CS-320

Computer Language Processing

Exercise Session 6

November 15, 2017

Overview

Today we'll explore a few more aspects of generating code for control flow constructs.

WebAssembly's *if*

WebAssembly has dedicated bytecodes for if expressions, i.e., *if*, *else*, *end*:

```
[e_cond]
if
   [e_then]
else
   [e_else]
end
[e_rest]
```

▷ Given the *block* and *br[_if]* instructions you saw this construct isn't necessary. How can we desugar snippets like the above?

WebAssembly's *if*

▷ Given the *block* and *br[_if]* instructions you saw this construct isn't necessary. How can we desugar snippets like the above?

```
block nAfter
  block nElse
    [!e_cond]
    br_if nElse
    [e_then]
    br nAfter
  end //nElse:
  [e_else]
end //nAfter:
[e_rest]
```

WebAssembly's *if*

▷ Given the *block* and *br[_if]* instructions you saw this construct isn't necessary. How can we desugar snippets like the above?

```
block nAfter
  block nElse
    [!e_cond]
    br_if nElse
    [e_then]
    br nAfter
  end //nElse:
  [e_else]
end //nAfter:
[e_rest]
```

▷ Can we avoid the negation on the branching condition e_{cond}?

Avoiding negation

- Can we avoid the negation on the branching condition e_{cond}?

```
block nAfter
  block nThen
    [e_cond]
    br_if nThen
    [e_else]
    br nAfter
  end //nThen:
  [e_then]
end //nAfter:
[e_rest]
```

Translating control flow structures more efficiently

Introduce an imaginary large instruction
`branch``(c,nThen,nElse)`.

Here `c` is a potentially complex boolean expression (the main reason why **`branch`** is not a built-in bytecode instruction), whereas `nTrue` and `nFalse` are the labels we jump to depending on the boolean value of `c`.

We will show how to

- use **`branch`** to compile **`if`** and short-circuiting operators,
- by expanding **`branch`** recursively into concrete bytecode instructions.

Translating control flow structures more efficiently

Implementing *if* using `branch`

$[\![\mathtt{if}\ (e_{cond})\ e_{then}\ \mathtt{else}\ e_{else}]\!] :=$

```
block nAfter
  block nElse
    block nThen
      branch(e_cond, nThen, nElse)
    end //nThen:
    [e_then]
    br nAfter
  end //nElse:
  [e_else]
end //nAfter:
[e_rest]
```

Decomposing conditions in branch

Negation and short-circuiting operators

```
branch(!e,nThen,nElse) :=
  branch(e,nElse,nThen)

branch(e1 && e2,nThen,nElse) :=
  block nLong
    branch(e1,nLong,nElse)
  end //nLong:
  branch(e2,nThen,nElse)

branch(e1 || e2,nThen,nElse) :=
  block nLong
    branch(e1,nThen,nLong)
  end //nLong:
  branch(e2,nThen,nElse)
```

Decomposing conditions in branch

Constant case and variable loads

```
branch(true,nThen,nElse) :=
  br nThen

branch(false,nThen,nElse) :=
  br nElse

branch(b,nThen,nElse) :=   (where b is a local var)
  get_local #b
  br_if nThen
  br nElse
```

Decomposing conditions in branch

Other built-in relations

```
branch(e1 == e2,nThen,nElse) :=
  [e1]
  [e2]
  i32.eq
  br_if nThen
  br nElse
```

(*where* e_1, e_2 *are of type* int)

`...` (*analogously for other relations*)

Returning the result from branch

Consider storing $x = c$
where x, c are boolean and c contains $\&\&$ or $\|$.

How do we put the result of c on the stack so it can be stored in x?

$[x = c]$:=

```
block nAfter
  block nElse
    block nThen
      branch(c,nThen,nElse)
    end //nThen:
    i32.const 1
    br nAfter
  end //nElse:
  i32.const 0
end //nAfter:
set_local #x
```

Destination label parameters

Recall that in **`branch`**`(c,nThen,nElse)` we had two arguments `nThen` and `nElse`, which told us where to jump to execute code of the corresponding branches.

Similarly, up until now we explicitly enclosed our translated program fragments in an `nAfter` block, so we could jump to the "rest" of the program.

Destination label parameters

Recall that in **`branch`**`(c,nThen,nElse)` we had two arguments `nThen` and `nElse`, which told us where to jump to execute code of the corresponding branches.

Similarly, up until now we explicitly enclosed our translated program fragments in an `nAfter` block, so we could jump to the "rest" of the program.

$\Rightarrow$ We can generalize our translation function $[\![\, \cdot \,]\!]$ to take a destination label designating the "rest" in the surrounding code.

Destination label parameters

Recall that in **`branch`**`(c,nThen,nElse)` we had two arguments `nThen` and `nElse`, which told us where to jump to execute code of the corresponding branches.

Similarly, up until now we explicitly enclosed our translated program fragments in an `nAfter` block, so we could jump to the "rest" of the program.

$\Rightarrow$ We can generalize our translation function $[\![\cdot]\!]$ to take a destination label designating the "rest" in the surrounding code.

$$[\![\cdot]\!] \Rightarrow [\![\cdot]\!]\,\texttt{nAfter}$$

$\Rightarrow$ The caller of the translation function determines where to continue!

Translations with an nAfter label parameter (1)

$[x = e]$ `nAfter :=`
```
  block nSet
    [e] nSet
    // (note that the rest of this block is never reached!)
  end //nSet:
  set_local #x
  br nAfter
```

$[s_1; s_2]$ `nAfter :=`
```
  block nSecond
    [s1] nSecond
  end //nSecond:
  [s2] nAfter
```

Translations with an `nAfter` label parameter (2)

```
[if (e_cond) e_then else e_else] nAfter :=
 block nElse
   block nThen
     branch(e_cond,nThen,nElse)
   end //nThen:
   [e_then] nAfter
 end //nElse:
 [e_else] nAfter

[return e] nAfter :=
 block nRet
   [e] nRet
 end //nRet:
 return
```

Switch statements

Let us assume our language had a switch statement (like C and Java do, for instance):

$$
\begin{array}{l}
\textbf{switch}\ \ (e_{scrutinee})\ \ \{ \\
\quad \textbf{case}\ \ c_1:\ \ e_1 \\
\quad \ldots \\
\quad \textbf{case}\ \ c_n:\ \ e_n \\
\quad \textbf{default}:\ \ e_{default} \\
\}
\end{array}
$$

▷ How can we compile such switch statements?

Compiling switch statements

```
[s_switch] nAfter :=
 block nDefault
   block nCase_n
     ...
       block nCase_1
         block nTest
           [e_scrutinee] nTest
         end //nTest:
         tee_local #s   (where s is some fresh local of type i32)
         i32.const c_1; i32.eq; br_if nCase_1
         get_local #s
         i32.const c_2; i32.eq; br_if nCase_2
         ...
         br nDefault
       end //nCase_1:
       [e_1] nCase_2
     ...
   end //nCase_n:
   [e_n] nDefault
 end //nDefault:
 [e_default] nAfter
```

Compiling switch statements

```
[s_switch] nAfter :=
 block nDefault
   block nCase_n
    ...
     block nCase_1
       block nTest
         [e_scrutinee] nTest
       end //nTest:
       tee_local #s   (where s is some fresh local of type i32)
       i32.const c_1; i32.eq; br_if nCase_1
       get_local #s
       i32.const c_2; i32.eq; br_if nCase_2
       ...
       br nDefault
     end //nCase_1:
     [e_1] nCase_2
    ...
   end //nCase_n:
   [e_n] nDefault
 end //nDefault:
 [e_default] nAfter
```

▷ How do we translate `break`?

Compiling switch statements

Translating `break`

At any point during the translation of **`switch`** we want to keep track not only where to jump *after*, but also where to jump on a `break`!

Compiling switch statements

Translating `break`

At any point during the translation of **`switch`** we want to keep track not only where to jump *after*, but also where to jump on a `break`!

$\Rightarrow$ Let us extend the translation function by another label parameter.

Compiling switch statements

Translating `break`

At any point during the translation of **`switch`** we want to keep track not only where to jump *after*, but also where to jump on a `break`!

⇒ Let us extend the translation function by another label parameter.

$$[\,\cdot\,]\ \texttt{nAfter} \;\Rightarrow\; [\,\cdot\,]\ \texttt{nAfter}\ \texttt{nBreak}$$

⇒ The caller of the translation function determines where to continue in the "normal" case, but also when `break` is called!

Compiling switch statements

Translating `break`

Translating `break` then is straightforward: One simply ignores `nAfter` and follows `nBreak` instead.

```
[break] nAfter nBreak :=
  br nBreak
```

▷ What do we have change in our translation of `switch` statements?

Compiling switch statements with breaks

```
[s_switch] nAfter nBreak :=
 block nDefault
  block nCase_n
   ...
    block nCase_1
     block nTest
      [e_scrutinee] nTest nBreak
     end //nTest:
     tee_local #s   (where s is some fresh local of type i32)
     i32.const c_1; i32.eq; br_if nCase_1
     get_local #s
     i32.const c_2; i32.eq; br_if nCase_2
     ...
     br nDefault
    end //nCase_1:
    [e_1] nCase_2 nAfter
   ...
  end //nCase_n:
  [e_n] nDefault nAfter
 end //nDefault:
 [e_default] nAfter nAfter
```