
WORKING = first-order-functions higher-order-functions indirect-functions \
          apply-two-function specialization inner-specialization fibonacci \
          fibonacci-piecewise linked-list church-lists

BROKEN  = indirect-functions-broken inner-specialization-broken \
          fibonacci-piecewise-broken self-reference-broken

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

all: $(WORKING)
broken: $(BROKEN)

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

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