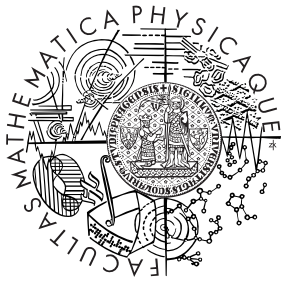

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Vít Šefl

λ-calculus as a Tool for Metaprogramming in C++

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: RNDr. Jan Hric

Study programme: Computer Science

Study branch: Theoretical Computer Science

Prague 2016

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague, 8th June 2016

Title: λ-calculus as a Tool for Metaprogramming in C++

Author: Vít Šefl

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Jan Hric, Department of Theoretical Computer Science and Mathematical Logic

Abstract: The template system of C++ is expressive enough to allow the programmer to write programs which are evaluated during compile time. This can be exploited for example in generic programming. However, these programs are very often hard to write, read and maintain. We introduce a simple translation from lambda calculus into C++ templates and show how it can be used to simplify C++ metaprograms. This variation of lambda calculus is then extended with Hindley-Milner type system and various other features (Haskell-like syntax, user-defined data types, tools for interaction with existing C++ template code and so on). We then build a compiler capable of transforming programs written in this language into C++ template metaprograms.

Keywords: C++ Haskell metaprogramming lambda calculus

I would like to thank RNDr. Jan Hric for his help, advice and feedback about this work. I would also like to thank RNDr. Petr Pudlák, Ph.D. for his help with Hindley-Milner type system and the compiler itself, and my friends and family for bearing with me.

Table of contents

Introduction

Templates were introduced into C++ to facilitate parametric polymorphism (also known simply as "generics"), which allows the programer to write programs that handle all values identically, no matter the type. The mechanisms behind templates (template specialization, substitution and now also variadic templates) are powerful enough to express far more complex programs.

In a way, templates form a language within a language. This metalanguage does not have mutable state nor typical control flow operators (if-then, while, etc.) and as such is basically a purely functional language.

However, as is usually the case with accidental features, this metalanguage is syntactically cumbersome – both hard to write and hard to read. This poses a maintainability problem for anyone who uses template metaprograms in their C++ code.

We therefore turn to traditional functional languages – the first and simplest of them is lambda calculus. Despite its simplicity, it is Turing complete and therefore makes a good foundation for many of today's functional languages. Establishing a connection between lambda calculus and template metalanguage would give us good basis upon which we could build more expressive and easier to use language.

In this work, we design a simple language based on lambda calculus which can be used to describe C++ metaprograms in a clear and simple manner. A compiler which translates programs written in this language into C++ code is also part of this work.

In the first chapter, we introduce template metaprogramming. Few simple examples are used to explain the basics of metaprogramming – representing values, functions, control flow and recursion. We then combine this knowledge to write a Turing machine simulator which shows that this metalanguage is Turing complete.

Lambda calculus is introduced in the second chapter. We show how the expressions of lambda calculus are formed and evaluated and how such simple language can be used to express more complex features (non-unary functions, recursion).

Third chapter is about type systems – essential feature of many modern programming languages. Type systems are used to provide some guarantees about our code before we run it and it is no different in the case of lambda calculus. Three type systems are introduced – simply typed lambda calculus, System F and Hindley-Milner type system – each with varying strengths and weaknesses. We also show how such systems can be extended with recursion and data types.

Fourth chapter defines translation of lambda calculus into template metaprograms. We show how to translate the basic lambda calculus as well as several features added on top (recursion, data types). Necessary metaprograms that need to be

defined outside of the language are also given, as are built-in data types and operations on them.

Fifth chapter is a quick summary of the implementation. It gives an overview of each module of the compiler along with the process of compilation. The type inference algorithm for Hindley-Milner type system is also quickly explained.

Sixth chapter shows how to write programs, use the compiler and also how to interact with the compiled code. This is done in the form of short, commented examples.

1 Metaprogramming in C++

Metaprogramming is programming about programs. A metaprogram is a program that operates and treats other programs as data. One of the first programming languages to allow metaprogramming was Lisp[1]. An interesting aspect is that Lisp's abstract syntax tree (AST) has the same structure as the programming language itself (this is known as *homoiconicity*). Today's programming languages are usually not built with metaprogramming in mind, so most of the added metaprogramming capabilities are expressed in a language different from the original language. Effort is usually made to have the "new" language similar to its original one (such as Template Haskell[2]) but sometimes metaprogramming arises as an accident of already existing features and the resulting "language" for expressing such programs is very different from the original (such as templates in C++).

1.1 Basic concepts

Metaprogramming in C++ is done using templates. Templates were originally meant as a tool for parametric polymorphism – being able to write one program that works on multiple types of data. However, they were general enough to allow expressing more complicated computations than just substituting a type. In fact, it has been shown that templates in C++ are Turing complete – meaning that any computation can be computed by the means of a template metaprogram[3]. In this work, we will focus exclusively on template metaprogramming as a tool for writing compile-time programs.

Basic knowledge of C++ templates is assumed. At first, we will explain the basic concepts of template metaprogramming on a handful of examples and then we will combine this knowledge to show that template metaprogramming is indeed Turing complete. Turing machines are of course poorly suited for programming, but Turing completeness shows the strength of template metaprogramming as a tool.

First, four basic concepts need to be explained: using compile-time constants as expressions of the metaprogram, using template parameters as function parameters, using template specializations as conditionals (if-then-else, case) and template recursion.

There are various kinds of compile-time constants. We will mainly focus on typedef and static constant members of classes as they allow the most flexibility. As an example, here is one possible way to write the expression `1 + 1`:

```
static const int value = 1 + 1;
```

While this works just fine for simple use cases, it is a good idea to introduce a level of indirection. In order to be able to manipulate the expressions with templates, expressions are wrapped in named classes.

```
struct Two {
    static const int value = 1 + 1;
};
```

The value of such expression can now be accessed simply by writing:

```
Two::value == 2;
```

One of the most important tricks of metaprogramming is using template parameters inside compile-time constants. This will allow the use of template parameters as inputs to some sort of a function. For example, a simple (meta)function that doubles its formal parameter:

```
template <int I>
struct Double {
    static const int value = 2 * I;
};
```

Calling such function is simply a matter of instantiating the template with desired parameter:

```
Double<4>::value == 8;
```

Template specialization can also be used to make decisions based on the template parameter. This allows expressing various kinds of conditionals. Let us implement unary negation on booleans:

```
template <bool>
struct Not;


template <>
struct Not<true> {
    static const bool value = false;
};


template <>
struct Not<false> {
    static const bool value = true;
};
```

However, templates do not have to be fully specialized. Sometimes it is more convenient to specialize only templates that handle edge cases. If none of the edge cases match, the general template is selected.

```
template <int I>
struct IsZero {
    static const bool value = false;
};

template <>
struct IsZero<0> {
    static const bool value = true;
};
```

Notice that the expression defining `value` needs to be evaluated at compile-time. `value` itself can also be evaluated at compile-time, which means more complicated expressions can be written by means of simpler ones, such as:

```
template <bool B>
struct NotNot {
   static const bool value = Not<Not<B>::value>::value;
};
```

This can be further extended to recursive programs. We can combine the techniques mentioned above and write a simple function that computes factorial of a given number:

```
template <int I>
struct Factorial {
    static const int value = I * Factorial<I - 1>::value;
};

template <>
struct Factorial<0> {
    static const int value = 1;
};
```

When `Factorial` is later used, such as

```
Factorial<5>::value == 120;
```

the compiler needs to instantiate the template `Factorial<5>`. In order to figure out the value of the constant `value`, it needs to instantiate `Factorial<4>`, then `Factorial<3>` and so on. Once it gets to `Factorial<0>`, it uses the specialization where the constant `value` is no longer defined in terms of `Factorial` and is simply defined to be `1`.

1.2 Type manipulation

Other types can be manipulated in a similar way. This is very important for generic programming and is one of the reasons why various type functions implemented via templates have made their way into the official C++ standard[4]. Of course, static constant members cannot be used to express types. This is where `typedef` comes into play. Say we would like to write a function that takes a type `T` and produces a

`T`-pointer (`T*`). The first steps are familiar: the formal parameter is represented as a template parameter.

```
template <typename T>
struct AddPtr {
    // ?
};
```

However, static constant members cannot be used to represent the result. Instead, we will create an inner type named `type`:

```
template <typename T>
struct AddPtr {
    typedef T* type;
};
```

This type function can be used in the same way as functions operating on values:

```
AddPtr<int>::type x = &y;
```

Both kinds of functions can be combined to create a function that operates both on types and values.

```
template <typename, int>
struct AddManyPtr;


template <typename T>
struct AddManyPtr<T, 0> {
    typedef T type;
};


template <typename T, int I>
struct AddManyPtr {
    typedef typename AddManyPtr<T, I - 1>::type* type;
};


int** x = 0;
AddManyPtr<int, 2>::type y = x;
```

Notice the use of keyword `typename`. C++ templates make no guarantee about the nature of inner types or members, so unless unambiguous from the context or otherwise specified, such qualified names refer to members. Indeed, it is possible to have a template where the name `type` can refer to both a type and a member (depending on the template specialization that is selected). For this reason, when we want to use such name as a type, we must inform the compiler to expect a type.

Starting with C++11, the standard library contains a header `<type_traits>` that features many such type functions. Few of those are in fact the backbone of the more prominent C++11 features, such as move semantics.

C++11 also introduced variadic templates – a kind of templates that do not have a fixed number of template parameters. This is a crucial feature for implementing

more complex structures at the template level. For example, variadic templates can be used to express type-level lists:

```
template <typename...>
struct List { };

typedef List<int, char, bool> list_of_types;
```

We can implement simple functions to add elements at either end of the list:

```
template <typename, typename>
struct PushFront;

template <typename A, typename... R>
struct PushFront<A, List<R...>> {
    typedef List<A, R...> type;
};

// PushFront<int, List<char, bool>>::type
// == List<int, char, bool>

template <typename, typename>
struct PushBack;

template <typename A, typename... R>
struct PushBack<A, List<R...>> {
    typedef List<R..., A> type;
};

// PushBack<int, List<char, bool>>::type
// == List<char, bool, int>
```

Or even access element at a given position:

```
template <int, typename>
struct At;

template <typename A, typename... R>
struct At<0, List<A, R...>> {
    typedef A type;
};

template <int N, typename A, typename... R>
struct At<N, List<A, R...>> {
    typedef typename At<N - 1, List<R...>>::type type;
};

// At<1, List<int, char, bool>>::type == char
```

This function will obviously not work when given a negative number or a number that represents a position past the list end. Compile errors of this kind are very hard

to decipher and for that reason it is a good idea to include `static_assert` to check if all preconditions are met.

1.3 Higher-order template functions

Another interesting fact is that C++ templates allow very restricted use of higher-order (template) functions (function that take another functions as their parameters). For example, here is one possible implementation of a filter operation that removes elements of a list based on a unary predicate.

```
template <template <typename> class, typename>
struct Filter;


template <template <typename> class F>
struct Filter<F, List<>> {
    typedef List<> type;
};


template <template <typename> class F, typename A, typename... R>
struct Filter<F, List<A, R...>> {
    typedef typename std::conditional<
            F<A>::value,
            typename PushFront<
                A,
                typename Filter<F, List<R...>>::type
            >::type,
            typename Filter<F, List<R...>>::type
        >::type type;
};
```

Notice that `F` does not refer to a simple type but instead to a template class (as opposed to a template function) expecting one type as its template parameter. `std::conditional` comes from the header `<type_traits>` and does what its name says: if the first (boolean) template parameter is true, it stores the second template parameter into the inner type `type`, otherwise it uses the third template parameter. The template-template parameter `F` is a function that, when given a type, produces a boolean value in the inner member `value`. Example of such function is:

```
template <typename T>
struct IsInt {
    static const bool value = false;
};


template <>
struct IsInt<int> {
    static const bool value = true;
};
```

1.4 Representation of basic values

One major problem of this representation is that we would need a type-level list for each type of data we would like to store. This also leads to duplication of all the

helper functions that are defined for a `List` (`PushFront`, `PushBack`, `Filter`). Very soon, the number of various `TypeLists`, `IntLists`, `IntFilters`, `BoolLists`, `BoolFilters` would not be manageable. However, there is a very simple trick that unifies these representations:

```
template <int I>
struct Int {
    static const int value = I;
};
```

Now, `Int<4>` is a type representing the integer value `4`. The crucial part is that `Int<4>` is now a type, which means it can be used in template parameters declared with the `typename` keyword. If we want a type-level list of integers, we can simply use the previously defined `List` this way:

```
typedef List<Int<1>, Int<2>, Int<3>> list_of_ints;
```

And since everything contained in the list is a type, previously defined `Filter` function can be reused. Only this time, the filtering function must first take the integer out of the type before it can do any work on it:

```
template <typename>
struct LessThanTwo;

template <int I>
struct LessThanTwo<Int<I>> {
    static const bool value = I < 2;
};

// Filter<LessThanTwo, list_of_ints>::type == List<Int<1>>
```

1.5 Simulating a Turing machine

At this point, we have enough tools to implement a simple Turing machine simulator. There are many equivalent descriptions of a Turing machine. For the sake of our example, we will use a machine that can move the tape only left and right (we do not have an instruction that would keep the tape in place) and a unary predicate describing whether an end state has been reached.

Before starting, a definition of the tape is needed. We shall use a triple – list of elements before the selected position (stored in reverse order for ease of implementation), the element at the selected position (also known as the head) and list of elements after. This provides very efficient access to the currently selected element and also efficient operations to move the head left or right. Although not necessary, few inner types are used for convenient access to the three parts of the tape:

```
template <typename L, typename X, typename R>
struct Tape {
    typedef L left;
    typedef X middle;
    typedef R right;
};
```

The first tape operation is movement. For simplicity, the direction is simply represented by boolean value describing whether to go left.

```
template
  < bool GoLeft
  , typename Tape
  , typename Blank
  >
struct Move;
```

There are four cases to consider. Only the first two are shown, full implementation can be found in the appendix. When the head is moved to the left and the left sublist has no elements, the newly created position has to be filled with a blank symbol. Movement to the right is symmetric.

```
template
  < typename X
  , typename... R
  , typename Blank
  >
struct Move<true, Tape<List<>, X, List<R...>>, Blank> {
    typedef Tape<List<>, Blank, List<X, R...>> type;
};
```

When the left sublist has elements, the element in the current position can be used:

```
template
  < typename L
  , typename... LS
  , typename X
  , typename... R
  , typename Blank
  >
struct Move<true, Tape<List<L, LS...>, X, List<R...>>, Blank> {
    typedef Tape<List<LS...>, L, List<X, R...>> type;
};
```

Lastly, a function that writes new symbol into the selected position:

```
template
  < typename Tape
  , typename Symbol
  >
struct Write;
```

```
template
  < typename L
  , typename X
  , typename R
  , typename Symbol
  >
struct Write<Tape<L, X, R>, Symbol> {
    typedef Tape<L, Symbol, R> type;
};
```

The actual implementation of the simulator is surprisingly simple. To simulate a Turing machine, the following parameters have to be provided: the blank symbol (to fill empty sections of the tape), current state, end states, transition function (a function accepting state and current symbol and producing new state, new symbol and direction of movement) and tape. The implementation uses two mutually recursive functions – first function (`Turing`) to figure out whether the simulation can be stopped (current state is one of the end states) and the second one (`Helper`) to simulate one step of the machine before calling the first function again.

```
template
  < typename Blank
  , typename State
  , template <typename> class End
  , template <typename, typename> class Step
  , typename Tape
  >
struct Turing {
    static const bool stop = End<State>::value;

    typedef typename Helper<
            stop, Blank, State, End, Step, Tape
        >::type type;
};
```

Notice that all the `Turning` function does is call `End` on the current state and then call the `Helper` function with the result. The magic happens in the `Helper` function:

```
template
  < bool Stop
  , typename Blank
  , typename State
  , template <typename> class End
  , template <typename, typename> class Step
  , typename Tape
  >
struct Helper;
```

The first case to consider is when current state is one of the end states (`End<State>::value` is true). In this case, current tape can be returned.

```
template
  < typename Blank
  , typename State
  , template <typename> class End
  , template <typename, typename> class Step
  , typename Tape
  >
struct Helper<true, Blank, State, End, Step, Tape> {
    typedef Tape type;
};
```

The second case is the actual simulation:

```
template
  < typename Blank
  , typename State
  , template <typename> class End
  , template <typename, typename> class Step
  , typename L
  , typename X
  , typename R
  >
struct Helper<false, Blank, State, End, Step, Tape<L, X, R>> {
    typedef Step<State, X> step;


    typedef typename Write<
            Tape<L, X, R>, typename step::newSymbol
        >::type tapeAfterWrite;


    typedef typename Move<
            step::dirIsLeft, tapeAfterWrite, Blank
        >::type tapeAfterMove;


    typedef typename Turing<
            Blank, typename step::newState,
            End, Step, tapeAfterMove
        >::type type;
};
```

This case is far more involved. The computation is broken down into a few steps. We begin by applying the transition function and storing the result into the `step` type. Then the new symbol is written to the tape (which is contained in an inner type `newSymbol` in the `step` type) by the `Write` function. After that, the head is moved in a direction specified by the transition function (boolean constant `dirIsLeft`) and finally `Turing` function is called recursively with the new tape and the new state (given by inner type `newState`).

And indeed, that is the whole implementation. To finish the example, we shall show how to write our own end state predicate and transition function. The Turing machine in this example operates on an alphabet {0, 1, 2}, where 0 is the blank symbol. It moves to the left until it encounters a 1, rewrites it to 2 and then terminates. For this, only two states are needed – one state that does the search and rewrite (`'A'`) and second state that is simply an end state (`'B'`). This gives the implementation of `MyEnd` function:

```
template <char C>
struct Char {
    static const char value = C;
};


template <typename>
struct MyEnd;


template <>
struct MyEnd<Char<'A'>> {
    static const bool value = false;
};


template <>
struct MyEnd<Char<'B'>> {
    static const bool value = true;
};
```

The transition function is also very straightforward. However, it is important to remember that the `Turing` (and `Helper`) function expects the class to have the following: two types (`newSymbol`, `newState`) and one static constant (`dirIsLeft`):

```
template <typename State, typename Symbol>
struct MyStep;


template <>
struct MyStep<Char<'A'>, Int<0>> {
    typedef Int<0> newSymbol;
    typedef Char<'A'> newState;
    static const bool dirIsLeft = false;
};


template <>
struct MyStep<Char<'A'>, Int<1>> {
    typedef Int<2> newSymbol;
    typedef Char<'B'> newState;
    static const bool dirIsLeft = false;
};


template <>
struct MyStep<Char<'A'>, Int<2>> {
    typedef Int<2> newSymbol;
    typedef Char<'A'> newState;
    static const bool dirIsLeft = false;
};
```

There is no need to provide specialization for the other state as `MyStep` will never be called on it. Here's how one can simulate the Turing machine specified by the above functions on the tape [0, 0, 1, 0, 1]:

```
Turing
    < Int<0>
    , Char<'A'>
    , MyEnd
    , MyStep
    , Tape<List<>, Int<0>, List<Int<0>, Int<1>, Int<0>, Int<1>>>
    >::type
```

The resulting type gives the expected tape [0, 0, 2, 0, 1] (recall that the left sublist is stored in reverse order):

```
Tape<List<Int<2>, Int<0>, Int<0>>, Int<0>, List<Int<1>>>
```

We have shown that C++ templates are indeed powerful enough to express any computation. However, at this point it should be absolutely clear that the language of C++ templates is extremely verbose and hard to read. In the following chapters, we will explore the possibility of using lambda calculus to solve the above mentioned problems.

2 Lambda calculus

Lambda calculus originated from an attempt at creating foundation for logic which would not be based on sets (Zermelo's set theory) but rather on functions. Alonzo Church published the first version of lambda calculus in 1932[5]. It was quickly found that the system was inconsistent (Kleene-Rosser paradox[6]). However, lambda calculus surprisingly provided very good description of computation. This part of the original lambda calculus relevant to computation is now known simply as the untyped lambda calculus. Over the next few years, Kleene, Church and Turing proved that untyped lambda calculus, Turing machines and partially recursive functions have the same computational power. Interestingly, Church solved the lambda calculus version of the halting problem shortly before Alan Turing independently solved halting problem for his model of computation.

2.1 Syntax

The syntax of lambda calculus is very simple. Expression is either a variable, an application or an abstraction. The grammar is as follows:

$$\begin{aligned} E ::= {} & x && \text{(variable)} \\ | \; & EE && \text{(application)} \\ | \; & \lambda x.E && \text{(abstraction)} \end{aligned}$$

While application might look unfamiliar, the expression MN would correspond to `M(N)` in C++. Abstraction can be thought of as a way of creating functions without the need for a separate definition. The variable x in $\lambda x.E$ is the formal parameter and E is the function body. Indeed, since C++11, C++ can express abstraction in similar way:

```
[] (int x) { return x + x; }
```

This is a shorthand for creating unnamed class with overloaded `operator()`. However, the end effect is the same: we create an unnamed function. Such expression can be though of as a literal for function types. In lambda calculus, the previous expression would be written as:

$$\lambda x.x + x$$

Before moving on, few conventions about the syntax. Application is left associative, that means:

$$MNL$$

is the same as:

$$(MN)L$$

which would be expressed in C++ as:

```
M(N)(L)
```

And the expression following the dot in lambda abstraction extends to the right as far as possible.

$$\lambda x.xy$$

in fact means

$$\lambda x.(xy)$$

and not

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

2.2 Semantics

So far we have only seen the syntax. But earlier, we mentioned that lambda calculus can be used to express computations. Since it is a language of functions – their introduction (abstraction) and elimination (application) – the computation is really just a simple substitution. When an abstraction is applied to another lambda expression, the expression can be simplified (or reduced) in the following way:

$$(\lambda x.M)N \rightsquigarrow M[x := N]$$

This is known as the β-reduction. $M[x := N]$ is a shorthand for capture-avoiding substitution, which takes all the free occurrences of x in the expression M and replaces them with the expression N. Why do we consider only free occurrences? Variable names are interchangeable, this means that the following two expressions are equivalent (this is called α-equivalence):

$$\lambda x.x$$
$$\lambda y.y$$

Now, let us consider slightly more complicated expressions:

$$\lambda x.\lambda x.x$$
$$\lambda x.\lambda y.y$$

These two are also equivalent. The variable x in the first expression refers to the inner abstraction. This is known in programming languages as variable shadowing. What happens if all occurrences of x are substituted?

$$(\lambda x.\lambda x.x)M \rightsquigarrow (\lambda x.x)[x :\approx M] \rightsquigarrow \lambda x.M$$
$$(\lambda x.\lambda y.y)M \rightsquigarrow (\lambda y.y)[x :\approx M] \rightsquigarrow \lambda y.y$$

Two equivalent expressions were applied to M but they produced two different results. Variables should be interchangeable, therefore this is unwanted behavior. This can be solved simply by substituting only free occurrences (those that are not bound by an abstraction).

When reducing an expression to a point where no further reduction can be made, we reach so called normal form. Lambda calculus contains expressions that have no normal form. Consider the following:

$$\Omega = (\lambda x.xx)(\lambda x.xx)$$

β-rule can be used:

$$\Omega = (\lambda x.xx)(\lambda x.xx) \rightsquigarrow (xx)[x := \lambda x.xx] \rightsquigarrow (\lambda x.xx)(\lambda x.xx) = \Omega$$

We tried to reduce the expression but arrived at the expression we started with. There are also expressions where normal form can be reached only if the reduction is done in the right order.

$$K = \lambda x.\lambda y.x$$
$$I = \lambda x.x$$
$$KI\Omega = (\lambda x.\lambda y.x)(\lambda x.x)\Omega \rightsquigarrow (\lambda y.\lambda x.x)\Omega \rightsquigarrow \lambda x.x = I$$

However, if the Ω part is repeatedly chosen for reduction, one will always arrive at the expression $KI\Omega$.

We have seen that one expression may have more than one subexpression that can be reduced in this way (these subexpressions are called β-redexes). The way of selecting which β-redex to reduce first is known as reduction strategy.

Selecting the outermost and leftmost β-redex is a reduction strategy known as normal order. If a lambda expression can be reduced to a normal form, normal order reduction is guaranteed to reach it.

So far, we have described the syntax and semantics of lambda calculus. Now we shall look at lambda calculus as a programming language. Despite the fact that lambda calculus is a language of functions, it is possible to express data as known in conventional programming languages (numbers, booleans, pairs, etc). Of course, such data would be encoded via functions. The actual encoding is out of the scope of this work, so it is just assumed that this encoding and the operations on the encoded data exist and can be used in the expected ways.

2.3 Currying

Notice that lambda calculus does not have function abstraction for functions of multiple parameters. Instead, a function with multiple parameters is represented as a chain of unary functions, each returning another function with the given parameter already fixed. This is known as currying.

A binary function – such as integer addition – can be rewritten in the following way:

$$\text{Add} = \lambda x.\lambda y.x + y$$

Or in shorthand notation:

$$\text{Add} = \lambda xy.x + y$$

Notice that when applying Add to a number, the result is a new function where the parameter x is fixed.

$$\text{Add } 5 = (\lambda x.\lambda y.x + y)5 \rightsquigarrow \lambda y.5 + y$$

Indeed, $\text{Add } 5$ is a function that simply adds 5. By fully applying the function, the expected result is obtained.

$$\text{Add } 5\ 4 = (\lambda xy.x + y)5\ 4 \rightsquigarrow (\lambda y.5 + y)4 \rightsquigarrow 5 + 4 \rightsquigarrow 9$$

When attempting to express the same principle in C++ lambda expressions, we will soon encounter a problem. While C++ allows the return type to be a function type (usually unnamed, implementation defined type or `std::function`), it has to solve a problem that lambda calculus does not have. Lambda calculus is a pure system of abstraction, application and substitution. There is no mutation, variables are indeed variables in the mathematical sense. In the subexpression:

$$\lambda y.x + y$$

the occurrence of variable x is free. What happens when someone changes the value of x from the outside? Should the occurrence in the above expression be updated? Or should the value be kept as it was? Unlike lambda calculus, C++ allows this kind of mutation and the programmer must decide which behavior should the lambda expression use: use a reference (allow mutation from outside) or use a copy (ignore mutation from outside).

To implement our curried `Add` function, we will capture the free variable using a copy. Once we apply the function to the first parameter, the result will be computed and the memory occupied by the parameter will be freed. Had we used capture by reference, the first parameter would be long gone at the point we tried to compute the result of the second application.

```
auto Add = [] (int x) { return [=] (int y) { return x + y; }; };
int result = Add(5)(4);
```

2.4 Recursion

Complex functions can be defined in terms of simpler operations, but there is no apparent way in which to define recursive functions. Indeed, lambda calculus does not seem to allow self-reference, a crucial feature required for recursion.

Self-reference can be accomplished using a fixed-point combinator. This is a lambda expression Y with the following property:

$$Yf = f(Yf)$$

Notice that Yf "replicates" itself in the expression applied to f. In this way, Yf gives the expression f access to itself, creating a self-reference.

There are many different fixed-point combinators. We shall use Curry's combinator Y:

$$Y = \lambda g.(\lambda x.g(xx))(\lambda x.g(xx))$$

$$\begin{aligned} Yf &\rightsquigarrow (\lambda x.f(xx))(\lambda x.f(xx)) \\ &\rightsquigarrow f((\lambda x.f(xx))(\lambda x.f(xx))) = f(Yf) \end{aligned}$$

What remains to be shown is how fixed-point combinator can be used to express recursion. For the sake of example, let us try to implement a factorial function. If recursion without the need for fixed-point combinator could be used, the expression would be:

$$\text{Fac} = \lambda n.\text{if } n == 0 \text{ then } 1 \text{ else } n * \text{Fac } (n-1)$$

The factorial function can be built from a one-step function.

$$\text{FacStep} = \lambda rest.\lambda n.\text{if } n == 0 \text{ then } 1 \text{ else } n * rest\ (n-1)$$

When n is zero, the function computing the rest of factorial (represented as the $rest$ variable) is not needed. Therefore $rest$ can be replaced by a dummy expression. A function computing factorial of zero can then be implemented:

$$\text{Fac0} = \text{FacStep dummy}$$

When n is one or zero, two steps can be done at once:

$$\text{Fac1} = \text{FacStep Fac0} = \text{FacStep (FacStep dummy)}$$

Let us do the reduction manually to see the behavior for one and zero:

$$\begin{aligned}\text{Fac1 } 0 &= (\lambda rest.\lambda n.\text{if } n == 0 \text{ then } 1 \text{ else } n * rest\ (n-1))\ \text{Fac0 } 0\\ &\rightsquigarrow (\lambda n.\text{if } n == 0 \text{ then } 1 \text{ else } n * \text{Fac0}\ (n-1))\ 0\\ &\rightsquigarrow 1\end{aligned}$$

$$\begin{aligned}\text{Fac1 } 1 &= (\lambda rest.\lambda n.\text{if } n == 0 \text{ then } 1 \text{ else } n * rest\ (n-1))\ \text{Fac0 } 1\\ &\rightsquigarrow (\lambda n.\text{if } n == 0 \text{ then } 1 \text{ else } n * \text{Fac0}\ (n-1))\ 1\\ &\rightsquigarrow 1 * \text{Fac0 } 0\\ &\rightsquigarrow 1 * 1\\ &\rightsquigarrow 1\end{aligned}$$

When n is one, only one step of the computation is done and then the Fac0 function is applied to zero – and Fac0 is already capable of computing the factorial of zero. More and more steps can be added to better approximate the actual factorial function:

$$\begin{aligned}&\text{Fac2} = \text{FacStep Fac1}\\ &\text{Fac3} = \text{FacStep Fac2}\\ &\quad\ldots\end{aligned}$$

If an infinite amount of steps could be added, we would have a function that computes factorial for all natural numbers. But this is precisely what the fixed-point combinator does:

$$\begin{aligned}Y\ \text{FacStep} &= \text{FacStep}\ (Y\ \text{FacStep})\\ &= \text{FacStep}\ (\text{FacStep}\ (\text{FacStep}\ (\ldots)))\end{aligned}$$

This gives the final implementation:

$$\text{Fac} = Y\ \text{FacStep}$$

Of course, having an infinite amount of steps leads to a question: how can such function ever terminate? If the reduction is done correctly (such as by using normal order), upon reaching zero, the rest of the infinite chain of FacStep functions does not need to be reduced. The last FacStep simply reduces to 1 and ignores the infinite chain.

This trick can be extended to all recursive functions. A simple replacement of the recursive occurrence by an additional abstraction (an extra function parameter) is all that is needed.

$$\text{F} = \ldots \text{F}\ x \ldots$$

can be transformed into

$$\text{F} = Y\ (\lambda rec.\ \ldots rec\ x \ldots)$$

3 Type systems

In 1940, Alonzo Church fixed the fundamental problem of untyped lambda calculus (its inconsistency when used as a logical system) by adding the notion of types[7]. A type is a syntactic construct that classifies lambda expressions. It restricts which lambda expressions are valid and how they can be used together. In a way, types of this kind of lambda calculus have the same purpose as the types used in programming languages today – catch and prevent unwanted behavior before running the code (reducing the lambda expression).

3.1 Simple types

Since lambda calculus only consists of functions and operations on them, the types for this system are very simple. They can be defined with a grammar:

$$\begin{aligned} T ::= a & \qquad \text{(type constant)} \\ \mid T \to T & \qquad \text{(function type)} \end{aligned}$$

Type constants can be thought of as some basic types of the language. Since everything is a function, the actual nature of type constants is not important.

The interpretation of such types is very natural. Let us start by showing few examples of C++ lambda expressions.

```
auto f = [] (int x) { return x + 2; };
auto g = [] (int x) { return [=] (int y) { return x + y; }; };
auto h = [] (std::function<int(int)> f) { return f(2); };
```

We can infer how the types of these expressions would look in the typed lambda calculus. The first function f has one parameter of type int and returns a value of type int, its type would be:

$$f : \mathrm{int} \to \mathrm{int}$$

The second function g has one parameter of type int and returns another function (taking int and returning int). In fact, we have seen this function in the previous chapter as an example of currying. Its type would be:

$$g : \mathrm{int} \to (\mathrm{int} \to \mathrm{int})$$

And the last function h has one parameter with a function type (taking int and returning int). It returns a value of type int.

$$h : (\mathrm{int} \to \mathrm{int}) \to \mathrm{int}$$

Notice that the arrow is not associative. If the type is presented as a tree, where the inner nodes are arrows and leafs are concrete types (type constants), the root of the tree decides what are parameters and what is the return type. Recall that lambda calculus has only unary functions, therefore the whole left subtree (of the root node) is the type of the (only) function parameter and the whole right subtree is the return type.

The arrow is conventionally assumed to be right associative, therefore some parentheses can be dropped:

$$g : \text{int} \to (\text{int} \to \text{int})$$

becomes

$$g : \text{int} \to \text{int} \to \text{int}$$

So far, we were using intuition to guess what the types should be. Typed lambda calculus is meant to be a system of logic, however. We therefore need rigid rules to derive the correct type for each lambda expression.

First of all, let us extend the syntax of untyped lambda calculus.

$$\begin{aligned} E ::= {} & x && \text{(variable)} \\ | \; & EE && \text{(application)} \\ | \; & \lambda x : T.E && \text{(abstraction)} \end{aligned}$$

Notice that the variable x in the abstraction is now annotated by its type. This is similar to the way in which C++ demands types for function parameters. The type after the colon can be any type given by the grammar above.

3.2 Simply typed lambda calculus

Before writing down the rules a few important concepts need to be introduced. Types are not derived in vacuum. When writing the C++ expression:

```
int x = -y;
```

a lot of things are assumed in the background – the type of `y` is `int`, the type of the unary operator `-` is `int (*)(int)`. In a similar way, when deriving the type of a lambda expression, we will carry around a list of variables (functions can be regarded as variables with a function type) whose type is already known. This is known as a *context*.

A context is usually called Γ (capital gamma) and it is represented as a list of pairs variable:type. The context for the previous expression in C++ could look like this:

$$\Gamma = \{y : \text{int}, - : \text{int} \to \text{int}\}$$

Rules are usually represented in the style of natural deduction. Each rule has premises (above the line) and one conclusion (below the line):

$$\frac{\text{premise}_1 \ \ldots \ \text{premise}_n}{\text{conclusion}}(\text{name of the rule})$$

If all the statements above the line hold, the rule can be used to derive the statement below the line.

And finally, some syntactic conventions. The symbol ":" (colon) means "has type" and "$\vdash$" (turnstile) means "can be derived". In particular, "$\Gamma \vdash x : \text{int}$" means "$x$ has the type int can be derived given the context Γ".

With that out of the way, let us present the rules.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}(\text{var})$$

$$\frac{\Gamma \vdash M : S \to T \quad \Gamma \vdash N : S}{\Gamma \vdash MN : T}(\text{app})$$

$$\frac{\Gamma, x : S \vdash M : T}{\Gamma \vdash (\lambda x : S.M) : S \to T}(\text{abs})$$

Let us go over the rules in detail. The first rule states that if the context contains the information that x has the type T, this information can be used to derive $x : T$. Simply stated: anything in context can be used in the derivation.

The second rule concerns application. If M is an expression with function type $S \to T$ and N is an expression of type S, then the application of M to N is an expression of type T.

The third rule is slightly more complicated. If we want to derive the type for a lambda abstraction, we first need to derive the type of the expression M (we could call this the body of the abstraction). However, the body can use the bound variable x, therefore extra information about the bound variable has to be included in the context for the premise.

Consider the expression $\lambda x : T.x$. Its type can be derived in the following way:

$$\frac{\dfrac{x : T \in \{x : T\}}{\{x : T\} \vdash x : T}(\text{var})}{\{\} \vdash (\lambda x : T.x) : T \to T}(\text{abs})$$

Lambda calculus with such modifications is known as simply typed lambda calculus (STLC). Reduction can be defined in exactly the same way as it was for the untyped version. However, reduction should respect the types. In particular, if an expression M had type T before reduction, it should be possible to derive the same type T for the reduced version of M. And indeed, STLC plays nicely with reduction. It also has more desirable properties.

The previously problematic expressions, such as:

$$\Omega = (\lambda x.xx)(\lambda x.xx)$$

are not valid expressions in STLC. There is no type that can be derived for this expression. In fact, a stronger property holds: STLC is strongly normalizing – meaning that any well typed expression can be reduced to a normal form, no matter the order in which the reduction is done. This is good news for STLC as a system of logic (as it is no longer inconsistent) but it also means that STLC is no longer Turing complete.

While the type system of STLC is a great help, we are still a long way from a language that would be convenient for everyday programming. In particular, the simple types of STLC help us recognize and reject broad class of meaningless expressions, but the lack of data types, general recursion and parametric polymorphism do not make for a good programming language.

3.3 Polymorphism

STLC is actually one step away from parametric polymorphism. Notice that the expressions:

$$\lambda x : T.x$$
$$\lambda x : S.x$$

have the same basic structure, they express the same function. However, the simple types of STLC are not able to capture this similarity and allow us to use one function in place of both of those. We will need a more powerful type system.

Again, let us start by extending the type language:

$$\begin{aligned} T ::= {} & x && \text{(type variable)} \\ | \; & T \to T && \text{(function type)} \\ | \; & \forall x.T && \text{(universal quantification)} \end{aligned}$$

The new addition is universal quantification – a way to express that an expression can be used with *any* type. This is very similar to function templates in C++. The following template:

```
template <typename T>
T id(T x) { return x; }
```

could be given the type $\forall x.x \to x$. This means that `id` is a function of type $x \to x$ for any type x. However, universal quantification is much more powerful than function templates in C++. One can even express function types where the parameter is a universally quantified function – in a way, a function expecting polymorphic function as its first parameter:

$$(\forall x.x \to x) \to \text{int} \to \text{int}$$

So while a function of this type could be used with the previously defined function `id` (since `id` works for any type), it could not be used with a function with simple type such as $\text{int} \to \text{int}$ or $\text{char} \to \text{char}$.

Let us extend the syntax of lambda calculus in the following way:

$$\begin{aligned} E ::= {} & x && \text{(variable)} \\ & \mid EE && \text{(application)} \\ & \mid \lambda x : T.E && \text{(abstraction)} \\ & \mid ET && \text{(type application)} \\ & \mid \Lambda x.E && \text{(type abstraction)} \end{aligned}$$

Type abstraction is represented in a similar way as the ordinary abstraction, but capital lambda is used (Λ) to express that the abstraction happens at a higher level (types instead of values). In the same way as ordinary abstraction serves to create a function, type abstraction serves as a way of creating polymorphic expressions. It can be thought of as the template declaration in C++. Indeed, the variable bound by the type abstraction is basically the typename variable in the template declaration.

```
template <typename T>
T f(T arg) { ... };
```

would correspond to:

$$f = \Lambda T.\lambda arg : T.\ \ldots$$

Using such polymorphic function is simply a matter of applying the function to a desired type. Again, the parallel on the C++ side is explicit template specialization.

```
f<int>(1);
```

would correspond to:

$$f\ \text{int}\ 1$$

Notice that the expression above uses both kinds of application. First, type application is used and f is applied to the type int. Then the resulting function $f\ \text{int}$ is applied to the actual value.

We have already seen what a free and bound variable is. To present typing rules for this extended STLC, the concept of a free variable need to be extended to the typing context. Notice that types of this extended system contain variables (unlike in STLC where the existence of type constants was assumed). Therefore it makes sense to talk about bound and free *type* variables (FTV). Let us define the free variables of a particular context in the following way:

$$\begin{aligned}\mathrm{FTV} &= \mathrm{FTV}(\{x_1 : T_1, x_2 : T_2, \ldots, x_n : T_n\}) \\ &= \mathrm{FTV}(T_1) \cup \mathrm{FTV}(T_2) \cup \cdots \cup \mathrm{FTV}(T_n)\end{aligned}$$

$$\begin{aligned}\mathrm{FTV}(x) &= \{x\} \\ \mathrm{FTV}(S \to T) &= \mathrm{FTV}(S) \cup \mathrm{FTV}(T) \\ \mathrm{FTV}(\forall x.T) &= \mathrm{FTV}(T) \setminus \{x\}\end{aligned}$$

3.4 System F

The typing rules for this extended system are:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}(\mathrm{var})$$

$$\frac{\Gamma \vdash M : S \to T \quad \Gamma \vdash N : S}{\Gamma \vdash MN : T}(\mathrm{app})$$

$$\frac{\Gamma, x : S \vdash M : T}{\Gamma \vdash (\lambda x : S.M) : S \to T}(\mathrm{abs})$$

$$\frac{\Gamma \vdash M : \forall x.T}{\Gamma \vdash MS : T[x := S]}(\mathrm{tapp})$$

$$\frac{\Gamma \vdash M : T \quad x \notin \mathrm{FTV}(\Gamma)}{\Gamma \vdash (\Lambda x.M) : \forall x.T}(\mathrm{tabs})$$

Lambda calculus extended in this way is known as System F[8][9]. The first three rules remain unchanged. Let us explore the last two rules. Type application rule simply expresses that if one has an expression that works for any type x, this expression can be applied to a type T, resulting in a specialized type where all the occurrences of x are replaced by T. For example:

$$\begin{aligned}&M : \forall x.x \to x \\ &M\ \mathrm{int} : \mathrm{int} \to \mathrm{int} \\ &M\ (\mathrm{int} \to \mathrm{int}) : (\mathrm{int} \to \mathrm{int}) \to \mathrm{int} \to \mathrm{int}\end{aligned}$$

Again, capture-avoiding substitution is used to deal with type variable shadowing.

The last rule refers to the creation of polymorphic expressions. Any type variable can be used to make the expression polymorphic (*generalized*) over it as long as the variable is "available" (it is not free in the typing context). Why do we require the type variable to not be free in the context? Let us consider this expression of untyped lambda calculus:

$$K = \lambda xy.x$$

The expect type of this expression when rewritten into System F is:

$$K_1 = (\Lambda a.\Lambda b.\lambda x : a.\lambda y : b.x) : \forall a.\forall b.a \to b \to a$$

or

$$K_2 = (\Lambda a.\lambda x : a.\Lambda b.\lambda y : b.x) : \forall a.a \to (\forall b.b \to a)$$

However, by allowing generalization even over free type variables, we can arrive at very different type:

$$K_3 = (\Lambda a.\Lambda b.\lambda x : a.\Lambda a.\lambda y : b.x) : \forall a.\forall b.a \to (\forall a.b \to a)$$

We can rename the variables and obtain:

$$K_3 : \forall a.\forall b.a \to (\forall c.b \to c)$$

K_1 and K_2 are very different from K_3. K_3 in fact breaks the guarantees expected from the type system. Consider:

$$K_3 \text{ int char} : \text{int} \to (\forall c.\text{char} \to c)$$
$$K_3 \text{ int char } 1 \text{ (int} \to \text{int)} \ 'a' : \text{int} \to \text{int}$$

K_3 simply takes the first parameter (in this case the number 1) and returns it. However, fully applied K_3 has the type $\text{int} \to \text{int}$. This means K_3 managed to convince the type system to use a number as a function, which is obviously a huge issue for the type system.

System F is more powerful than STLC. Despite this, it is still strongly normalizing (reduction terminates for any expression). But we can see that the additional features make System F much less convenient to use as a programming language. In particular, C++ function templates can automatically deduce the template parameter from context and because of that, the programmer often does not have to write down the type applications. We would like similar feature for System F.

Whether a program given in any particular system is valid should be verifiable by a computer. There are two kinds of verification: type checking (given Γ, E, T, decide whether $\Gamma \vdash E : T$) and type inference (given Γ and E, compute a type T such that $\Gamma \vdash E : T$ if such type exists, fail otherwise). Type checking is generally easier to do, because more information about the problem is provided. Type inference is harder but is a more desirable property – the whole type of an expression does not have to be provided because it can be computed from the expression itself (if it exists).

Both problems are decidable for STLC and System F. Let us try to remove the explicit type annotations from our language and see if these problems are still decidable. We will go back to expressions of untyped lambda calculus (we remove type abstraction, type application and the type of variable bound by abstraction). STLC can do without these hints, type checking and type inference are still decidable. However, both type checking and type inference become undecidable in System F without these hints[10].

Can we find a version of lambda calculus that is weaker than System F but still strong enough to express a subset of parametric polymorphism while still retaining the decidability of type checking and type inference?

3.5 Hindley-Milner type system

The first thing that comes to mind is the restriction of universal quantification (in a system without Λ, it can be thought of as implicit type abstraction). A polytype (or type scheme) is a subset of types of System F where all universal quantifiers are at the beginning. That is, each polytype looks like:

$$\forall a.\forall b.\ \ldots\ \forall x.T$$

where T is a type without any quantifiers. For example, the following type is not a polytype:

$$\forall x.x \to (\forall y.y \to x)$$

but it can be rewritten as a polytype without changing the meaning (each expression with the first type can be given the second type and vice versa):

$$\forall x.\forall y.x \to y \to x$$

A shortened form can be used:

$$\forall xy.x \to y \to x$$

However, some types do not have an equivalent polytype:

$$\forall y.(\forall x.x \to x) \to y \to y$$

There is no possible way to move the quantifier inside the parentheses without changing the meaning of the type. Generally, when the type is presented as a tree, quantifiers can be moved up only along the right subtrees without changing the meaning. A type without quantifiers (a monotype) is trivially a polytype.

Polytypes form the basis of Hindley-Milner type system[11][12]. The types of HM are:

$$\begin{aligned} Mono &::= x && \text{(variable)} \\ &\mid Mono \to Mono && \text{(function type)} \end{aligned}$$

$$\begin{aligned} Poly &::= Mono && \text{(monotype)} \\ &\mid \forall x.Poly && \text{(universal quantification)} \end{aligned}$$

As we have seen before, a type with universal quantifier can be specialized by substituting a type into the bound variable. Indeed, one can always get from the

type $\forall x.x \to x$ to $\text{int} \to \text{int}$ but not vice versa. We say that $\forall x.x \to x$ is more general than $\text{int} \to \text{int}$. A definition of ordering by "generality" on the types of HM is given by:

$$\frac{T = S[x_i := T_i] \quad y_i \notin \text{FTV}(\forall x_1.\ \ldots\ \forall x_n.S)}{\forall x_1.\ \ldots\ \forall x_n.S \sqsubseteq \forall y_i.\ \ldots\ \forall y_n.T}(\text{order})$$

$S \sqsubseteq T$ means that S is more general than T (or that T is more specialized than S). As expected, this definition coincides with the previously mentioned example:

$$\forall x.x \to x \sqsubseteq \text{int} \to \text{int}$$

When the type has multiple quantifiers, the ordering respects the fact that any bound variable can be specialized:

$$\forall xy.x \to y \to x \sqsubseteq \forall x.x \to \text{int} \to x \sqsubseteq \text{char} \to \text{int} \to \text{char}$$
$$\forall xy.x \to y \to x \sqsubseteq \forall y.\text{char} \to y \to \text{char} \sqsubseteq \text{char} \to \text{int} \to \text{char}$$

An important property of HM is that if an expression can be given more than one type, there always exists unique type (not considering variable renaming) that is the most general one among all such types. This is called the principal type (of an expression). That is, if one particular type is derived for an expression, the principal type can always be specialized to obtain the derived type.

We can now introduce the language of HM. It is based on untyped lambda calculus with an extra feature – local definitions via the *let* expression.

$$\begin{aligned} E ::= {} & x && \text{(variable)} \\ \mid {} & EE && \text{(application)} \\ \mid {} & \lambda x.E && \text{(abstraction)} \\ \mid {} & \text{let } x = E \text{ in } E && \text{(let)} \end{aligned}$$

The semantics of let expression can be given by translation to untyped lambda calculus:

$$\text{let } x = M \text{ in } N = (\lambda x.N)M$$

This raises a question: if the let expression can be expressed using only abstraction and application, why have the expression at all? The purpose of this expression is to enable parametric polymorphism in subexpressions, as we shall see later.

We can now present the typing rules for HM:

$$\frac{x : P \in \Gamma}{\Gamma \vdash x : P}(\text{var})$$

$$\frac{\Gamma \vdash M : T_1 \to T_2 \quad \Gamma \vdash N : T_1}{\Gamma \vdash MN : T_2}(\text{app})$$

$$\frac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash (\lambda x.M) : T_1 \to T_2}(\text{abs})$$

$$\frac{\Gamma \vdash E_1 : P \quad \Gamma, x : P \vdash E_2 : T}{\Gamma \vdash (\text{let } x = E_1 \text{ in } E_2) : T}(\text{let})$$

$$\frac{\Gamma \vdash E : P_1 \quad P_1 \sqsubseteq P_2}{\Gamma \vdash E : P_2}(\text{inst})$$

$$\frac{\Gamma \vdash E : P \quad x \notin \text{FTV}(\Gamma)}{\Gamma \vdash E : \forall x.P}(\text{gen})$$

The first three rules are basically the rules of STLC or System F. The last two rules correspond to $tapp$ and $tabs$ rules of System F. $inst$ rule talks about type specialization – if P_2 is more specialized than P_1, P_2 can be used in place of P_1. This is basically an implicit application of the $tapp$ rule of System F. gen is the generalization rule. Again, much like the $tabs$ rule of System F, if type variable x is available, it can be used to create a polytype (over the variable x).

The let rule is basically the combination of app and abs rules. Indeed, if the translation mentioned above is used ($\text{let } x = E_1 \text{ in } E_2$ as $(\lambda x.E_2)E_1$), a rule that looks suspiciously similar can be derived.

$$\frac{\Gamma \vdash E_1 : S \quad \dfrac{\Gamma, x : S \vdash E_2 : T}{\Gamma \vdash (\lambda x.E_2) : S \to T}(\text{abs})}{\Gamma \vdash (\lambda x.E_2)E_1 : T}(\text{app})$$

However, the main distinction comes from the careful use of polytypes and monotypes. In the rules above, polytypes are represented with the letter P and normal types (monotypes) with the letter T. Rules app and abs only deal with monotypes, they cannot use or create expression whose type is a polytype. Indeed, this guarantees that all types in the derivation remain polytypes. Consider what would happen if the app rule allowed polytypes:

$$M : \forall x.x \to x \to x$$
$$N : \forall y.y \to y$$

$$MN : (\forall y.y \to y) \to (\forall y.y \to y) = \forall x.(\forall y.y \to y) \to x \to x$$

As shown above, this type does not have a polytype equivalent.

However, we can afford to allow polytypes for the let expression (and the variable representing it). This allows us to have polymorphic subexpressions without risking the above problem. To see this in practice, consider the following function:

$$g : \text{int} \to \text{char} \to \text{int}$$

The following expression cannot be given a type using the rules above:

$$(\lambda f.g(f\ 1)(f\ 'a'))(\lambda x.x)$$

However, the equivalent let expression is perfectly fine:

$$\text{let } f = \lambda x.x \text{ in } g(f\ 1)(f\ 'a')$$

The difference is that the type of $\lambda x.x$ must be a monotype in the first example, while it can be given a polytype ($\forall x.x \to x$) in the second.

Type checking and type inference are both decidable for HM type system. In fact, type inference can be implemented in such way that it always finds the principal type of an expression if the expression can be given a type at all. This result is due to Damas and Milner[13].

3.6 Recursion in HM

Let us now focus on making the language given by HM type system more programmer friendly, in particular, we will add data types and recursion into the language.

STLC, System F and HM are all strongly normalizing. This means that arbitrary recursion is not part of the language and there are no fixed-point combinators. Recursion can be added simply by adding new constant into the language which represent a fixed-point combinator. For example:

$$\frac{}{\Gamma \vdash \text{fix} : \forall x.(x \to x) \to x}(\text{fix})$$

This constant has the following semantics:

$$\text{fix } f \rightsquigarrow f(\text{fix } f)$$

As we have shown in the previous chapter, such combinator can be used to implement a wide class of recursive functions.

3.7 Algebraic data types

Data types require modification of our language at both the value level and type level. Of course, our language already allows polymorphic functions, we should also support parametric polymorphism for data types. That is, instead of having a "hardcoded" IntCharPair, we should have a Pair *type constructor* that can be used to express int-char pairs by Pair int char, where int and char are *type parameters*.

A type constructor can be thought of as a simple function at type level, much like class templates in C++. Type constructor can be used as a type only when fully applied (each parameter has been provided). Such type constructors are called *nullary* (as they expect zero additional parameters). Pair itself is not a type, neither is Pair int. Only Pair int char is an actual type that can be used. This can be expressed

as a *type* of type constructor – usually called a *kind*. The kind of simple types and fully applied type constructors is known as $*$.

$$\begin{aligned}&\mathrm{Pair} : * \to * \to *\\&\mathrm{Pair\ int} : * \to *\\&\mathrm{Pair\ int\ char} : *\end{aligned}$$

From the given kinds it can be easily seen that Pair requires two additional type parameters before it can be used as a type. Similarly, $\mathrm{Pair\ int}$ requires one additional type parameter.

In the following section, we will refer to the pair type with an infix operator $\times$. That is, $\mathrm{Pair}\ S\ T = S \times T$. Pair types are also known as *products*.

Only type constructors with the kind $*$ can be used as a types. This gives us very simple metalanguage for describing types:

$$\begin{aligned}K ::= {} & * \\ & \mid * \to K\end{aligned}$$

$$\frac{\Gamma \vdash M : * \to K \quad \Gamma \vdash N : *}{\Gamma \vdash MN : K}(\mathrm{kapp})$$

Now that we know what should data types do at type level, we shall look at their value level component. We need to describe how one can create (introduce) a value of the data type and also how to then use such value (eliminate). For pairs, the introduction is a pair constructor:

$$\frac{\Gamma \vdash M : S \quad \Gamma \vdash N : T}{\Gamma \vdash \langle M, N \rangle : S \times T}(\text{pair-I})$$

There are two types of elimination that can be used. The first one is a pair of projections, each extracting one of the components of the pair. This is the usual style in which pairs are presented (as a negative type – elimination is given, introduction is derived).

$$\frac{\Gamma \vdash M : S \times T}{\Gamma \vdash \mathrm{fst}\ M : S}(\text{pair-E-1})$$

$$\frac{\Gamma \vdash M : S \times T}{\Gamma \vdash \mathrm{snd}\ M : T}(\text{pair-E-2})$$

The second one is more complicated. This eliminator has two parameters: the pair and the elimination function, which is passed both components of the pair and then produces a value with a type unrelated to the pair. This is the presentation as a positive type (introduction is given, elimination is derived)

$$\frac{\Gamma \vdash M : S \times T \quad \Gamma \vdash E : S \to T \to V}{\Gamma \vdash \text{elimpair}(M, E) : V}(\text{pair-E})$$

The semantics of all three eliminators can be shown by applying the eliminators to a pair where both values are known, this leads to the following reduction rules:

$$\text{fst } \langle X, Y \rangle \rightsquigarrow X$$
$$\text{snd } \langle X, Y \rangle \rightsquigarrow Y$$

$$\text{elimpair}(\langle X, Y \rangle, E) \rightsquigarrow EXY$$

These two presentations can be shown to be equivalent for pair type:

$$\text{fst } P = \text{elimpair}(P, \lambda xy.x)$$
$$\text{snd } P = \text{elimpair}(P, \lambda xy.y)$$

$$\text{elimpair}(P, E) = E(\text{fst } P)(\text{snd } P)$$

The other important type to consider is the choice type (also known as *disjunctive sum*). Again, much like pairs above, $+$ is a binary type constructor. $S + T$ can hold exactly one value – either a value of type S or a value of type T. This gives the following two introductions:

$$\frac{\Gamma \vdash E : S}{\Gamma \vdash \text{left } E : S + T}(\text{sum-I-1})$$

$$\frac{\Gamma \vdash E : T}{\Gamma \vdash \text{right } E : S + T}(\text{sum-I-2})$$

Disjunctive sum can basically only be presented as a positive type. Eliminations like fst and snd cannot be provided with the current system. Elimination is simply a matter of saying what to do for each case:

$$\frac{\Gamma \vdash E : S + T \quad \Gamma \vdash L : S \to V \quad \Gamma \vdash R : T \to V}{\Gamma \vdash \text{elimsum}(E, L, R) : V}(\text{sum-E})$$

With semantics given by:

$$\text{elimsum}(\text{left } E, L, R) \rightsquigarrow LE$$
$$\text{elimsum}(\text{right } E, L, R) \rightsquigarrow RE$$

Products and sums can be freely combined to create more complex types. However, constructing values gets very unwieldy very quickly.

$$\text{con} : D \to E \to F \to (A \times B) + (C + (D \times (E \times F)))$$
$$\text{con} = \lambda def.\text{right } (\text{right } (\langle d, \langle e, f \rangle \rangle))$$

We can introduce a general construction that will allow to express custom data types without the inconvenience brought by combination of products and sums. Each custom data type has its name and a list of parameters (this determines the arity of the type constructor). The actual values are made of an arbitrary number of choices and each choice is an arbitrary product. Haskell notation is used for expressing such types:

$$\text{data Maybe } a = \text{Just } a \mid \text{Nothing}$$

This introduces a unary type constructor Maybe ($\text{Maybe} : * \to *$). The actual value can be either a value of type a (introduction via constructor Just) or a trivial value (introduction via constructor Nothing).

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \text{Just } M : \text{Maybe } A}(\text{maybe-I-1})$$

$$\frac{}{\Gamma \vdash \text{Nothing} : \text{Maybe } A}(\text{maybe-I-2})$$

Each such type is presented as a positive type. This means the elimination can always be derived:

$$\frac{\Gamma \vdash E : \text{Maybe } A \quad \Gamma \vdash J : A \to V \quad \Gamma \vdash N : V}{\Gamma \vdash \text{maybe}(E, J, N) : V}(\text{maybe-E})$$

$$\text{maybe}(\text{Just } E, J, N) \rightsquigarrow JE$$
$$\text{maybe}(\text{Nothing}, J, N) \rightsquigarrow N$$

In general:

$$\text{data D } x_1 \dots x_n = \text{C}_1\ x_{11} \dots x_{1i_l} \mid \dots \mid \text{C}_m\ x_{m1} \dots x_{mi_n}$$

The above introduces an n-ary type constructor D and m constructors named C_1 to C_m. Each constructor represents a product of types x_{ij} (each such type must be either a type constant or a type variable bound by the type constructor D).

$$\frac{\Gamma \vdash E_1 : x_{j1} \quad \Gamma \vdash E_2 : x_{j2} \quad \dots \quad \Gamma \vdash E_m : x_{jm}}{\Gamma \vdash \text{C}_j E_1 E_2 \dots E_m : \text{D } x_1 \dots x_n}(\text{d-I-j})$$

Again, much like above, the eliminator can be given simply be specifying what to do for each constructor.

$$\frac{\Gamma \vdash E : \text{D } x_1 \dots x_n \quad \Gamma \vdash \text{CC}_1 : x_{11} \to \dots \to x_{1i_l} \to V \quad \dots}{\Gamma \vdash \text{d}(\text{CC}_1, \dots, \text{CC}_m, E) : V}(\text{d-E})$$

And the corresponding reductions:

$$\mathrm{d}(\mathrm{CC}_1, \ldots, \mathrm{CC}_m, \mathrm{C}_1\ a\ b\ c\ \ldots) \rightsquigarrow \mathrm{CC}_1\ a\ b\ c\ \ldots$$
$$\ldots$$
$$\mathrm{d}(\mathrm{CC}_1, \ldots, \mathrm{CC}_m, \mathrm{C}_m\ a\ b\ c\ \ldots) \rightsquigarrow \mathrm{CC}_m\ a\ b\ c\ \ldots$$

We will even allow the use of D itself in the types of constructors. This creates *recursive* data types. For example, singly-linked lists can be expressed in the following way:

$$\text{data List } a = \text{Node } a\ (\text{List } a) \mid \text{End}$$

A list of numbers 1, 2 and 3 would be expressed as:

$$\text{Node 1 (Node 2 (Node 3 End))}$$

The eliminator can be used to write a function which checks whether a list is empty:

$$\text{isEmpty} = \lambda x.\text{list } (\lambda ab.\text{false})\ \text{true}\ x$$

$$\begin{aligned}\text{isEmpty End} &= (\lambda x.\text{list } (\lambda ab.\text{false})\ \text{true}\ x)\ \text{End} \\ &\rightsquigarrow \text{list } (\lambda ab.\text{false})\ \text{true End} \\ &\rightsquigarrow \text{true}\end{aligned}$$

$$\begin{aligned}\text{isEmpty (Node } x\ xs) &= \ldots \\ &\rightsquigarrow \text{list } (\lambda ab.\text{false})\ \text{true (Node } x\ xs) \\ &\rightsquigarrow \text{false}\end{aligned}$$

More than one recursive occurrence is also allowed:

$$\text{data Tree } a = \text{Node } a\ (\text{Tree } a)\ (\text{Tree } a) \mid \text{End}$$

The language used for describing template metaprograms is a version of lambda calculus. The base is the syntax of untyped lambda calculus. We add let expressions, integer and boolean literals, type annotations and top-level definitions of constants (and functions), data types and assumptions. We use Hindley-Milner type system extended with custom data type definitions.

One very important feature of HM is that once type checking is finished, the types are no longer important to evaluation (runtime). This means that we do not have to concern ourselves with the translation of types, type annotations and assumptions.

4 Translation

As we have seen, template metaprogramming is very capable of expressing programs in functional style. There is however one big issue preventing straightforward translation of lambda calculus into C++ templates – templates do not handle higher-order functions very well. Consider the identity function:

```
template <typename T>
struct id { typedef typename T::type type; };
```

`id` works perfectly fine when given non-function values:

```
id<Int<4>>::type == Int<4>
```

But `id` is expected to be able to handle function parameters, such as:

```
id<id>::type == id
```

While there are some higher-order functions that can be expressed, most of them cannot. Recall that lambda calculus does not have functions of multiple parameters, each function is unary and n-ary functions are expressed via currying. Functions returning functions is the single biggest issue preventing the translation. Templates can express function parameters via the template-template parameter, but `typedef` cannot be used to return such values.

4.1 Higher-order functions in C++

The way to solve this issue is to unify simple types (that can be manipulated easily – both as parameters and as "return" values via `typedef`) with templates. Luckily, C++ allows the use of templates inside a class. Each template can therefore be associated with a simple type simply by wrapping the template in another class, which can then be manipulated in the usual way. More flexible identity function can thus be implemented:

```
struct id {
    struct type {
        template <typename T>
        struct apply {
            typedef typename T::type type;
        };
    };
};
```

Some syntactic convenience is lost on the C++ side, but we gain elegant implementation of higher-order functions. Notice the extra layer of indirection via

the `type` structure. This is to stay consistent with the previous definitions for simple types:

```
struct Two {
    typedef Int<2> type;
};
```

Much like the structure `Two` above, `id` is name for the expression, the expression itself is hidden in the inner type `type`. Each function expression has inner template `apply`, which is the place where the actual parameter can be provided:

```
id::type::apply<Two>::type == Int<2>
```

Notice that the template `apply` expects not an expression, but a name (something with inner type `type`). Again, this is for consistency reasons – we can treat the application `f x` in the same way no matter what `f` and `x` represent (be it variables bound by abstraction or previously defined constants). A simple helper structure can be defined which does the wrapping on fly:

```
template <typename T>
struct wrap {
    typedef T type;
};

id::type::apply<wrap<Int<2>>>::type == Int<2>
```

Notice that the previously impossible application is now trivial.

```
id::type::apply<id>::type == id::type
```

The expression indeed behaves as expected:

```
id id 2 → id 2 → 2

id::type::apply<id>::type::apply<wrap<Int<2>>>::type == Int<2>
```

4.2 Translating untyped lambda calculus

This unification of simple types with templates is the basis of the translation. So far only translation of the untyped lambda calculus can be given:

```
translate(x, name) :=
    struct name {
        typedef typename x::type type;
    };
```

```
-- \x -> E = λx. E
translate(\x -> E, name) :=
    struct name {
        struct type {
            template <typename x>
            struct apply {
                translate(E, localname)
                typedef typename localname::type type;
            };
        };
    };


translate(M N, name) :=
    struct name {
        translate(M, localname1)
        translate(N, localname2)
        typedef typename
            localname1::type::template apply<localname2>::type
            type;
    };
```

There are two interesting things to note about the translation. The use of the keyword `template` in the translation of application is not the usual template definition known from before. Much like the `typename` keyword, `template` must be used to distinguish between a member named `apply` and a template named `apply`. Funnily enough, while the statement is usually unambiguous given the angle brackets, there are cases where the angle brackets can express less-than and greater-than operations.

The other thing to note is that each subexpression is named. This means that any subexpression can be directly used with function application and does not need to be wrapped in the `wrap` structure.

4.3 Translating syntactic extensions

Recall that the extended language has additional syntactic features at the expression level: type annotations, literals, fixed-point combinator and let expression. Translation for literals is fairly simple:

```
translate(True, name) :=
    struct name {
        typedef Bool<true> type;
    };


translate(False, name) :=
    struct name {
        typedef Bool<false> type;
    };


translate(n, name) :=
    struct name {
        typedef Int<n> type;
    };
```

As mentioned above, type annotations can be skipped:

```
translate(E : T, name) := translate(E, name)
```

Fixed-point combinator is translated simply as the application of predefined function `fix`:

```
translate(fix x. E, name) := translate(fix (\x -> E), name)
```

And finally, the let expression. The language offers a syntactic sugar for chains of let expressions, such as:

```
let x1 = E1 in let x2 = E2 in E3
```

The above can be shortened into the more readable:

```
let x1 = E1; x2 = E2 in E3
```

```
translate(let x1 = E1; ...; xn = En in E, name) :=
    struct name {
        translate(x1 = E1)
        ...
        translate(xn = En)
        translate(E, localname)
        typedef typename localname::type type;
    };
```

The translation of (local) definitions such as `x1 = E1` is used both in let expression and for top-level definitions:

```
translate(x = E) :=
    struct x {
        translate(E, localname)
        typedef typename localname::type type;
    };
```

This variant of translate does not have the second argument. The name is already specified in the definition. As types are erased in translation, top-level type annotations and assumptions are simply ignored. What remains is the translation of data type definitions.

4.4 Translating user-defined data types

At first, the encoding of data types into C++ templates has to be defined. Recall that each data type is a series of sums (choices), each containing a product (tuple). Products can be expressed very easily:

```
template <typename S, typename T>
struct Pair { };
```

```
Pair<Int<4>, Int<3>>
```

In fact, with the use of variadic templates, a tuple with arbitrarily many elements can be defined:

```
template <typename... T>
struct Tuple { };
```

For disjoint sums, an already existing sum type is required. The simplest sum type that can be used in templates is boolean.

```
template <bool B, typename T>
struct Sum { };

Sum<true, Int<1>> (= left 1)
Sum<false, Bool<true>> (= right true)
```

C++ does not have a sum type that offers arbitrarily many choices, but integers will do:

```
template <int I, typename T>
struct ManySum { };
```

The two definitions can be combined:

```
template <int Choice, typename... Fields>
struct Data { };
```

While this definition can perfectly represent any user defined type of the language, there is a small issue with template specialization (C++ standard does not allow certain kinds of template specialization). This can be fixed by adding a dummy parameter. The final definition is thus:

```
template <int Choice, typename Dummy, typename... Fields>
struct __data { };
```

The choice of dummy type in the actual encoding does not matter, but we still use a special type for consistency and readability:

```
struct __dummy { };
```

As an example, consider the following program:

```
data List a = Node a (List a) | End

list = Node 1 End
```

and its translation:

```
list::type == __data<0, __dummy, Int<1>, __data<1, __dummy>>
```

The first use of the template `__data` has 4 template parameters: the number `0` (representing the 0^{th} constructor – `Node`), the dummy parameter, the head of the list – `Int<1>` and finally the rest of the list. It consists of another template `__data`, but this time with only two parameters: the number `1` (representing the 1^{st} constructor – `End`) and the dummy parameter. Since the `End` constructor does not have any data fields, nothing more is needed.

There is no need to translate type constructors, only value constructors and eliminators. Constructors are translated as a top level definition of curried function of the form:

```
C = \a b c... -> C a b c ...
```

The number of parameters is given by the number of fields of the constructor `C`. In the previous chapter, constructors were not presented in the function form. Indeed, the constructor needed to be fully saturated (all fields provided). However, it is more convenient to be able to partially apply the constructor. For instance, consider the mapping function:

```
map = fix rec. \f -> list (\x xs -> Node (f x) (rec f xs)) End

map (\x -> x + 1) (Node 1 (Node 2 End)) → Node 2 (Node 3 End)
```

Inserting new element at the beginning of each list can be done simply by partially applying the `Node` constructor. For readability, we write lists in brackets.

```
map (Node 2) [[], [3,4], [5]] → [[2], [2,3,4], [2,5]]
```

The translation of abstraction is given above, only the translation of the constructor expression has to be provided:

```
translate(C a b c ..., name) :=
    struct name {
        typedef
            __data<ctorNumber(C), __dummy, typename a::type, ...>
            type;
    };
```

`ctorNumber(C)` is simply the order in which the constructor `C` appears in the definition.

The last part of translation is the eliminator. Its name is derived automatically: it is simply the lower-case name of type constructor (as type constructors are required to start with capital letter). Recall that the eliminator has $n+1$ parameters: one function for each of the n constructors and then the actual structure. Therefore, the first part of translation can be described by the above definitions:

```
translate(elim for T) :=
    translate(t = \handle-ctor-0 ... handle-ctor-n s ->
        elim-body)
```

The translation of `\s -> elim-body` is where the magic happens. Recall that template specialization can be employed for case analysis (which constructor was used to construct the given value). The intermediary step can be cut and the template specialization done directly in the last abstraction `\s -> elim-body`.

```
translate(\s -> elim-body, name) :=
struct name {
    struct type {
        template <typename>
        struct apply_alt;

        template <typename dummy, typename field1,
            ..., typename fieldi>
        struct apply_alt<__data<0, dummy, field1, ..., fieldi>> {
            translate(handle-ctor-0 wrap(field1) ...
                wrap(fieldi), localname0)
            typedef typename localname0::type type;
        };

        ...

        template <typename dummy, typename field1,
            ..., typename fieldj>
        struct apply_alt<__data<n, dummy, field1, ..., fieldj>> {
            translate(handle-ctor-n wrap(field1) ...
                wrap(fieldj), localnamen)
            typedef typename localnamen::type type;
        };

        template <typename structArg>
        struct apply {
            typedef typename
                apply_alt<typename structArg::type>::type
                type;
        };
    };
};
```

This code is rather complicated, let us go over it in detail. The base of case analysis is the template `apply_alt`. Its only template argument is the actual structure (`apply_alt` expects a type of the form `__data<...>`). This template has a specialization for each constructor, the specialization is selected based on the constructor number.

Once selected, the actual data stored in the data fields is associated with the other template parameters of the specialization (`field1` through `fieldi`). This extracted data is almost ready to be passed to the function handling this particular constructor (recall that this function is given as part of the preliminary translation of the first n abstractions). However, application expects the parameter to have an inner type `type`, therefore we need to wrap the extracted data before using it.

As a final step, the template `apply` is defined. Its purpose is simply to extract the inner type of the last passed parameter and use it with the `apply_alt` template.

4.5 Implementing the runtime

We can now translate any part of the language, however, the translation makes use of various predefined types. Most notably, it makes use of the fixed-point combinator `fix`, which is not defined at the level of the language. Since HM is strongly normalizing, there are no fixed-point combinators, it therefore seems that this combinator has to be implemented manually in C++.

Direct definition via the equation $\text{fix}\ f = f(\text{fix}\ f)$ is not possible. The `fix` type is used as it is defined, therefore C++ declines its definition due to the use of an incomplete type. However, a trick can be used. An auxiliary combinator $\text{fix-alt}\ f\ n = f(\text{fix-alt}\ f\ (n+1))$ has the same behavior but does not suffer from the above problem. The original combinator can be recovered by setting $\text{fix}\ f = \text{fix-alt}\ f\ 0$.

```
struct fix {
    struct type {
        template <typename f, int i>
        struct apply_rec {
            typedef typename
                f::type::template apply<
                        apply_rec<f, i + 1>
                    >::type type;
        };

        template <typename f>
        struct apply {
            typedef typename apply_rec<f, 0>::type type;
        };
    };
};
```

However, we do not use pure HM. The language has been extended with data types. It turns out that there is a data type that can be used to express Y combinator even in typed language.

```
data Rec a = In (Rec a -> a);

out : Rec a -> Rec a -> a;
out = rec \x -> x;

y : (a -> a) -> a;
y = \f -> (\x -> f (out x x)) (In \x -> f (out x x))
```

Notice that without `In` and `out`, the expression defining `y` is indeed the original Y combinator. The existence of fixed-point combinator is of course a desired property from the programmer's perspective, however, if needed, there are ways of adding data types without losing strong normalization. The definition of data types can be

restricted to only allow so called *strictly positive* types. Such types have additional restrictions placed on recursive occurrences that disallow problematic types such as `Rec` above.

The other predefined types are `__data`, `__dummy`, `Int` and `Bool` (for literals). Some operations on integers and booleans are also needed. The structure is basically the same, let us just introduce an example of unary operation, binary operation and the `if` expression.

```
struct not_ {
    struct type {
        template <typename A>
        struct apply {
            typedef Bool<(!A::type::value)> type;
        };
    };
};

struct plus {
    struct type {
        template <typename A>
        struct apply {
            struct type {
                template <typename B>
                struct apply {
                    typedef Int<
                            (A::type::value + B::type::value)
                        > type;
                };
            };
        };
    };
};
```

Notice that the name `not` cannot be used (as it is one of C++ reserved keywords). For the same reason, `if` is replaced with `if_`.

```
struct if_ {
    struct type {
        template <typename A>
        struct apply {
            struct type {
                template <typename B>
                struct apply {
                    struct type {
                        template <typename C>
                        struct apply {
                            template <bool b, typename __dummy>
                            struct __check;

                            template <typename __dummy>
                            struct __check<true, __dummy> {
                                typedef typename B::type type;
                            };
```

```
                                template <typename __dummy>
                                struct __check<false, __dummy> {
                                    typedef typename C::type type;
                                };

                                typedef typename
                                    __check<
                                        A::type::value, __dummy
                                    >::type type;
                            };
                        };
                    };
                };
            };
        };
};
```

There is something very suspicious about the `if_` expression. It is very similar to the translation of eliminators. Indeed, the `if` expression is the eliminator for the boolean type:

```
data Bool = True | False;

if = \cond t f -> bool t f cond

-- if true  a b → bool a b true  → a
-- if false a b → bool a b false → b
```

4.6 Syntax of the language

And lastly, some additional information about the language. Each file represents a module. A module is basically a list of top-level entities, each separated by a semicolon. Top-level entities are, as mentioned above, data definitions, value definitions, type annotations (top-level type annotations are known as type signatures) and assumptions. Instead of using universal quantification, type variables are implicitly quantified – the language represents type variables with uncapitalized names and concrete types with capitalized names. For instance:

```
a -> Int -> b = ∀a b. a → Int → b
int -> A      = ∀int. int → A
```

where `Int` and `A` are concrete types. In the same way, data definitions require type and value constructors to start with a capital letter.

```
data bool = True | False
    -- error: bool must be capitalized

data Bool = true | false
    -- error: true and false must be capitalized

data A a = Con1 a | Con2 int
    -- error: unknown type variable int
```

```
data A Int = Con1
    -- error: Int cannot be used as type variable

data A a = Con1 a | Con2 Int
    -- correct
```

An empty data type can also be defined. Such type can be useful in certain circumstances:

```
data Empty    -- no constructors
```

The same distinction is used at the value level. Capitalized names refer to constructors, uncapitalized names to everything else (defined values, variables introduced in abstraction, eliminators). Value definitions offer syntactic sugar for defining functions. There is a shorthand notation for lambda abstractions:

```
f = \a b c -> E
f a b c = E
```

Type signatures simply ask whether the inferred type of a top-level definition coincides with the given type. This is useful as a form of documentation or as a way to make sure that the defined value has the expected type.

```
f : a -> b -> a;
f x y = x
-- correct

f : a -> Int -> a;
f x y = x
-- also correct, a -> Int -> a is a
-- specialization of the inferred type a -> b -> a

f : a -> b;
f x = x
-- incorrect, a -> b does not match the inferred a -> a
```

An assumption starts with the keyword `assume` and is basically a type signature without corresponding definition. Assumptions let the compiler know that certain value with the given type exists and is defined elsewhere. This is useful for interaction between the language and user defined template metaprograms. For ease of use, the existence of abstract type `Type` is also assumed that represents a C++ type. This way, the user can implement template metaprograms operating on types (such as `add_pointer`) and then bring those in scope as functions operating on `Type`:

```
compose : (b -> c) -> (a -> b) -> a -> c
compose f g x = f (g x)

assume add_pointer : Type -> Type
```

```
add_two_pointers : Type -> Type
add_two_pointers = compose add_pointer add_pointer


-- in C++
add_two_pointers::type::apply<wrap<int>>::type == int**
```

At the level of expression, there are two important changes. Since the lambda symbol is not available on most keyboards, it is replaced with the backslash symbol. And for readability's sake, the dot symbol in abstraction is replaced with an arrow (as is the Unicode arrow: `->`). Many predefined operations on integers and booleans have their own prefix and infix variant, for instance:

```
four : Int;
four = plus 2 2


four : Int;
four = 2 + 2
```

`--` are line comments, `{-` and `-}` are multiline comments.

The use of fixed-point combinator is implicit, the user can write normal recursive definitions and they are then translated using the fixed-point combinator.

```
fac : Int -> Int;
fac n = if_ (n == 0) 0 (n * fac (n - 1))
```

If needed, the combinator can be written using the data type trick above or simply by defining:

```
fix_ : (a -> a) -> a
fix_ f = f (fix_ f)
```

Full grammar for the language can be found in the appendix.

5 Implementation

The compiler can be broken down into roughly four parts: parser, type checker, translator and the glue that holds it all together. Let us go over the modules in detail, starting with modules that deal with parsing.

5.1 Lexing and parsing

The module `Compiler.AST` holds the definition of the abstract syntax tree of the language. It also implements type comparison up to α-equivalence, so that:

```
∀a b. a -> b  =  ∀a b. b -> a
```

`Compiler.Lexer` defines the basic lexer for the language with the help of the *parsec* library[14]. Parsec provides the function `makeTokenParser`, which uses the information about reserved keywords (both from C++ and the language itself), operators, identifier names and comment style to create a set of token parsers that (among other things) automatically handle white space (comments included).

`Compiler.Parser` is the home of the actual parser. Again, like `Compiler.Lexer`, `Compiler.Parser` makes extensive use of the parsing library parsec. The applicative (and sometimes monadic) interface allows the definition of parsers in a style very close to actual BNF grammar definition. All the intermediate parsers for expressions, top-level entities, types, etc are combined into the parser `file`, which parses the whole file and stores the result (if any) as an abstract syntax tree as defined by `Compiler.AST`.

5.2 Type checking

Type checking happens unsurprisingly in the modules `Compiler.TypeChecking.*`.

`Compiler.TypeChecking.Free` defines how to extract free type variables from various types (monotypes, polytypes – type schemes, type contexts).

`Compiler.TypeChecking.Error` defines all kinds of errors that can arise during type checking. The final error carries not only what caused the error but also the location of the error that can then be reported alongside the error to the user. Notice that there is a wide range of errors: unification errors (types do not match), kind errors (`f : Int Int -> Int`), scope errors (undefined values, types, redefinitions, etc) and type errors (type is too general).

Substitutions are defined in the module `Compiler.TypeChecking.Subst`. We already know that a (capture-avoiding) substitution is simply the act of replacing a variable with an expression (or, in this case, a type). Substitutions as defined in this module are simultaneous – that is, they replace multiple variables at once. The most

important operation is the application of substitution. They can be applied not only to simple types, but also polytypes (type schemes) and even whole contexts.

A unifier of types S and T is a simultaneous substitution sub such that $\text{apply}(sub, S) = \text{apply}(sub, T)$. In other words, two types S and T can be unified if we can substitute variables in both types, such that the types are the same after the substitution. A *most general unifier* of S and T is simply a unifier that can be specialized (by further substitution) to any other unifier of S and T. This is a crucial part of the Damas-Milner type inference algorithm. MGU is implemented as the function unify in the module `Compiler.TypeChecking.Unify`.

`Compiler.TypeChecking.Context` gives the definitions of various kinds of contexts used during the type checking. It is not only the typing context used in the rules (Γ), but also kind context (which stores the kinds of custom data types), context for type signatures and error locations.

Type class instance for the monad stack used for type inference is defined in the module `Compiler.TypeChecking.Infer.Monad`. The monad stack consists of 4 monad transformers – two instances of state monad, one reader monad and finally error monad. The first state monad gives access to global substitution (that is updated throughout the inference), the second state monad is used to generate fresh variables, the reader monad provides type inference and error contexts and finally, the error monad can be used to stop the computation upon encountering an error. The rest of the module consists of helper functions providing access to each monad on the stack.

This leaves us with `Compiler.TypeChecking.Infer`. This is the implementation of Damas-Milner type inference algorithm. This particular variant is inspired by *Typing Haskell in Haskell*[15]. We present a quick summary of the algorithm. The main part of the inference algorithm is type inference for expressions. Variables are simply looked up in the context (and error is reported if they are not found), the resulting polytype is instantiated with fresh type variables and returned.

```
infer(f, Γ):
    t = find(f, Γ)
    return instantiate(t)
```

For instance:

```
Γ = {f : ∀a b. a -> b -> a}

infer(f, Γ) = fresh1 -> fresh2 -> fresh1
```

Fresh type variables can then be unified with other types or – if not used – generalized back to quantified variables.

The type of an abstraction is inferred by giving the parameter a fresh type variable, adding it to the context and inferring the type of the abstraction body. Note that this extended context is only used locally, once we leave the abstraction, the information about the parameter is thrown away (as it is no longer needed and could interfere in

case of variable shadowing). The resulting type is given by both the fresh type variable and the inferred type of the abstraction body.

```
infer(\x -> E, Γ):
    Γ' = Γ ∪ {x : fresh}
    type-of-E = infer(E, Γ')
    return (x -> type-of-E)
```

Inferring application is done by inferring types of both subexpressions and then making sure the types can be matched.

```
infer(M N, Γ):
    type-of-M = infer(M, Γ)
    type-of-N = infer(N, Γ)
    unify(type-of-N -> fresh, type-of-M)
    return fresh
```

The type of subexpression `M` should of course be a function type, type of its parameter should match the inferred type of `N` (as `N` is the actual parameter). The resulting type is unknown, therefore it is simply represented as a fresh variable that can be specified by further unification.

Recall that let expressions in HM are crucial for parametric polymorphism in subexpressions, therefore simple type inference of the subexpression is not enough. This type must first be generalized (made polymorphic) before it can be added to the context and used later.

```
infer(let x = E1 in E2, Γ):
    type-of-E1 = infer(E1, Γ)
    g = generalize(type-of-E1, Γ)
    return infer(E2, Γ ∪ {x : g})
```

Generalization can be done by taking all type variables of the type and universally quantifying those that are not among the free variables of the context:

```
generalize(type, Γ):
    {var1, ..., varn} = FTV(type) \ FTV(Γ)
    return (∀var1 ... varn. Type)
```

This is the basis of Damas-Milner type inference algorithm. We left many details unspecified – particularly the handling of global substitution, inference in presence of type annotations and how data types influence the inference. If more detail is required, the reader is advised to check the implementation or read one of the above mentioned resources.

5.3 Compilation

Translation happens in the module `Compiler.Compile`. It implements the translation as defined in previous chapter. The function `compileModule` is the main translation operation – it takes AST of the whole module and produces a string with C++ code.

This leaves us with few modules that glue the rest together.

`Compiler.Transform` implements two important AST transformations – renaming and recursion elimination. Recall that type checking already expects the code to be free of explicit recursion – every recursive definition is type checked with the use of fixed-point operator fix in mind. Therefore before type checking happens, every recursive definition (both local and top-level ones) need to be transformed to use fixed-point operator. This is done by the function `fixifyModule`.

After type checking is done and errors (if present) are reported to the user, the actual names of variables bound in abstractions and fixed-point operators or the names of local definition inside a let expression are no longer needed. To make sure there is no unwanted name conflicts on the C++ side, each such name is replaced with a freshly created one.

5.4 Auxiliary modules

Reporting errors needs some context – the expression where the error happened, where this expression is in the code, etc. However, at the point of type checking, only AST is available. To provide readable error messages, there needs to be a way of translating AST back to readable code. This translation is available in the module `Compiler.Pretty` (the act of translating AST back to code is known as pretty printing). While pretty printers are usually not guaranteed to produce the same code we started with (especially since comments are stripped in parsing), parsing a pretty printed AST produces the original AST. The only exception is the fixed-point operator in AST as it has no corresponding code.

The module `Default` contains default contexts for type checking (predefined types and operations on integers and booleans – `defaultTICtx`), the name of default output file (`defaultOutput`), the name of default runtime directory (`defaultInclude`) and the names of runtime files (`defaultRuntime`).

Few helper functions that do not fit anywhere else are in the `Utility` module.

Error reporting makes extensive use of the modules `Compiler.TypeChecking.Error` and `Compiler.Pretty`. It itself is in the module `Report`. Functions in this module do not only report the errors but also the context in which the error occurred.

`Options` module is responsible for handling command line options. *optparse-applicative* library[16] is behind the implementation.

And finally, the `Main` module combines all these together into an actual program. The process of compilation is simple. First, command line options are parsed using the optparse-applicative library (`execParser` function), the code is read from the input file (specified by the options) and then parsed using the parsec library and `Compiler.Parser` module. Recursion is eliminated via the `fixify` function from `Compiler.Transform`. Type checking is performed and if no error occurred, the AST is ready for translation. Recall that we do not need the result (i.e., the inferred types) since the language has the property of type erasure – types do not matter during execution. And finally, the names in the AST are replaced with fresh ones and the

module is translated using the functions from `Compiler.Compile` and then written to an output file specified on the command line.

6 Usage and examples

The compiler is distributed as a cabal package. To install a cabal package, the users needs GHC[17] and of course Cabal[18]. Since the code does not make use of any GHC-specific language extensions, it can be compiled by a wide variety of GHC versions, though it has been only tested with GHC 7.4 or newer. These two programs can be easily obtained as a part of the *Haskell Platform*[19] – the all-in-one starter package for Haskell programming. To compile and install the compiler, simply run `cabal install` in the base directory (where the `norri.cabal` file is). Few libraries need to be downloaded first, but if everything goes well, this should be handled by `cabal install`.

After `cabal install` finishes, the binary is placed into the `bin` directory (usually `~/.cabal/bin` or `%APPDATA%\cabal\bin`). If Haskell Platform is installed, this directory should be in `PATH` by default.

6.1 Command line options

The compiler can be executed with the following options:

`-h` or `--help` prints a short overview of all command line options.

`-o` or `--output` specifies a file where the resulting C++ code should be written. By convention, such files should use the `.hpp` suffix. When not specified, `a.hpp` is used.

`-a` or `--addruntime` instructs the compiler to copy the runtime files directly into the compiled code (instead of only using the `#include` directive).

`-i` or `--includedir` is the location of runtime. This is used both for compilation with and without the `-a` flag. Even if the `-a` flag is not used, the location is required for `#include` directive. When not specified, `runtime` is used.

`-n` or `--noguards` produces a code without the `#ifndef` guards used for C++ headers.

The final command line parameter is the file to be compiled. By convention, although the compiler does not care, such files use the `.nri` suffix (from norri).

If any errors occur during the compilation, non-zero return code is returned and an error is printed to the standard error stream. If the compilation is successful, the compiled metaprogram is written into the specified file and zero is returned.

6.2 First program

Let us go over some examples and ways in which the compiled metaprogram can be used.

```
-- ex1.nri
number : a;
number = 5
```

Compiling the above code produces an error: the declared type of number is too generic. We state that number has a type `a` for all possible types `a`. This is of course false, `number` has to have the type `Int`. After fixing the type error, this code can be successfully compiled.

```
-- ex2.nri
number : Int;
number = 5
```

When we open the compiled `a.hpp` file, this is the result:

```
#ifndef NORRI_OUTPUT_A
#define NORRI_OUTPUT_A
#include "runtime/fix.hpp"
#include "runtime/data.hpp"


struct number
{
struct __def
{
typedef Int<5> type;
};
typedef typename __def::type type;
};


#endif // NORRI_OUTPUT_A
```

The code has everything we expected. Include guards are there, named after the header file. Both runtime files are included and the value `number` is represented by a structure number with an inner type `type`, such that:

```
number::type == Int<5>
```

There are a number of ways in which the previous statement can be checked. The one we use here relies on making a deliberate type error, such as:

```
// ex2a.cpp
#include “a.hpp”


int main() { number::type x = 0; }
```

Compiling the previous code with *gcc* (`g++ -std=c++11 a.cpp`) yields the following error:

```
a.cpp: In function 'int main()':
a.cpp:3:31: error: conversion from 'int' to non-scalar type
'number::type {aka Int<5>}' requested
...
```

The "normal form" (as evaluated by the compiler) is described in the braces following `number::type`. Since the type is figured out during compilation, its inner member `value` can be used to initialize other constant expressions:

```
// ex2b.cpp
static const int compile_time_constant = number::type::value;

int main() { }
```

6.3 Simple functions

Let us compile a more complex expression.

```
-- ex3.nri
id : a -> a;
id x = x
```

The compiled metaprogram is far more involved. We cut the unneeded includes and guards:

```
struct id
{
struct __def
{
struct type
{
template <typename _T0>
struct apply
{
struct __local0
{
typedef typename _T0::type type;
};
typedef typename __local0::type type;
};

};

};
typedef typename __def::type type;
};
```

While this code contains some extra indirection, it is indeed the identity function presented in previous chapters. It can be used in the same way:

```
template <typename T>
struct wrap { typedef T type; };

id::type::apply<wrap<Int<2>>>::type == Int<2>
```

6.4 Higher-order functions

The translation is indeed robust enough to handle higher-order functions.

```
-- ex4.nri
twice : (a -> a) -> a -> a;
twice f x = f (f x);

result : Int;
result = twice (plus 4) 1

result::type == Int<9>
```

Here we use the predefined function `plus` (this is the prefix version of the infix `+`). Notice that by applying `plus` to `4`, we obtain `plus 4 : Int -> Int`, which can then be used with the function `twice`.

6.5 Custom data types

Compiling a data type definition creates a structure for each constructor and a structure for the eliminator. Thanks to type erasure, nothing more is needed.

```
-- ex5.nri
data Maybe a = Just a | Nothing
```

The above code compiles into:

```
struct Just { ... };
struct Nothing { ... };
struct maybe { ... };
```

Recall that this definition introduces these values into the typing context:

```
Just : a -> Maybe a
Nothing : Maybe a
maybe : (a -> b) -> b -> Maybe a -> b
```

Recursive types are also available:

```
data List a = Node a (List a) | End
```

After compilation, recursive and non-recursive types are indistinguishable. In fact, the previous definition compiles into the exact same code (sans renaming) as:

```
data ListStep a b = Node a b | End
```

The difference between `List` and `ListStep` is only relevant during type checking. With that out of the way, let us implement some functions that operate on lists.

```
-- ex6.nri
data List a = Node a (List a) | End;
```

```
replicate : Int -> a -> List a;
replicate n x =
    let go m = if_ (m <= 0)
                   End
                   (Node x (go (m - 1)))
    in  go n;


map : (a -> b) -> List a -> List b;
map f = list (\x xs -> Node (f x) (map f xs)) End;


sum : List Int -> Int;
sum = list (\x xs -> x + sum xs) 0;


result : Int;
result = sum (map (plus 2) (replicate 4 1))
```

`replicate n x` is a list consisting of `n` copies of `x`. Notice the use of inner function `go`. This is sometimes known as worker/wrapper transformation and is frequently used in Haskell and similar languages. The recursion happens only on the numeric parameter, the element to be replicated would be just passed around unchanged. We can factor this out with a local function.

`map` is a higher-order function that applies a given function (the first parameter) to every element of the list. If the list is non-empty (consisting of head `x` and tail `xs`), the function `f` is applied to the head and `map f` is applied to the tail, then these two are reconstructed back into a list. If the list is empty, there is nothing to be done and empty list is returned.

`sum` behaves in similar way. The difference is that instead of reconstructing the list, it simply adds the numeric value in head to the rest of the sum obtained from the list tail. Let us see what the expected value should be and then compare. A shorthand notation for lists is used.

```
sum (map (plus 2) (replicate 4 1))
    → sum (map (plus 2) [1, 1, 1, 1])
    → sum [3, 3, 3, 3]
    → 12
```

And indeed:

```
result::type == Int<12>
```

6.6 Using encoded values in C++

So far we have focused on the language and the interaction with other C++ code was limited to extracting simple numeric or boolean values and applying functions. Previous chapters gave the basic idea of data type encoding, but there is still much that needs to be done before such values can be used with usual template code.

```
-- ex7.nri
data List a = Node a (List a) | End;
```

```
result = Node 1 (Node 2 (Node 3 End))
```

The resulting list is consistent with the encoding but hardly usable with C++ library templates. Dummy type is replaced with an underscore for readability.

```
__data<0, _, Int<1>,
    __data<0, _, Int<2>,
        __data<0, _, Int<3>,
            __data<1, _>>>>
```

However, since C++11, numeric lists at type level can be represented by variadic templates. Indeed, this is representation is far more useful for interaction with C++ libraries. At first, we need the variadic template itself:

```
// ex7.cpp
template <int... I>
struct int_list { };
```

And since the encoded list is a recursive data type, recursion is the appropriate tool to use. Adding new elements to the front of the list will be needed for the simplification operation:

```
template <int I, typename T>
struct add_front;


template <int I, int... J>
struct add_front<I, int_list<J...>> {
    typedef int_list<I, J...> type;
};
```

And finally, we can extract all the numeric values from the encoded list and store them inside an `int_list` variadic template:

```
template <typename T>
struct simplify;


template <int I, typename Dummy, typename Rest>
struct simplify<__data<0, Dummy, Int<I>, Rest>> {
    typedef typename add_front<
            I, typename simplify<Rest>::type
        >::type type;
};


template <typename Dummy>
struct simplify<__data<1, Dummy>> {
    typedef int_list<> type;
};
```

And indeed, after simplification we obtain the expected `int_list`:

```
simplify<result::type>::type == int_list<1, 2, 3>
```

However, we can go even further. If a template parameter `P` represents a template parameter pack, `P` can be expanded not only in other types but also in expressions. This can be used to initialize arrays with these compile time constants. Since only functions and classes can be templated, a trick is used to "template" an array. Instead of using template on the array directly, the array is instead placed in a template class.

```
template <typename T>
struct array_init;

template <int... I>
struct array_init<int_list<I...>> {
    static int array[];
};
```

The definition of the member array can be written outside of the class:

```
template <int... I>
int array_init<int_list<I...>>::array[] = {I...};
```

Notice the template parameter pack expansion happens in the initializer list for the array. After that, it is simply a matter of using the array.

```
typedef array_init<simplify<result::type>::type> result_array;

int main() {
    for (int i = 0; i < 3; ++i) {
        std::cout << result_array::array[i] << '\n';
    }
}
```

6.7 Creating functions in C++

The interaction can also go the other way around. It is possible to write template metaprograms representing values and functions and then use those with a code obtained from the compiler. Let us go back to the twice function:

```
-- ex8.nri
twice f x = f (f x)
```

The language cannot represent a certain class of template metaprograms, namely those operating on regular C++ types. However, such functions can be written in the same way the compiled functions are. Type checking is not performed, therefore the programmer has to be careful so as to avoid unwanted behavior (passing functions where an integer is expected, for instance).

```
// ex8.cpp
struct add_ptr {
    struct type {
        template <typename T>
        struct apply {
            typedef typename T::type* type;
        };
    };
};
```

And indeed, applying such function twice (via the compiled `twice`) yields the expected type:

```
twice::type::apply<add_ptr>::type::apply<wrap<int>>::type
    == int**
```

6.8 Creating encoded values in C++

Going from encoded types to variadic templates is not the only useful direction. Compiled functions that expect a list as one of their parameters are very annoying to use directly. We solve this problem by implementing a metaprogram that does the encoding for us.

For instance, consider a function that computes greatest common divisor of a list of numbers. To handle negative numbers, we will need to define what absolute value is:

```
-- ex9.nri
abs : Int -> Int;
abs n = if_ (n < 0) (~n) n;
```

Unary negation uses the prefix operator `~`. Before we compute greatest common divisor (gcd) for a list of numbers, we need to know how to obtain gcd of two numbers. A simple method is Euclid's algorithm. This algorithm assumes non-negative numbers, but the definition can be extended to negative numbers simply by using absolute value.

```
gcd : Int -> Int -> Int;
gcd x y =
    let go a b = if_ (b == 0) a (go b (a % b))
    in  go (abs x) (abs y);
```

Again, this is an example of worker/wrapper transformation. The actual recursion happens in the local function `go`, the outer function `gcd` simply prepares the parameters before calling `go`. Now, for the list part:

```
data List a = Node a (List a) | End;
```

Folding is a general operation on wide class of data types. Folds are higher order functions that describe how to reduce a data structure into a single value. In terms of lists:

```
fold f z [x₁, x₂, ..., xₙ] → f x₁ (f x₂ (... (f xₙ z)))
```

Notice that list fold simply replaces the `Node` constructor with function `f` and the `End` constructor with value `z`. Indeed, this operation can be implemented very easily.

```
foldr : (a -> b -> b) -> b -> List a -> b;
foldr f z = list (\x xs -> f x (foldr f z xs)) z;
```

Few implemented functions already use the above simplification, let us explain it in detail. Notice that `foldr`'s type expresses that three parameters should be provided before the final value is produced. But the definition only ever mentions the first two parameters.

This is the consequence of curried functions. Indeed, it holds that:

$$(\lambda x.Mx)N = MN$$

for every M and N. Therefore the following two definitions are equivalent:

```
foldr f z l = list (\x xs -> f x (foldr f z xs)) z l
foldr f z   = list (\x xs -> f x (foldr f z xs)) z
```

This is known as the η-reduction. It is sometimes added to lambda calculi along with the β-reduction rule.

$$\lambda x.Mx \rightsquigarrow M$$

And finally, $\gcd$ of a list of numbers can be implemented in terms of `foldr` and `gcd`. Much like 0 was used as the neutral element for summation, 0 is also the neutral element for $\gcd$. Indeed, $\gcd(a, 0) = a$ for any a.

```
gcds : List Int -> Int;
gcds = foldr gcd 0
```

On the C++ side, we need a template that converts template parameter pack into an encoded list. Again, this can be easily done with recursion.

```
// ex9.cpp
template <int... I>
struct ints_to_list;


template <>
struct ints_to_list<> {
    typedef __data<1, __dummy> type;
};


template <int I, int... J>
struct ints_to_list<I, J...> {
    typedef __data<0, __dummy, Int<I>,
        typename ints_to_list<J...>::type> type;
};
```

```
gcds::type::apply<ints_to_list<81, 45, 120>>::type == Int<3>
```

And indeed:

```
gcds [81, 45, 120]
    → gcd 81 (gcd 45 (gcd 120 0))
    → gcd 81 (gcd 45 120)
    → gcd 81 15
    → 3
```

6.9 Laziness and infinite data structures

The compilation of C++ templates is surprisingly strict. Recall that lambda calculus has various reduction strategies and some values can be reduced to normal form (i.e., fully reduced) only when using particular strategy. Normal order is one of the strategies that are lazy – if something does not have to be reduced, it will not be. Strict strategies are the opposite – they reduce some expressions that will not be needed. This is not necessarily a bad idea, since strict strategies have other desirable properties – they are easy to implement for normal programming languages and they are also easy to reason about (for instance when analyzing runtime behavior).

Consider the expression `let x = x in x`. This is the simplest expression that has no normal form, `x` is defined to be `x`, which forms a simple infinite loop. However, when compiling the following:

```
result = if_ True 0 (let x = x in x)
```

the infinite loop will be evaluated and trying to compile the resulting metaprogram results in exceeding the maximum template instantiation depth.

But we have seen that other recursive definitions are perfectly fine. This is because of the laziness introduced by lambda abstraction. This is the reason why the above expression fails but the similar:

```
fac n = if_ (n == 0) 1 (n * fac (n -1))
```

works. Using the fixed-point operator with functions is not a problem, using it on values creates problems. For instance, creating an infinite list such as

```
data List a = Node a (List a) | End;

inf = Node 1 inf
```

fails even when only finite part of the list is used (such as extracting few elements from the start). However, we can use the laziness of functions to create infinitely large data types. Instead of storing the tail of a list directly, we can hide it behind a function. For this, it is convenient to have a data type that does not add any extra information – a type with one trivial value.

```
-- ex10.nri
data T = T;
```

Indeed, `T -> a` and `a` hold the same information. We can get from one to the other and vice versa very easily:

```
to : a -> (T -> a);
to a tt = a;

from : (T -> a) -> a;
from f = f T;
```

A stream is simply a list without end. To enable infinite streams, the tail of a stream has to be hidden behind a function. This is where the previously defined type `T` comes in:

```
data Stream a = Node a (T -> Stream a);

head : Stream a -> a;
head = stream \a t -> a;

tail : Stream a -> Stream a;
tail = stream \a t -> t T;

node : a -> Stream a -> Stream a;
node x xs = Node x \tt -> xs;
```

Notice that in order to extract the tail of a stream, the tail "generating" function has to be extracted first and then applied to (the only) value of type `T`. Mapping a function `f` over a stream is simply a matter of applying the function `f` to the head of the stream and then mapping `f` over the tail.

```
map : (a -> b) -> Stream a -> Stream b;
map f = node (f (head s)) (map f (tail s));
```

A stream of all natural numbers can be implemented with in terms of `map`. This slightly odd use of recursion is known as *corecursion* (and types such as `Stream` are also known as *corecursive* types). A stream of natural numbers start with 0 and then a stream of natural numbers, each incremented by one.

```
natural : Stream Int;
natural = node 0 (map (plus 1) natural);
```

To access an element at a given position, we can apply `tail` until we reach the position and then apply `head` to extract the element. Repeated application of a function can be implemented for instance in the following way:

```
repeat : Int -> (a -> a) -> a -> a;
repeat n f x =
    let go n = if_ (n == 0)
                   x
                   (f (go (n - 1)))
    in  go n;
```

And finally, the 8th element:

```
number : Int;
number = head (repeat 7 tail natural)
```

Indeed, we obtain the expected result:

```
number::type == Int<7>
```

6.10 Expressing partial functions

Notice that the only way in which an expression can fail to produce normal value is by non-termination. While the programmer should strive to write programs that work for every input, sometimes it is more convenient to have an easy way of expressing partial functions (i.e., functions not defined for every input). For example, if the programmer can guarantee that a function is always used with non-empty list, then extracting the head of such a list is fine despite being only partial.

Of course, using a placeholder value is not an option for polymorphic functions. Consider the head function hinted at above:

```
head : List a -> a;
head = list (\x xs -> x) ?
```

The missing expression must have a type `∀a. a`. The only expression with such type is:

```
fail : a;
fail = fail
```

As seen above, such head function would not work even on non-empty lists as the `fail` value would halt the compilation. Instead, we need to introduce special value at the C++ level and then import it via the assumption declaration.

```
struct fail { };
```

The best way to implement the fail structure is to use `static_assert`. This allows the use of custom message. It also stands out among other template messages making it much more noticeable. The problem with `static_assert` is that it would stop the compilation no matter whether `fail` is actually used. This can be solved by using `static_assert` inside a template – the idea is that only if the template is instantiated, `static_assert` will be encountered.

```
struct fail {
    template <typename T>
    struct report_error {
        static_assert(false, "fail encountered");
    };

    typedef report_error<void> type;
};
```

This does not work. The problem is C++'s two stage template compilation. In the first stage, C++ compiler will check uninstantiated template (if something does not depend on the template parameter, it can be checked in this stage). The second stage only happens when the template is actually used. We need to make sure the `static_assert` is checked only during the second stage. This can be done by making the assertion depend on the template parameter, for instance:

```
struct fail {
    template <typename T>
    struct report_error {
        static_assert(T::value, "fail encountered");
    };

    typedef report_error<Bool<false>> type;
};
```

Now `fail` can be brought into scope with `assume` declaration:

```
data List a = Node a (List a) | End;


assume fail : a;


head : List a -> a;
head = list (\x xs -> x) fail;


result_ok = head (Node 1 End);
result_ko = head End
```

Using `result_ok` gives the expected:

```
result_ok::type == Int<1>
```

Using `result_ko`, on the other hand:

```
ex11.cpp: In instantiation of
    'struct fail::report_error<Bool<false> >':
ex11.cpp:16:30:   required from here
ex11.cpp:8:9: error: static assertion failed: fail encountered
         static_assert(T::value, "fail encountered");
         ^
```

7 Future work

The compiler can be extended in various ways. The language was based on Haskell. However, Haskell is a much richer language and its features were proven to be useful by developers and users alike. Let us see which features would be feasible to implement.

In particular, one of the most important features is pattern matching (be it the case expression or function clauses). The translation into template specializations is not straightforward as Haskell makes strong guarantees about the order in which the cases are tried – template specializations do not have such guarantees. This is the reason why we went with explicit eliminators in the end – there is always only one possible template specialization, therefore the ordering problems cannot happen. However, case expressions can be expressed in terms of eliminators (the language needs to be extended to allow partiality in ways other than non-terminating expressions, but we have shown how to do this in the example section).

Haskell also does not require ordering of definitions. It is possible to refer to values defined later, which also allows mutual recursion. Such recursion can seem problematic when all recursion is done only via `fix`. However, there are fixed-point combinators that can create mutually recursive functions, consider the following Haskell code:

```
fixmany :: [[a] -> a] -> [a]
fixmany fs = map ($ fixmany fs) fs


[odd, even] = fixmany
    [ \[o, e] n -> if n == 0 then False else e (n - 1)
    , \[o, e] n -> if n == 0 then True  else o (n - 1)
    ]
```

Notice the lambda expressions inside the list get access to both recursive functions that are being defined: the variable `o` represents the `odd` function and `e` the `even` function. Indeed, in the same way simple recursive definitions are translated in terms of `fix`, mutually recursive functions could be translated in terms of `fixmany`.

Type classes were a consideration but were ultimately dropped. This is one of the reasons why the kind system is limited to kinds of the form $* \to * \to \cdots \to *$ (as opposed to, for instance, $(* \to *) \to *$). Without type classes, higher kinded types are not all that useful. The main problem with type classes is that there is no way to translate them into C++ without creating a semantic gap. The use of a type class is implicit in Haskell, but would have to be explicit in C++.

Another possible improvement is the module system. Although importing values is sort of possible with the `assume` declaration, this process could be automated. For example, when compiling a module, the compiler could create an interface file

containing the types of all defined values which could then be used whenever another file imported that module.

Type synonyms could also be easily implemented. Although without complex types (type classes, higher kinded types, etc), their usefulness would go down. The `newtype` definition is also an option, although with (mostly) strict evaluation and much less emphasis on runtime efficiency, `newtype` would be – much like `type` – much less useful.

User defined operators would need more work and their usefulness is limited (as overuse of user defined operators usually makes code much less readable). The main problem is that names of operators are usually not valid names in C++. This could be solved by requiring every operator definition to also contain compiled name. The Haskell fixity declaration could be reused and extended with the naming component:

```
infixl 7 * as times;
infixl 6 + as plus;
```

Another very useful part of Haskell are where clauses. Translation of those is very simple (there is 1:1 correspondence with let expression) and they make the code more readable.

Haskell also allows for omission of braces and semicolons via something called layout. Indeed, indentation is important part of the syntax, allowing us to distinguish between two separate expressions and one expression that continues over multiple lines simply by observing how are these expressions aligned. *Haskell report*[20] defines a procedure which converts layout back to explicit braces and semicolons. Similar process could be employed.

Another direction is extending the type system. Since most of the more powerful type systems build on the same foundation (the semantics of untyped lambda calculus), the type system could easily be extended without changing the underlying translation. For instance, explicit universal quantification could be added to go back from HM to System F. Sometimes, only a fraction of System F is needed. GHC extension `ScopedTypeVariables` lets the programmer explicitly quantify type variables with `forall`. Such variables can then be used in local definitions (`let` or `where`) without referring to a fresh type variable (which is the current behavior).

GADTs[21] (generalized algebraic data types) do not require new translation and yet add quite a bit of power to the type system. Weaker version are the existential types, though these are not that useful without the presence of type classes.

We could also go further and implement a type system with dependent types (such as Martin-Löf's type theory[22]). Again, the underlying language is the same. After types are erased, the original translation can be used.

The main problem we would have to face is that to keep type checking decidable, these type systems often need typing hints. When not needed, the values are figured

out for instance by unification and then implicitly used. This again creates a semantic gap between the language and the compiled code and having everything be explicit creates much unnecessary burden on the programmer.

8 Related work

The idea of using tools to ease up template metaprogramming is not new. Most tools can be divided into two main approaches: C++ libraries and external tools.

C++ libraries that facilitate metaprogramming include examples such as *Boost.MPL*[23] or *Boost.Metaparse*[24]. The aim of these libraries is to provide simpler way of writing metaprograms while still saying within the limits of C++. However, they still have to deal with the cumbersome syntax of template metaprogramming and thus retain some of the problems related to maintainability.

The other approach (such as ours) is the use of external tools. The main advantage is that these tools do not need to rely on the complexities of template metaprogramming. *MetaFun*[25] is an example of a tool that transforms simple language into a C++ metaprogram. The translation is even simpler than ours, making heavy use of template-template parameters (template parameters that are templates themselves). However, as we have seen previously, template-template parameters are at odds with higher order functions. As far as we know, *MetaFun* does not support currying and lambda abstraction.

Another example is the language *EClean*[26]. The idea is to translate the input language just enough to be able to interpret it at the level of C++. The compiler therefore produces C++ code that cannot be directly compiled. Instead, it is given to a metaprogram which interprets it. This interpreter is basically a graph rewriting engine, evaluating expressions in a way similar to how compiled Haskell programs are executed. Another similar approach is suggested in *Functional Programming with C++ Template Metaprograms*[27].

Conclusion

In this work, we have built a compiler capable of translating simple functional language into C++ template metaprograms. The compilation process can handle basics of lambda calculus as well as several extensions (let expressions, build-in data types, user-defined data types and recursion). Runtime needed to correctly use metaprograms is also implemented.

The language itself is loosely based on Haskell, the key differences being the lack of layout and several of the more advanced features (type classes, higher kinds, etc). Much like Haskell, the compiler is capable of inferring and checking types without explicit annotations (thanks to Hindley-Milner type system).

It is interesting to note that despite being one of the main examples of procedural languages, C++ had a purely functional fragment long before functional features (such as higher-order functions or lambda expressions) became commonplace in mainstream programming languages.

This purely functional foundation of templates allowed us to create simple functional language which can describe great deal of template metaprograms.

The language itself is a compromise – more powerful language features (such as type classes) are more convenient for the programmer but often carry around nontrivial requirements on the compiled code side. We opted for simpler language which allowed us to have straightforward translation without hidden surprises or corner cases. This makes the interaction with existing metaprograms much easier.

Simplicity was also the main goal of implementation – the language can be easily extended as can be the translation.

This work is also interesting experiment to see to what extremes can template metaprogramming be taken. In fact, if one is able to use C++ macro machinery to convert string literals into character lists (encoded using either variadic templates or the encoding mentioned earlier), it wouldn't be very hard to use this language to implement this very compiler as a metaprogram.

References

[1] JOHN MCCARTHY, *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*, Massachusetts Institute of Technology, Cambridge, Mass., 1960

[2] TIM SHEARD, SIMON PEYTON JONES, *Template metaprogramming for Haskell*, Proc. Haskell Workshop 2002, Pittsburgh, pp. 1-16

[3] TODD L. VELDHUIZEN, *C++ Templates are Turing Complete*, 2003

[4] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, *ISO/IEC 14882:2014*

[5] FELICE CARDONE, J. ROGER HINDLEY, *History of Lambda-calculus and Combinatory Logic*, Swansea University Mathematics Department Research Report No. MRRS-05-06, 2006

[6] S. C. KLEENE, J. B. ROSSER, *The inconsistency of certain formal logics*, Annals of Mathematics 36 (3) pp. 630-636, 1935

[7] A. CHURCH, *A Formulation of the Simple Theory of Types*, The Journal of Symbolic Logic, Vol. 5, No. 2., pp. 56-68., 1940

[8] JEAN-YVES GIRARD, *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*, Ph.D. thesis, Université Paris 7, 1972

[9] JOHN C. REYNOLDS, *Towards a Theory of Type Structure*, Syracuse University, 1974

[10] J. B. WELLS, *Typability and type checking in System F are equivalent and undecidable*, Annals of Pure and Applied Logic, Volume 98, Issues 1-3, pp. 111-156, 1999

[11] J. ROGER HINDLEY, *The Principal Type-Scheme of an Object in Combinatory Logic*, Transactions of the American Mathematical Society 164, pp. 29-60, 1969

[12] ROBIN MILNER, *A Theory of Type Polymorphism in Programming*, Journal of Computer and System Science 17, pp. 348-374, 1978

[13] LUIS DAMAS, ROBIN MILNER, *Principal type-schemes for functional programs*, 9th Symposium on Principles of programming languages, pp. 207-212, 1982

[14] DAAN LEIJEN, PAOLO MARTINI, *parsec library*, github.com/aslatter/parsec

[15] MARK. P JONES, *Typing Haskell in Haskell*, Proceedings of the 1999 Haskell Workshop, Paris, France

[16] PAOLO CAPRIOTTI, *optparse-applicative library*, github.com/pcapriotti/optparse-applicative

[17] THE GLASGOW HASKELL TEAM, *Glasgow Haskell Compiler*, haskell.org/ghc/

[18] ISAAC JONES, DUNCAN COUTTS, *Cabal*, haskell.org/cabal/

[19] HASKELL PLATFORM DEVELOPERS, *Haskell Platform*, haskell.org/platform/

[20] SIMON MARLOW (EDITOR) ET AL., *Haskell 2010 Language Report*, 2010

[21] SIMON P. JONES, DIMITRIOS VYTINIOTIS, STEPHANIE WEIRICH, GOEFFREY WASHBURN, *Simple unification-based type inference for GADTs*, ICFP 2006, pp. 50-61

[22] PER MARTIN-LÖF, *Intuitionistic Type Theory*, ISBN 88-7088-105-9, 1984

[23] ALEKSEY GURTOVOY, DAVID ABRAHAMS, *Boost MPL library*, boost.org/doc/libs/1_61_0/libs/mpl/doc/index.html

[24] ABEL SINKOVICS, *Boost Metaparse library*, boost.org/doc/libs/1_61_0/doc/html/metaparse.html

[25] GERGŐ ÉRDI, *MetaFun*, gergo.erdi.hu/projects/metafun/

[26] ÁDÁM SIPOS, VIKTÓRIA ZSÓK, *EClean – An Embedded Functional Language*, Department of Programming Languages and Compilers, Eötvös Loránd University, Budapest, Hungary

[27] ZOLTÁN PORKOLÁB, *Functional Programming with C++ Template Metaprograms*, Department of Programming Languages and Compilers, Eötvös Loránd University, Budapest, Hungary

Appendix

A. Grammar

$$
\begin{aligned}
Module &::= TopLevel\ ;\ Module \\
&\quad \mid \lambda
\end{aligned}
$$

$$
\begin{aligned}
TopLevel &::= DataDefinition \\
&\quad \mid Definition \\
&\quad \mid TypeSignature \\
&\quad \mid Assumption
\end{aligned}
$$

$$
\begin{aligned}
DataDefinition &::= \text{data}\ TyCon = Variants \\
&\quad \mid \text{data}\ TyCon
\end{aligned}
$$

$$
TyCon ::= UIdentifier\ TyVars
$$

$$
\begin{aligned}
TyVars &::= LIdentifier\ TyVars \\
&\quad \mid \lambda
\end{aligned}
$$

$$
\begin{aligned}
Variants &::= Variant \mid Variants \\
&\quad \mid Variant
\end{aligned}
$$

$$
Variant ::= UIdentifier\ DataConArgs
$$

$$
\begin{aligned}
DataConArgs &::= DataConArg\ DataConArgs \\
&\quad \mid \lambda
\end{aligned}
$$

$$
\begin{aligned}
DataConArg &::= LIdentifier \\
&\quad \mid UIdentifier \\
&\quad \mid (\ Type\ )
\end{aligned}
$$

$$
Assumption ::= \text{assume}\ AnyIdentifier : Type
$$

$$
TypeSignature ::= LIdentifier : Type
$$

$$
Definition ::= LIdentifier\ Args = Expression
$$

$$
\begin{aligned}
Args &::= LIdentifier\ Args \\
&\quad \mid \lambda
\end{aligned}
$$

$$
\begin{aligned}
Type ::= & \; TypeFactors \texttt{ -> } Type \\
& \mid TypeFactors
\end{aligned}
$$

$$
\begin{aligned}
TypeFactors ::= & \; TypeFactors \; TypeFactor \\
& \mid TypeFactor
\end{aligned}
$$

$$
\begin{aligned}
TypeFactor ::= & \; UIndentifier \\
& \mid LIdentifier \\
& \mid \texttt{(} \; Type \; \texttt{)}
\end{aligned}
$$

$$
Expression ::= OrExpression
$$

$$
\begin{aligned}
OrExpression ::= & \; XorExpression \; || \; OrExpression \\
& \mid XorExpression
\end{aligned}
$$

$$
\begin{aligned}
XorExpression ::= & \; AndExpression \; \hat{} \; XorExpression \\
& \mid AndExpression
\end{aligned}
$$

$$
\begin{aligned}
AndExpression ::= & \; CompExpression \; \&\& \; AndExpression \\
& \mid CompExpression
\end{aligned}
$$

$$
\begin{aligned}
CompExpression ::= & \; AddExpression < AddExpression \\
& \mid AddExpression <= AddExpression \\
& \mid AddExpression > AddExpression \\
& \mid AddExpression >= AddExpression \\
& \mid AddExpression == AddExpression \\
& \mid AddExpression /= AddExpression \\
& \mid AddExpression
\end{aligned}
$$

$$
\begin{aligned}
AddExpression ::= & \; AddExpression \; + \; MultExpression \\
& \mid AddExpression \; - \; MultExpression \\
& \mid MultExpression
\end{aligned}
$$

$$
\begin{aligned}
MultExpression ::= & \; MultExpression \; * \; PrefixExpression \\
& \mid MultExpression \; / \; PrefixExpression \\
& \mid MultExpression \; \% \; PrefixExpression \\
& \mid PrefixExpression
\end{aligned}
$$

$$
\begin{aligned}
PrefixExpression ::= & \; \sim PrefixExpression \\
& \mid \; ! \; PrefixExpression \\
& \mid TypeSigExpression
\end{aligned}
$$

$$
\begin{aligned}
TypeSigExpression ::= & \; ExpressionFactors \; : \; Type \\
& \mid ExpressionFactors
\end{aligned}
$$

$$
\begin{aligned}
ExpressionFactors &::= ExpressionFactors\ ExpressionFactor \\
&\mid ExpressionFactor
\end{aligned}
$$

$$
\begin{aligned}
ExpressionFactor &::= AnyIdentifier \\
&\mid NumericLiteral \\
&\mid BooleanLiteral \\
&\mid Lambda \\
&\mid LetIn \\
&\mid (\ Expression\ )
\end{aligned}
$$

$$
Lambda ::= \backslash\ LIdentifier\ Args \texttt{ -> } Expression
$$

$$
LetIn ::= \text{let}\ Definitions\ \text{in}\ Expression
$$

$$
\begin{aligned}
Definitions &::= Definition\ ;\ Definitions \\
&\mid Definition
\end{aligned}
$$

$$
NumericLiteral\ ::=\ 0 \mid 1 \mid 2 \mid \ldots
$$

$$
BooleanLiteral\ ::=\ \text{True} \mid \text{False}
$$

$$
\begin{aligned}
AnyIdentifier &::= LIdentifier \\
&\mid UIdentifier
\end{aligned}
$$

It remains to mention that $LIdentifier$ and $UIdentifier$ represent identifiers starting with lower-case and upper-case letters, respectively.

B. Implementation

The implementation of the runtime mentioned in Chapter 4 can be found in the directory `runtime`, modules described in Chapter 5 in `src` and finally the examples from Chapter 6 in `examples/thesis`.

The source code is also available online at github.com/vituscze/norri.