Report Mini Project E

Compiler Construction

Jordy van Dortmont, Fabian Thorand

December 2015

1 Introduction

For our solution to the assignment we created some substantial additions to the base language we were given which are covered in the next section. The third section gives a brief introduction to our approach to laziness and the fourth section briefly describes the conversion to normal form. Finally, the last section covers the use of attribute grammars in our implementation and explains in greater detail how the aforementioned aspects are actually realized.

2 Language Design

In our language the already defined functionality is still present as is. We extended the language with the primitive functions, which you can call with **prim** followed by the string literal indicating the primitive function and the required arguments. There is no restriction on what an argument can be, since more complex expressions are handled by the transformation to ANF. An example is shown in the following code snippet.

```
let n = prim "primAddInt" 1 2 in
    prim "primAddInt" (prim "primAddInt" 3 4) n
ni
```

Primitive calls always have to be given the exact number of arguments and may neither be over nor under saturated. To allow currying of primitive operations, one has to define auxilliary bindings in terms of lambda expressions.

We also added user definable types with a Haskell-like syntax. There can be arbitrarily many data declarations in the source code, but they must be placed in the beginning before any other expression. The two required data types for the assignment are shown below.

```
data Bool = False | True
data List = Nil | Cons(2)
```

In contrast to Haskell, we work here with an untyped language. For this reason, we specify the number of arguments for constructors instead of giving the types of the

arguments. The specification of the number of arguments is optional, if there is none then zero is assumed. The constructors are assigned tags from left to right, starting at zero. Hence, in the above example **False** and Nil have tag zero and **True** and Cons have tag one.

To this end, we also introduced a new root data type for the HM language called `Mod` (previously, an HM program was just represented as `Tm`). It encompasses both the list of data declarations as well as the actual HM term.

These data declarations are also used for generating the meta data for interoperability with primitve operations. The above declarations yield the following CoreRun meta data. In particular, this means that for using the *primEqInt* primitive (retuning a boolean value), the source code file must contain the corresponding data declaration for booleans (keeping the order of the constructors in mind).

```
data "Bool" = "False" -> 0, "True" -> 1;
data "List" = "Nil" -> 0, "Cons" -> 1;
```

The following list contains some examples using the data types shown above.

list of integers: Cons 1 (Cons 2 Nil)

list of lists of integers: Cons (Cons 1 Nil) (Cons Nil Nil)

infinite list of functions: **let** ids = Cons (\x . x) ids **in** ... **ni**

To handle conditional branching we added if-statements of the following form:

if cond **then** tmTrue **else** tmFalse **fi**

An if-statement inspects the tag of the condition (cond) and executes the then term if the tag is one and the else part if the tag is zero. In a typed language, the only valid type for cond would be —Bool—, but since we are in an untyped setting any type with at least two constructors will do it here.

To inspect the datatypes and apply functions to them we need to specify some kind of pattern matching to match on the constructors and tell us what kind of value we are dealing with. This is why we introduced a case statement. We can use a case statement like this:

```
data List = Nil | Cons(2)
let append = \xs . \ys .
  case xs of
    Nil = ys;
    Cons h t = Cons h (append t ys)
  esac
in append (Cons 1 Nil) (Cons 2 Nil) ni
```

The case statement pattern matches on the scrutinee (evaluating it in the process) and checks the alternatives accordingly. In the above case we see an example of appending two lists. Each constructor of the type of the value we are matching on has to correspond to *exactly* one case alternative, otherwise the translation to CoreRun will produce invalid code. On the other hand, the order of the alternatives is not important.

Using case expressions, we can define functions like isNil ourselves as following. For that reason, they are not built into the language, but instead their implementation is left to the user.

```
data Bool = False | True
data List = Nil | Cons(2)
let isNil = \xs .
  case xs of
    Nil = True;
    Cons h t = False
  esac
in if isNil (Cons 1 Nil) then 2 else 3 fi ni
```

3 Laziness

In our implementation of the language, we have the following convention concerning lazy values:

1. A lambda expression always expects a lazy value (i.e. thunk) as an argument.
2. A lambda expression always returns a lazy value.
3. Local let bindings as well as global bindings are lazy.
4. Constructor arguments are lazy.
5. Constructor calls are lazy.

Those "rules" allow a uniform handling of terms throughout the translation to low level code.

Laziness can for example be used to work with infinite lists, as demonstrated by the following example.

```
data Bool = False | True
data List = Nil | Cons(2)

let eq   = \ x . \ y . prim "primEqInt" x y in
let add  = \ x . \ y . prim "primAddInt" x y in
let sub  = \ x . \ y . prim "primSubInt" x y in
let foldl = \ f .
  let go = \ x . \ xs .
    case xs of
      Nil = x;
      Cons h t = go (f x h) t
    esac
  in go ni in
let take =
  \n . \xs .
    case eq n 0 of
```

```
      True = Nil;
      False =
        case xs of
          Nil  = Nil;
          Cons h t = Cons h (take (sub n 1) t)
        esac
    esac
  in
let repeat = \x . let xs = Cons x xs in xs ni in
let sum = foldl add 0 in
let ones  = repeat 1 in
sum (take 5 ones)
ni ni ni ni ni ni ni ni
```

4 Recursion

As evident above, our language also supports recursion, by providing the identifier bound in a let expression not only to the let body, but also to the bound expression.

Mutual recursion can be done by explicitly "binding" the participating functions after their definitions, because variable declarations are only propagated downwards.

The following example demonstrates the concept. We would like to define two mutually recursive functions `f` and `g`. We first define a variant `f_explicit` of `f` which takes the functions it depends on, but which are declared later, as arguments (in this case only `g`). Since `g` is defined after `f_explicit` it can directly call it, passing itself as an argument. If `f` is meant to be used at a later point, an additional binding can be introduced which passes `g` as first argument to `f_explicit`.

```
let  f_explicit  = \g . \x . g x in
let g = \x .  f_explicit  g x in
let f =  f_explicit  g in
 ...
ni ni ni
```

Note that it is intented for the sake of demonstration that `f` and `g` will never terminate, but one can easily imagine a more sensible recursion with a termination condition.

5 Attribute Grammars

5.1 Pos.ag

This is a helper attribute grammar which defines a synthesized attribute *pos* for `Tm` values to provide the source position of a term to its ancestors, and an inherited attribute of the same name for all descendants of `Tm` values.

The source position is used during the translation to ANormal form and for producing better error messages.

5.2 ToANF.ag

This attribute grammar is used for converting an arbitrary HM program to ANormal form. It thus introduces additional let bindings whenever a subexpression is required to be a simple expression. This is the case for

- primitive call arguments,
- conditions in *if* expressions,
- scrutinees in *case* expressions and
- function arguments.

Going one step further, we also disallow natural numbers in these positions because of laziness. A natural number in the source language will eventually be translated to a thunk evaluating to a natural number. Therefore, it would not be a simple expression anymore. The only valid expression in argument positions are variables.

In the the process we also wrap the main expression of a program in an additional binding. This results in a CoreRun main expression which simply refers to that value. The reason for this simply is that we were advised by Atze to refrain from using let bindings in the main expression.

We use the following attributes in this grammar:

`anf :: Mod`, *synthesized*	returns the final result of the translation to ANormal form of an HM module (not to be confused with the CoreRun data type of the same name).
`counter :: Int`, *chained*	provides unique numbers to all term parts of the AST which are used for introducing fresh variables. It uses the *uniqueref* mechanism provided by attribute grammars.
`bindings :: Bindings`, *synthesized*	is a list of let bindings that need to be reintroduced at the next higher scope. When translating to ANormal form, we have to deal with possibly nested let bindings. In the CoreRun language, let expressions do not return (i.e. are always in a tail call position), and thus, they always have to occur in the beginning of a new scope, such as at the root of a lambda expression or of case alternatives. Whenever we need to introduce an auxilliary binding, it is added to the `bindings` attribute. Lambda, case and if expressions as well as the module root and let-bound values then reintroduce these bindings.
`noLetTm :: Tm`, *synthesized*	is a term that does not contain let expressions, and it gets passed alongside the aforementioned `bindings` attribute. All variables that need to be in scope for this term are conveyed separately through the list of bindings.

`noLetTmL :: TmL`, *synthesized* — is gathers the values of the `noLetTm` attribute when handling a list of terms, as it is the case in the arguments of a primitive call.

`anfAltL :: AltL`, *synthesized* — returns the result of converting the alternatives of a case expression to ANormal form.

5.3 HMToCR.ag

This attribute grammar performs the translation from the HM source language to the CoreRun language. It is, in some sense, split in two phases. First, data declarations are handled by introducing the necessary global bindings, and then, the actual program is translated.

We use the following attributes for the translation of data declarations:

`names :: Set Name`, *chained* — is simply a set of all data type names encountered so far while traversing the data declarations. It is used to check for duplicate names (yielding an error).

`metal :: CR.MetaL`, *synthesized* — returns the list of meta information for data types generated from the data declarations.

`conIndex :: Int`, *inherited* — is used locally in the list of data constructors of a declaration to provide the index of a data constructor, which is also as its tag.

`metaConl :: CR.MetaDataConL`, *synthesized* — is the list of meta information about the data constructors in a declaration. It is used in constructing the `metal` values in the ancestor.

`globalOffset :: Int`, *chained* — is used to determine the next free slot for global bindings. We need this to provide global bindings for data constructors.

`conTags :: TagMap`, *synthesized* — returns a map from constructor names to its corresponding tag. This information is needed to correctly reorder alternatives when translating case expressions.

`globalBinds :: GlobalBinds`, *synthesized* — returns a map of the global data constructor bindings.

`env :: Environment`, *chained* — returns a map from identifiers to a CoreRun reference to the associated value. For each data constructor, we add the corresponding module level reference to the binding introduced in `globalBinds`. This value is chained instead of synthesized in order to check for duplicate constructor names.

Given the information acquired from the data declarations we translate the actual HM term using the following attributes:

`exp :: CR.Exp`, *synthesized*	returns the final translation result of an HM term to a CoreRun expression.
`env :: Environment`, *inherited*	has the same purpose as during the translation of data types, but now it is just inherited instead of chained, because scoping is now hierarchical, and we only need to pass bound variables to descendants.
`level :: Int`, *inherited*	propagates the lexical level of the current expression. It is increased when descending into a lambda or thunk expression. We need it to create local references across scope boundaries.
`globalScope :: Bool`, *inherited*	is a flag that is initially set to `True`, but set to false as soon as we enter any nested scopes like case or if expressions, let-bound values or lambdas. As long as this flag is true, let-bound values are translated to global bindings instead of let expressions.
`globalOffset :: Int`, *inherited*	has the same use as during the translation of data values, but doesn't have to be passed upwards anymore.
`globalBinds :: GlobalBinds`, *synthesized*	returns the let-bound values that have been promoted to module-level bindings instead.
`offset :: Int`, *inherited*	is the offset on the current stack where new bindings are placed. It is increased by let bindings and reset when entering a scope with a new stack (i.e. lambda expressions and thunks). Note that all alternatives of a case expression share the same stack offset, since only one alternative will be executed.
`laziness :: Laziness`, *inherited*	tells a child node whether its ancestor requires a `Strict` or `Lazy` value. For example, when introducing a thunk, the expression wrapped inside the thunk must then evaluate to a strict value. When an expression does not meet the requirement of the ancestor, it will insert a thunk or an evaluation expression as necessary.
`conTags :: TagMap`, *inherited*	is the mapping of constructors to their tags produced by the translation of data declarations.
`expL :: [CR.Exp]`, *synthesized*	is used for translating a list of HM terms to a list of CoreRun expressions. This is required for the arguments of a primitive call.
`altMap :: Map.Map Int CR.Exp`, *synthesized*	is produced by the list of alternatives of a case expression. It maps the constructor tag of a pattern to the corresponding expression executed in that alternative. The translation for case expressions then takes these expressions ordered by the tag.

`scrutRef :: DeclRef`, *inherited*	tells the alternative of a case expression the reference to the scrutinee, in order to bind the pattern variables to the corresponding constructor field.

In the environment mapping identifiers to references, we use our own data type `DeclRef` instead of `CR.Ref`. The only difference is that our data type represents a local reference using an absolute level instead of a level difference. Only when such a variable is used, the difference between the level of the declaration and the current level (witnessed by the `level` attribute) is computed.

5.4 ToCoreRun.ag

We modified the `ToCoreRun` grammar provided with the assignment to actually calculate the right stack size for lambda expressions and thunks, instead of using the default value of 100, in order to ensure correctness of the generated programs. In most cases such a large stack size wastes a lot of space, which is bad, but doesn't affect correctness. But there might be programs, that require a larger stack, and then this becomes incorrect.

We also added data types to the corresponding `Base.ag` to model the meta information needed for marshalling data types when interacting with primitive operations. This meta information now also needs to be converted using the actual CoreRun API.

To achieve these goals, we use the following attributes:

`crmetal :: [CR.Meta]`, *synthesized*	generates a list of meta information consumed by the `mkModWithMetas` smart constructor in the CoreRun API.
`crdataconl :: [CR.DataCon]`, *synthesized*	generates the list of data constructors needed for the `CR.Meta` values.
`stksize :: Int`, *synthesized*	calculates the actual stack size that is required for executing a given expression. When encountering bindings, the stack size is added, when handling case expressions, the resulting stack size is the maximum of the alternative's stack sizes. The latter is correct as only one alternative is executed, and therefore all the others don't need stack space. When encountering a lambda expression, we reset the stack size in the outer scope to one (since the lambda only requires on stack slot), but we set the required stack size of the lambda to the value computed for the body.