Prettyprinting Intermediate Computations from a Bytecode

January 14, 2017

1 Programs

Programs are defined like this. Variable accesses have been converted to deBruijn indices when the program was converted from an OCaml one.

```
type op = Add | Sub | Mul | Div

type prog =
  Int of int
| Bool of bool
| Var 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
```

For example, the OCaml program

```
let x = 5 in if x = 4 then 1 else (fun x -> x + 1) 2
```

may be represented as:

```
Let(Int 5,
    If(Eq(Var 1, Int 4),
       Int 1,
       Apply(Lambda(Op(Var 1, Add, Int 1), Int 2))))
```

2 Compilation Scheme

The abstract machine instructions are as followed (Leroy plus BOOL, IF, EQ)

EMPTY
CONST(integer)
BOOL(boolean)
OP(op)
EQ
ACCESS(integer)
CLOSURE(instructions)
LET
ENDLET
APPLY
RETURN
IF

Here is the compilation scheme, again extended from Leroy:

$$\begin{aligned}
\mathcal{C}(\texttt{Int}(i)) &= \mathrm{INT}(i)\\
\mathcal{C}(\texttt{Bool}(b)) &= \mathrm{BOOL}(b)\\
\mathcal{C}(\texttt{Op}(a, \oplus, b)) &= \mathcal{C}(a); \mathcal{C}(b); \mathrm{OP}(\oplus)\\
\mathcal{C}(\texttt{Eq}(a, b)) &= \mathcal{C}(a); \mathcal{C}(b); \mathrm{EQ}\\
\mathcal{C}(\texttt{Var}(n)) &= \mathrm{ACCESS}(n)\\
\mathcal{C}(\texttt{Lambda}(a)) &= \mathrm{CLOSURE}(\mathcal{C}(a); \mathrm{RETURN})\\
\mathcal{C}(\texttt{Let}(a, b)) &= \mathcal{C}(a); \mathrm{LET}; \mathcal{C}(b); \mathrm{ENDLET}\\
\mathcal{C}(\texttt{Apply}(a, b)) &= \mathcal{C}(a); \mathcal{C}(b); \mathrm{APPLY}\\
\mathcal{C}(\texttt{If}(a, b, c)) &= \mathcal{C}(\texttt{Lambda}(b)); \mathcal{C}(\texttt{Lambda}(c)); \mathcal{C}(a); \mathrm{IF}
\end{aligned}$$

So our example compiles to:

```
ICONST 5
LET
CLOSURE
  CONST 1
  RETURN
CLOSURE
  CLOSURE
```

```
    ACCESS 1
    CONST 1
    OP +
    RETURN
  CONST 2
  APPLY
  RETURN
ACCESS 1
CONST 4
EQ
BRANCH
ENDLET
EMPTY
```

3 Evaluation Scheme

Here is the evaluation scheme $\mathcal{E}$, again extended from Leroy.

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.

4 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, 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
```

Similarly, we add names to the ACCESS, CLOSURE and LET instructions:

EMPTY
CONST(integer)
BOOL(boolean)
OP(op)
EQ
ACCESS(name, integer)
CLOSURE(name, instructions)
LET(name)
ENDLET
APPLY
RETURN
IF

Decompilation is performed by going through the instructions in order, holding a stack a little like the evaluation stack, but which may also contain decompiled program fragments. When we have gone through all the instructions, the final program is at the top of the stack. We do not need the environment, since we are not running the code, just decompiling it.

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

This decompiler works for:

- Any program-stack pair (P, EMPTYSTACK) where P was compiled by $\mathcal{C}$ above.
- program,stack pair (P, S) which is an intermediate state of the evaluation procedure $\mathcal{E}$ (minus the environment) where P begins with OP or APPLY.

Our example program decompiles properly from bytecode.

5 Worked examples

The following pages contain a worked example of the compilation $\mathcal{C}$, the evaluation $\mathcal{E}$, and full-program and partial-evaluation invocations of the decompiler $\mathcal{D}$.

Compilation under $\mathcal{C}$:

$\mathcal{C}$(`Let(Int 5, If(Eq(Var 1, Int 4), Int 1, Apply(Lambda(Op(Var 1, Add, Int 1), Int 2)))))`

Rule $\mathcal{C}$-`Let`

$\mathcal{C}$(`Int 5`); LET; $\mathcal{C}$(`If(Eq(Var 1, Int 4), Int 1, Apply(Lambda(Op(Var 1, Add, Int 1), Int 2))))`); ENDLET

Rule $\mathcal{C}$-`Int`

INT 5; LET; $\mathcal{C}$(`If(Eq(Var 1, Int 4), Int 1, Apply(Lambda(Op(Var 1, Add, Int 1), Int 2))))`); ENDLET

Rule $\mathcal{C}$-`If`

INT 5; LET; $\mathcal{C}$(`Lambda (Int 1)`); $\mathcal{C}$(`Lambda(Apply(Lambda(Op(Var 1, Add, Int 1), Int 2)))`); $\mathcal{C}$(`Eq(Var 1, Int 4)`); IF; ENDLET

Rule $\mathcal{C}$-`Eq` then Rule $\mathcal{C}$-`Eq` then Rule $\mathcal{C}$-`Eq`

INT 5; LET; $\mathcal{C}$(`Lambda (Int 1)`); $\mathcal{C}$(`Lambda(Apply(Lambda(Op(Var 1, Add, Int 1), Int 2)))`); VARACCESS 1; INT 4; EQ; IF; ENDLET

Rule $\mathcal{C}$-`Lambda` then Rule $\mathcal{C}$-`Int`

INT 5; LET; CLOSURE [INT 1; RETURN]; $\mathcal{C}$(`Lambda(Apply(Lambda(Op(Var 1, Add, Int 1), Int 2)))`); VARACCESS 1; INT 4; EQ; IF; ENDLET

Rule $\mathcal{C}$-`Lambda`

INT 5; LET; CLOSURE [INT 1; RETURN]; CLOSURE [$\mathcal{C}$(`Apply(Lambda(Op(Var 1, Add, Int 1), Int 2))`); RETURN]; VARACCESS 1; INT 4; EQ; IF; ENDLET

Rule $\mathcal{C}$-`Apply`

INT 5; LET; CLOSURE [INT 1; RETURN]; CLOSURE [$\mathcal{C}$(`Lambda(Op(Var 1, Add, Int 1))`; $\mathcal{C}$(`Int 2`); APPLY; RETURN]; VARACCESS 1; INT 4; EQ; IF; ENDLET

Rule $\mathcal{C}$-`Int` then Rule $\mathcal{C}$-`Lambda` then Rule $\mathcal{C}$-`Op` then Rule $\mathcal{C}$-`Var` then Rule $\mathcal{C}$-`Int`

ICONST 5; LET; CLOSURE [CONST 1; RETURN]; CLOSURE [CLOSURE [ACCESS 1; CONST 1; OP +; RETURN]; CONST 2; APPLY; RETURN]; ACCESS 1; CONST 4; EQ; IF; ENDLET