Compiler Construction

Clever Translations

Translating expressions

How to translate the expression
$\alpha < \beta$ in HIR?

Naive translation

```
eseq
  seq
   cjump (α < β) ltrue lfalse
   label ltrue
   move temp t const 1
   jump lend
   label lfalse
   move temp t const 0
   label lend
  seq end
  temp t
```

Closer look to the naive translation

Naive translation is costly

- one cjump
- one jump
- two label
- one temporary

$\Rightarrow$ Can we do better?

Note: *jumps and cjumps are costly is the microprocessor*
$\Rightarrow$ We must try to minimize them!

Can we exploit additional information?

```
let
    ...
in
   α < β,
   ...
end
```

In this situation we don't care about the translation of $<$
$\Rightarrow$ We are only interested about side effect of α and β

Improved translation

```
seq
   sxp $\alpha$
   sxp $\beta$
seq end
```

Improved translation

```
seq
   sxp α
   sxp β
seq end
```

- 0 cjump / 0 jumps
- 0 label
- 0 temporary

$\Rightarrow$ Better!

Yet another example

```
let
    ...
in
    if α < β
    then /*TRUE*/
    else /*FALSE*/
end
```

In this situation a naive translation would produce a lot of useless jump/cjump

Improved translation

```
cjump α < β ltrue, lfalse
label ltrue
/* TRUE translation */
label lfalse
/* FALSE translation */
```

Only one cjump (and one jump an the end of ltrue)!
$\Rightarrow$ Better than the naive translation!

Translating Conditions

What is the right translation for $\alpha < \beta$,
with α and β two arbitrary expressions?

It depends on the *use*:

1. `if` $\alpha < \beta$ `then ...`
2. `a :=` $\alpha < \beta$
3. `(`$\alpha < \beta$`, ())`.

Problem statement

When the visitor is about to translate $\alpha < \beta$, it does not know the context.

Context Sensitive Translation

- The right translation depends upon the *use*. **This is context sensitive!**
- How to implement this?
  - When entering an *IfExp*, warn "I want a condition",
  - then, depending whether it is an expression or a statement, warn "I want an expression" or "I want a statement".
- Don't forget to preserve the demands of higher levels...
- Eek.

Prototranslation, Expression Shells

Rather, delay the translation until the use is known:

Ex	Expression shell, encapsulation of a proto value,
Nx	Statement shell, encapsulating a wannabe statement,
Cx	Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)			
Cx($a < b$)			
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known:

Ex	Expression shell, encapsulation of a proto value,
Nx	Statement shell, encapsulating a wannabe statement,
Cx	Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)		
Cx($a < b$)			
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known:

Ex	Expression shell, encapsulation of a proto value,
Nx	Statement shell, encapsulating a wannabe statement,
Cx	Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	
Cx($a < b$)			
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known:

Ex	Expression shell, encapsulation of a proto value,
Nx	Statement shell, encapsulating a wannabe statement,
Cx	Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump($e \neq 0$, t, f)
Cx($a < b$)			
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known:

Ex	Expression shell, encapsulation of a proto value,
Nx	Statement shell, encapsulating a wannabe statement,
Cx	Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump($e \neq 0$, t, f)
Cx($a < b$)	seq(sxp(a), sxp(b))		
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known:

Ex	Expression shell, encapsulation of a proto value,
Nx	Statement shell, encapsulating a wannabe statement,
Cx	Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump($e \neq 0$, t, f)
Cx($a < b$)	seq(sxp(a), sxp(b))	eseq($t \leftarrow (a < b)$, t)	
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known:

Ex	Expression shell, encapsulation of a proto value,
Nx	Statement shell, encapsulating a wannabe statement,
Cx	Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump($e \neq 0$, t, f)
Cx($a < b$)	seq(sxp(a), sxp(b))	eseq($t \leftarrow (a < b)$, t)	cjump($a < b$, t, f)
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known:

Ex	Expression shell, encapsulation of a proto value,
Nx	Statement shell, encapsulating a wannabe statement,
Cx	Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump($e \neq 0$, t, f)
Cx($a < b$)	seq(sxp(a), sxp(b))	eseq(t $\leftarrow$($a < b$), t)	cjump($a < b$, t, f)
Nx(s)	s		

Prototranslation, Expression Shells

Rather, delay the translation until the use is known:

Ex	Expression shell, encapsulation of a proto value,
Nx	Statement shell, encapsulating a wannabe statement,
Cx	Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump($e \neq 0$, t, f)
Cx($a < b$)	seq(sxp(a), sxp(b))	eseq($t \leftarrow (a < b)$, t)	cjump($a < b$, t, f)
Nx(s)	s	**error**	

Prototranslation, Expression Shells

Rather, delay the translation until the use is known:

Ex	Expression shell, encapsulation of a proto value,
Nx	Statement shell, encapsulating a wannabe statement,
Cx	Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump($e \neq 0$, t, f)
Cx($a < b$)	seq(sxp(a), sxp(b))	eseq(t $\leftarrow$($a < b$), t)	cjump($a < b$, t, f)
Nx(s)	s	**error**	**error**

if 11 < 22 | 22 < 33 then print_int(1) else print_int(0)

```
cjump ne
    eseq seq cjump 11 < 22 name l0 name l1
               label l0  move temp t0 const 1
                         jump name l2
               label l1  move temp t0
                         eseq seq move temp t1 const 1
                                  cjump 22 < 33 name l3 name l4
                                  label l4
                                  move temp t1 const 0
                                  label l3
                              seq end
                              temp t1
                        jump name l2
               label l2
         seq end
         temp t0
    const 0
    name l5
    name l6
label l5   sxp call name print_int const 1
           jump name l7
label l6   sxp call name print_int const 0
           jump name l7
label l7
```

A Better Translation: lx

```
seq
    cjump 11 < 22 name l3 name l4
    label l3
      cjump 1 <> 0 name l0 name l1
    label l4
      cjump 22 < 33 name l0 name l1
  seq end
label l0
  sxp call name print_int const 1
  jump name l2
label l1
  sxp call name print_int const 0
label l2
```

Summary

Naive translation

Proto translation