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*:

1 \([e_{\text{cond}}]\)
2 \(\text{if}\)
3 \([e_{\text{then}}]\)
4 \(\text{else}\)
5 \([e_{\text{else}}]\)
6 \(\text{end}\)
7 \([e_{\text{rest}}]\)

▷ Given the *block* and \(br[_\text{if}]\) 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?

```plaintext
1 block  nAfter
2   block  nElse
3     [!e_{cond}]
4   br_if  nElse
5     [e_{then}]
6   br  nAfter
7 end  //nElse:
8     [e_{else}]
9 end  //nAfter:
10    [e_{rest}]
```

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

```plaintext
1 block  nAfter
2 block  nElse
3  [!e_{cond}]
4 br_if  nElse
5   [e_{then}]
6 br  nAfter
7 end  //nElse:
8  [e_{else}]
9 end  //nAfter:
10  [e_{rest}]
```

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

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

```plaintext
1  block nAfter
2  block nThen
3   [e_{cond}]
4   br_if nThen
5   [e_{else}]
6  br nAfter
7 end //nThen:
8   [e_{then}]
9 end //nAfter:
10 [e_{rest}]
```
Translating control flow structures more efficiently

Introduce an imaginary large instruction \texttt{branch}(c,nThen,nElse).

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

We will show how to

- use \texttt{branch} to compile \texttt{if} and short-circuiting operators,
- by expanding \texttt{branch} recursively into concrete bytecode instructions.
Translating control flow structures more efficiently

Implementing if using branch

\[
[\text{if} \ (e_{\text{cond}}) \ e_{\text{then}} \ \text{else} \ e_{\text{else}}] := \\
\text{block} \ n_{\text{After}} \\
\text{block} \ n_{\text{Else}} \\
\text{block} \ n_{\text{Then}} \\
\quad \text{branch}(e_{\text{cond}}, \ n_{\text{Then}}, \ n_{\text{Else}}) \\
\text{end} \ //n_{\text{Then}}: \\
[\ e_{\text{then}} \] \\
\text{br} \ n_{\text{After}} \\
\text{end} \ //n_{\text{Else}}: \\
[\ e_{\text{else}} \] \\
\text{end} \ //n_{\text{After}}: \\
[\ e_{\text{rest}} \]
\]
Decomposing conditions in branch

Negation and short-circuiting operators

\[
\begin{align*}
\text{branch}(!e, \text{nThen}, \text{nElse}) & := \\
& \quad \text{branch}(e, \text{nElse}, \text{nThen}) \\
\text{branch}(e_1 \land e_2, \text{nThen}, \text{nElse}) & := \\
& \quad \text{block } \text{nLong} \\
& \quad \text{branch}(e_1, \text{nLong}, \text{nElse}) \\
& \quad \text{end } //\text{nLong:} \\
& \quad \text{branch}(e_2, \text{nThen}, \text{nElse}) \\
\text{branch}(e_1 \lor e_2, \text{nThen}, \text{nElse}) & := \\
& \quad \text{block } \text{nLong} \\
& \quad \text{branch}(e_1, \text{nThen}, \text{nLong}) \\
& \quad \text{end } //\text{nLong:} \\
& \quad \text{branch}(e_2, \text{nThen}, \text{nElse})
\end{align*}
\]
Decomposing conditions in branch

Constant case and variable loads

\[
\text{branch}(true,nThen,nElse) := \text{br } nThen \\
\text{branch}(false,nThen,nElse) := \text{br } nElse \\
\text{branch}(b,nThen,nElse) := \text{ (where } b \text{ is a local var)} \\
\quad \text{get\_local } #b \\
\quad \text{br\_if } nThen \\
\quad \text{br } nElse
\]
Decomposing conditions in \textit{branch}

Other built-in relations

\begin{verbatim}
branch(e_1 == e_2, nThen, nElse) := (where e_1, e_2 are of type int)
  [e_1]
  [e_2]
  i32.eq
  br_if nThen
  br nElse

... (analogously for other relations)
\end{verbatim}
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 \texttt{branch}(c,nThen,nElse) we had two arguments \texttt{nThen} and \texttt{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 \texttt{nAfter} block, so we could jump to the “rest” of the program.

⇒ We can generalize our translation function [·] to take a destination label designating the “rest” in the surrounding code.
Recall that in \texttt{branch}(c, nThen, nElse) we had two arguments \texttt{nThen} and \texttt{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 \texttt{nAfter} block, so we could jump to the “rest” of the program.

⇒ We can generalize our translation function \([\cdot]\) to take a destination label designating the “rest” in the surrounding code.

\[
[\cdot] \Rightarrow [\cdot] \texttt{nAfter}
\]

⇒ 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
  \[ s_1 \] nSecond
end //nSecond:
\[ s_2 \] nAfter
Translations with an nAfter label parameter (2)

\[
[\textit{if} \ (e_{\textit{cond}}) \ e_{\textit{then}} \ \textit{else} \ e_{\textit{else}}] \ nAfter :=
\]
\[
\textbf{block} \ nElse
\]
\[
\textbf{block} \ nThen
\]
\[
\text{branch}(e_{\textit{cond}}, nThen, nElse)
\]
\[
\textbf{end} \ //nThen:
\]
\[
[e_{\textit{then}}] \ nAfter
\]
\[
\textbf{end} \ //nElse:
\]
\[
[e_{\textit{else}}] \ nAfter
\]

\[
[\textit{return} \ e] \ nAfter :=
\]
\[
\textbf{block} \ nRet
\]
\[
[e] \ nRet
\]
\[
\textbf{end} \ //nRet:
\]
\[
\textbf{return}
\]
Switch statements

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

```plaintext
switch (e_{scrutinee}) {
    case c_1: e_1 
    ...
    case c_n: e_n 
    default: e_{default}
}
```

▷ How can we compile such switch statements?
Compiling switch statements

```assembly
[s\text{switch}] \text{nAfter} :=
\text{block} \text{nDefault}
\text{block} \text{nCase}_n
...
\text{block} \text{nCase}_1
\text{block} \text{nTest}
\text{[\text{e}_{\text{scrutinee}}]} \text{nTest}
end //\text{nTest}:
\text{tee_local} #s (where s is some fresh local of type i32)
i32\text{.const } c_1; i32\text{.eq}; \text{br_if} \text{nCase}_1
\text{get_local} #s
i32\text{.const } c_2; i32\text{.eq}; \text{br_if} \text{nCase}_2
...
\text{br} \text{nDefault}
end //\text{nCase}_1:
\text{[\text{e}_1]} \text{nCase}_2
...
end //\text{nCase}_n:
\text{[\text{e}_n]} \text{nDefault}
end //\text{nDefault}:
\text{[\text{e}_{\text{default}}]} \text{nAfter}
```
Compiling switch statements

\[
\text{\texttt{\texttt{ switch }}\ nAfter := \texttt{block\ nDefault}}
\]
\[
\text{\texttt{block\ nCase_1}}
\]
\[
\text{\texttt{block\ nTest}}
\]
\[
\text{\texttt{[e_{\text{scrutinee}}]\ nTest}}
\]
\[
\text{\texttt{end}}\ //\texttt{nTest:}
\]
\[
\text{\texttt{tee_local\ \#s\ (where\ s\ is\ some\ fresh\ local\ of\ type\ i32)}}
\]
\[
\text{\texttt{i32.const\ c_1;\ i32.eq;\ br_if\ nCase_1}}
\]
\[
\text{\texttt{get_local\ \#s}}
\]
\[
\text{\texttt{i32.const\ c_2;\ i32.eq;\ br_if\ nCase_2}}
\]
\[
\text{\texttt{end}}\ //\texttt{nCase_1:}
\]
\[
\text{\texttt{[e_1]\ nCase_2}}
\]
\[
\text{\texttt{end}}\ //\texttt{nCase_n:}
\]
\[
\text{\texttt{[e_n]\ nDefault}}
\]
\[
\text{\texttt{end}}\ //\texttt{nDefault:}
\]
\[
\text{\texttt{[e_{default}]\ nAfter}}
\]

▷ How do we translate 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!
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.
Compiling switch statements

Translating break

At any point during the translation of \texttt{switch} we want to keep track not only where to jump \textit{after}, but also where to jump on a \texttt{break}!

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

\[
[\cdot] \text{nAfter} \Rightarrow [\cdot] \text{nAfter nBreak}
\]

⇒ The caller of the translation function determines where to continue in the “normal” case, but also when \texttt{break} is called!
Compiling switch statements

Translating break

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

\[
\text{[break]} \ nAfter \ nBreak := \ \text{br} \ nBreak
\]

▷ What do we have change in our translation of switch statements?
Compiling switch statements with breaks

```
[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
```