Running Haskell on the CLR

“but does it run on Windows?”

Jeroen Leeuwestein, Tom Lokhorst
jleeuwes@cs.uu.nl, tom@lokhorst.eu

January 29, 2009
Don't get your hopes up!

module Main where

foreign import ccall "primAddInt" (+) :: Int → Int → Int

inc :: Int → Int
inc x = x + 1

data List = Nil | Cons Int List

length :: List → Int
length Nil = 0
length (Cons x xs) = inc (length xs)

five :: List
five = Cons 1 (Cons 2 (Cons 3 (Cons 4 (Cons 5 Nil))))

main = length five
Why target the CLR?

A lot of presence.

- **Multiple versions** of Windows desktops.
- **OS X** and **Linux** desktops, through Mono.
- Web browsers, through **Silverlight** and **Moonlight**.
- Mobile devices:
  - Windows **Mobile**.
  - Mono on the **iPhone** and **Android**.
- In the cloud!
  - Windows **Azure**: Distributed computation environment.
Why target the CLR?

Rich environment.

- Interop with other languages.
- Access a huge set of libraries.
- Provide libraries developed in Haskell.
What is the CLR?

Common Language Runtime / Mono Project

- Stack-based virtual machine.
- First-class support for classes with methods.
- Basic operations for reference types and value types.
- Type safe: operations must match the exact type.
- Dynamic casting is allowed.
- Executes Common Intermediate Language (CIL).
- CIL has a concrete syntax.
  - ilasm
  - ildasm / monodis
What is the CLR?

```csharp
.class private Test extends [mscorlib] System.Object
{
    .method private static void Main () cil managed
    {
        .entrypoint
        .locals init (int32 x)
        ldc_i4 2
        stloc 0
        ldc_i4 3
        ldloc 0
        add
        call void class [mscorlib] System.Console :: WriteLine (int32)
        ret
    }
}
```
Architecture of .NET backend

$ bin/8/ehc -ccil Test.hs
$ ls
Test.hs Test.il
$ ilasm Test.il
$ ls
Test.exe Test.hs Test.il
$ mono Test.exe
42
Architecture of .NET backend
Architecture of .NET backend

Haskell package language-cil.

Abstract syntax for the Common Intermediate Language.

With build functions and pretty printer for concrete syntax.

Future:

- Support all CIL constructs
- Parser for concrete syntax
- Analysis functions
- Release on Hackage
Philosophy on the Runtime System

How to treat the RTS?

- As an abstract machine?
  - simulate virtual memory
  - simulate registers
  - simulate functions and function calls

- Use it for what it was designed
  - build strongly typed objects
  - use inheritance
  - use method calling conventions
  - interop with other languages

Look at what other languages do (F#).
Philosophy on the Runtime System

Some questions

```
data List = Nil | Cons Int List
```

What is the **type** of List?

What are the **types** of Nil and Cons?

How do we handle do **thunks** and **partial applications**?

And what about **updates**?
Philosophy on the Runtime System

Some questions

\textbf{data} \quad List = Nil \mid Cons \text{ Int List}
Some questions

```
data List = Nil | Cons Int List

Cons 1 (xs `append` ys)
```
Philosophy on the Runtime System

Some questions

data List = Nil | Cons Int List
xs = [1, 2]
Code generation

- Generate code from GRIN
- Direct translation of GRIN constructs
Evaluate $expr$ and bind the result to $x$.

$expr; \lambda x \rightarrow \ldots \text{length } x \ldots$

expr
STLOC $x$
...
LDLOC $x$
CALL length(object)
...
Code generation

Case

Match a tag variable against different alternatives.

case tag of
    CNil → ...
    CCons → ...

    tag
L1:
    DUP
    ISINST CNil
    BRFALSE L2
    POP
    ...
L2:
Store a value on the heap and return a pointer to it.

\[ \text{store val} \]

val
NEWOBJ RefObj::ctor(object)
All our values are already stored on the heap, so we only have to create a pointer.
Code generation

Update

Update the value pointed to by pointer $x$ with $val$.

update $x$ $val$

LDLOC $x$
val
STFLD RefObj::Value
Fetch the **tag** of a node, following **pointer** \( x \).

\[
\text{fetch } x [0]
\]

LDLOC \( x \)
LDFLD RefObj::Value

We have no representation for **stand-alone tags**. We use the **complete node**.
Fetch the first field of a node, following pointer $x$.

$fetch \ x\ [1]$

LDLOC $x$
LDFLD RefObj::Value
LDFLD Int/Int::Value
LDFLD Int/Int::Value

Uh oh! We have to know the class.
Fortunately, GRIN stores this information for us:

\[ GrExpr\_FetchField \ x \ 1 \ (Just \ (GrTag\_Con \ \{1,1\} \ 0 \ Int)) \]

Phew.
However:

...; \lambda x \rightarrow
inc x; \lambda (y z) \rightarrow
...

- We have to extract the first field to bind to z.
- We need the \texttt{class} information for this. LDFLD ?/?::Value
- But we don’t know what y is!
We need the possible tags of every variable, so we can figure out which class to use.

Basically type (tag) inferencing. A lot of work!

Fortunately, the heap points-to analysis does this already.
Heap points-to analysis

The analysis gives us, for each variable, what kind of values it can contain.

Example:

\[
\begin{align*}
fetch T 1; x & \rightarrow \\
inc x ; \lambda(y z) & \rightarrow \\
update T (y z)
\end{align*}
\]

\(T\) is a thunk here.
Heap points-to analysis

\[
\begin{align*}
\text{fetch } T & \ 1; \ x \rightarrow \\
\text{inc } x & \quad ; \ \lambda(y \ z) \rightarrow \\
\text{update } T & \ (y \ z)
\end{align*}
\]

Variables:

- \( T \) Pointer \ [13,14]
- \( \text{inc} \) Node \ [((\text{CInt, [Basic]}))]
- \( x \) Pointer \ [13,14]
- \( y \) Tag \ CInt
- \( z \) Basic

Heap:

- 13 Node \ [((\text{CInt, [Basic]}))]
- 14 Node \ [((\text{CInt, [Basic]}), (\text{Finc, [Pointer [13,14]]})))]
Future work

Obvious enhancements

- stloc x, ldloc x
- more stack focussed code
  - Silly-like
  - tail calls!
- remove RefObj indirection
- use value types
- more polymorphic code
  - inline unboxed values
Future work
More ‘out there’ stuff

Simon Peyton Jones on Haskell for CLR:

- Generate IL
  - Runtime representation for thunks
- Interop with .NET libraries
  - No foreign import ... for everything
- Other GHC primitives:
  - the I/O monad
  - arbitrary precision arithmetic
  - concurrency
  - exceptions
  - finalisers
  - stable pointers
  - Software transactional memory
- Existing libraries
In conclusion

We think our runtime representation is workable.

We have an interesting prototype that shows this.

There’s much work still to be done...
EOF