Extended Abstract Machine for Prettyprinting Intermediate Computations

January 14, 2017

1 Compilation Scheme

```
type prog =
  Int of int
| Bool of bool
| VarAccess of int
| Eq of prog * prog
| Op of prog * op * prog
| Apply of prog * prog
| Lambda of prog
| Let of prog * prog
| If of prog * prog * prog

type instr =
  IEmpty
| IConst of int
| IBool of bool
| IOp of op
| IEq
| IAccess of int
| IClosure of instr list
| ILet
| IEndLet
| IApply
| IReturn
| IBranch
```

$$\begin{aligned}
\mathcal{C}(i) &= \text{INT}(i)\\
\mathcal{C}(b) &= \text{BOOL}(b)\\
\mathcal{C}(a \oplus b) &= \mathcal{C}(a); \mathcal{C}(b); \text{OP}(\oplus)\\
\mathcal{C}(a = b) &= \mathcal{C}(a); \mathcal{C}(b); \text{EQ}\\
\mathcal{C}(\underline{n}) &= \text{ACCESS}(n)\\
\mathcal{C}(\lambda a) &= \text{CLOSURE}(\mathcal{C}(a); \text{RETURN})\\
\mathcal{C}(\texttt{let}\ a\ \texttt{in}\ b) &= \mathcal{C}(a); \text{LET}; \mathcal{C}(b); \text{ENDLET}\\
\mathcal{C}(ab) &= \mathcal{C}(a); \mathcal{C}(b); \text{APPLY}\\
\mathcal{C}(\texttt{if}\ a\ \texttt{then}\ b\ \texttt{else}\ c) &= \mathcal{C}(\lambda b); \mathcal{C}(\lambda c); \mathcal{C}(a); \text{IF}
\end{aligned}$$

e.g `let x = 1 in let y = 2 in x + y` compiles to:

2 Evaluation Scheme

Machine state before			Machine state after		
Code	Env	Stack	Code	Env	Stack
INT$(i); c$	e	s	c	e	$i.s$
BOOL$(b); c$	e	s	c	e	$b.s$
OP$(\oplus); c$	e	$i.i'.s$	c	e	$\oplus(i, i').s$
EQ; c	e	$i.i'.s$	c	e	$(i = i').s$
ACCESS$(n); c$	e	s	c	e	$e(n).s$
CLOSURE$(c'); c$	e	s	c	e	$c'[e].s$
LET; c	e	$v.s$	c	$v.e$	s
ENDLET; c	$v.e$	s	c	e	s
APPLY;c	e	$v.c'[e'].s$	c'	$v.e'$	$c.e.s$
RETURN;c	e	$v.c'.e'.s$	c'	e'	$v.s$
IF;c	e	$\mathsf{T}.c'[e'].c''[e''].s$	c'	e'	$c[e].s$
IF;c	e	$\mathsf{F}.c'[e'].c''[e''].s$	c''	e''	$c[e].s$

The final result is at the top of the stack when the code is empty.

3 Decompilation Scheme

We need to be able to decompile:

- Any program which has been compiled by the compilation scheme above.

- Certain incomplete evaluations under the evaluation scheme above. That is to say, given (c, e, s) we can decompile a program which represents the evaluation at that stage.

We need not be able to decompile arbitrary (c, e, s) triples.

Extend ACCESS and LET with names, not required for evaluation, but for decompilation.

We add names to `VarAccess`, `Lambda` and `Let`:

```
type prog =
  Int of int
| Bool of bool
| VarAccess of name * int
| Eq of prog * prog
| Op of prog * op * prog
| Apply of prog * prog
| Lambda of name * prog
| Let of name * prog * prog
| If of prog * prog * prog

type instr =
  IEmpty
| IConst of int
| IBool of bool
| IOp of op
| IEq
| IAccess of name * int
| IClosure of name * instr list
| ILet of name
| IEndLet
| IApply
| IReturn
| IBranch
```

$$
\begin{aligned}
\mathcal{D}(\text{EMPTY}, v.s) &= v \\
\mathcal{D}(\text{INT}(i); c, s) &= \mathcal{D}(c, \texttt{Int}(i).s) \\
\mathcal{D}(\text{BOOL}(i); c, s) &= \mathcal{D}(c, \texttt{Bool}(b).s) \\
\mathcal{D}(\text{OP}(\oplus); c, i.i'.s) &= \mathcal{D}(c, \texttt{Op}(i, \oplus, i').s) \\
\mathcal{D}(\text{EQ}; c, i.i'.s) &= \mathcal{D}(c, \texttt{Eq}(i, i').s) \\
\mathcal{D}(\text{ACCESS}(n, l); c, s) &= \mathcal{D}(c, \texttt{VarAccess}(n, l).s) \\
\mathcal{D}(\text{CLOSURE}(n, c'); c, s) &= \mathcal{D}(c, c'[n, e].s) \\
\mathcal{D}(\text{LET}(n); c, v.s) &= \mathcal{D}(c, \texttt{Let}(n, v, \mathcal{D}(c, s').s)) \\
\mathcal{D}(\text{ENDLET}; c, s) &= \mathcal{D}(c, s) \\
\mathcal{D}(\text{APPLY}; c, v.c'[e'].s) &= \texttt{Apply}(\mathcal{D}(\text{EMPTY}, c'), v) \\
\mathcal{D}(\text{RETURN}; c, v.c'.e'.s) &= \mathcal{D}(v.s, c') \\
\mathcal{D}(\text{RETURN}; c, s) &= \mathcal{D}(c, s) \\
\mathcal{D}(\text{IF}; c, e.c'[e'].c''[e''].s) &= \mathcal{D}(c, \texttt{If}(e, \mathcal{D}(c', s), \mathcal{D}(c'', s)).s)
\end{aligned}
$$