Compiler Construction

Mini Project

December 26, 2014

E Code generation for a lazy functional language

The aim of this mini project is generate runnable code from a small functional language, thereby dealing with the issues arising particular to laziness. Issues originating from partial parameterization and higher-order function implementation are left out. Code generation targets a virtual machine part of UHC (`https://github.com/UU-ComputerScience/uhc`) and its smaller hackage pacakge extract `uhc-light` (`https://hackage.haskell.org/package/uhc-light`) called CoreRun. Operational semantics is provided in a separate document (`http://foswiki.cs.uu.nl/foswiki/pub/Ehc/Documentation/corerun.pdf`).

The project distribution is the same as the distribution available for the lab on "Type Reconstruction" extended [1] abstract-syntax types for CoreRun and a pretty-printer `pp-core` generating CoreRun input runnable by `uhcr` from the `uhc-light` package. [2] Our objective is to implement a code generator for terms either in the implicitly typed language or the explicitly typed System-F variant as used by the lab on "Type Reconstruction". The choice between these two is free.

What to Do

Before you proceed, read the CoreRun semantics and its API. The following assumes you have read this.

1. Implement a program `hm2cr` if your starting point is the `ht` langauge or `systemf2cr` if your starting point is System F. This program generates CoreRun code, for example the following can be compiled:

   let $fv = 5$ **in**
   let $id = \lambda x.x$
   in $id\ fv$
   ni ni

   This should result in CoreRun similar to:

   ```
   module Main 0,6 -> eval(g.0.2);
   \1,2 -> d.0.0;
   5;
   \0,2 -> app (g.0.0)(g.0.1);
   ```

[1] Actually a replacement

[2] As of the first (2014) edition of this project this part of the distribution is not yet tested, it comes as "only compilable" code

The example CoreRun already takes into account stack size information; this and other details are not yet supported by the wrapper syntax for CoreRun and if necessary should be changed according to the tasks in this project .

Initially ignore the following issues, i.e. just make a minimal implementation work:

- CoreRun limits arguments to refer to identifiers (or integer constants) only; CoreRun is in so-called ANormal form. However, the source program needs not be in ANormal form. Limit source programs to ANormal form too so a direct translation to equivalent constructs can be easily implemented. The following fragment is not in ANormal form, hence not (yet) need to be accepted.

$$\begin{array}{l}\textbf{let } fv = 5 \textbf{ in}\\ \textbf{let } id = \lambda x.x\\ \textbf{in } id\ (id\ fv)\\ \textbf{ni ni}\end{array}$$

  However, it is necessary to take care of the following:

  - Mapping identifiers to references in CoreRun, which consist of offsets. Nesting and shadowing of identifiers need to be taken care of too.

2. Introduce an intermediate transformation to convert source programs into ANormal form. The example above including $id\ (id\ fv)$ would now be transformed to

$$\begin{array}{l}\textbf{let } fv = 5 \textbf{ in}\\ \textbf{let } id = \lambda x.x \textbf{ in}\\ \textbf{let } idfv = id\ fv\\ \textbf{in } id\ idfv\\ \textbf{ni ni ni}\end{array}$$

   A transformation like this can be done on the source program AST but also on the wrapper AST for CoreRun. In the latter case the structure of the CoreRun wrapper AST would require modifications to cater for both ANormal and non-ANormal form programs. The choice is free.

3. Extend the source language by a construct which allows to call primitives. For example, the following declares a function for invoking a runtime primitive (which should (and is) provided by the programming running CoreRun):

$$\begin{array}{l}\textbf{let } plus = \lambda x.\lambda y.\textbf{prim } \texttt{"primAddInt"}\ x\ y \textbf{ in}\\ \textbf{let } fv = 5 \textbf{ in}\\ \textbf{let } id = \lambda x.x\\ \textbf{in } id\ (plus\ (id\ fv)\ 4)\\ \textbf{ni ni ni}\end{array}$$

   Running the resulting program should yield 9. You can assume the existence of the following primitives but no more: *primAddInt*, *primSubInt*, *primEqInt*.

4. Uptil now all evaluation is strict, that is all arguments can safely (i.e. no non-termination occurs) be evaluated before being passed to a function. Laziness is essential for infinite structures of which only part is used.

```
let plus  = λx.λy.prim "primAddInt" x y in
let minus = λx.λy.prim "primSubInt" x y in
let eq    = λx.λy.prim "primEqInt" x y in
let l     = cons 3 l in
let sum   = λl.if isnil l then 0 else head l `plus` sum (tail l) fi in
let take  = λn.λl.if eq n 0
                  then nil
                  else  if isnil l
                        then nil
                        else  cons (head l) (take (minus n 1) (tail l)) fi fi
in sum (take 3 l)
ni ni ni ni ni ni ni
```

   In order to make this work the following needs to be implemented:

   - **Boolean and list types.** Various solutions exist:
     - In a full-blown language implementation like for *Haskell* this would be done by user definable data types.
     - Add special builtin syntax to only deal with booleans and lists.
     - Add special builtin syntax to deal with low level details (similar to those for primitives) allowing to allocate memory nodes (as done by *cons* and *nil*) and inspect memory node tags (as done by *isnil*).
     - Avoid special syntax and define the required functionality in the compiler in such a way that it always is included in the resulting program.

     You are free to choose between these alternatives (and others you can come up with), but in the end it is required that a function like *cons* will allocate a memory node with (say) tag 0 and two fields. Similarly, *nil* allocates a memory node without fields and tag 1. A boolean is represented by a memory node without fields, with tag 0 for a value for *False*.
   - **If expression.** A CoreRun case expression can be used for this.
   - **Recursion.** CoreRun supports (mutual) recursion but it needs to be ensured that variable references in the source program resolve correctly.
   - **Laziness.** Uptil now we have ignored the fact that the infinite list *l* will make the program crash unless it is represented lazily. CoreRun allows lazy values by representing those as parameterless functions. When such a value is to be inspected (e.g. in the condition of an **if**-expression) it must be ensured that the lazy value is forced to evaluate, but only then. Additionally, primitives now can expect

lazy values to be passed as arguments; these need to be evaluated as well before passed to the runtime system.

The distribution for this project contains a CoreRun example implementing the above source program.

Submitting

The source code of your implementation should be handed in according to the submission instructions on the website of this course.

Submit the source code of your implementation (including both UUAG sources and Haskell sources *not* generated by the UUAG system).

Include in your submission a number of example programs that illustrate the capabilities of your implementation.