CS 131
Programming Languages

November 30, 2000
Staged Computation
Staged Computation

• Many algorithms naturally can be divided into stages
  - Later stages exploit results of earlier stages for efficiency

• Earlier stages may not depend on all inputs
  - Naturally expressed by curried function types
    \[
    \text{send} : \text{connection} \to (\text{data} \to \text{unit})
    \]
  - Efficiencies possible by reusing results of earlier stages

\[
\text{val send\_to\_c } \equiv \text{send c}
\]

\[
\ldots\text{send\_to\_c(msg1)}\ldots\text{send\_to\_c(msg2)}\ldots
\]
Today's topics

• Two techniques supporting staged computation
  - Partial Evaluation
  - Run-time Code Generation
Consider this code

```plaintext
fun power(x,n) =
  if (n=0) then
    1.0
  else
    x*power(x,n-1)

...power(x1,3)...power(x2,3)...etc.
```
fun power(x,n) =
    if (n=0) then
        1.0
    else
        x*power(x,n-1)
fun cube x = power(x,3)

cube(x1) ... cube(x2) ... etc.
fun power n =
  (fn x =>
    if (n=0) then
      1.0
    else
      x*power (n-1) x)
val cube = power 3

...cube(x1)...cube(x2)...etc.
fun power n =
  if (n=0) then
    (fn x => 1.0)
  else
    let val recurse = power(n-1)
    in
      fn x => x * (recurse x)
    end
  end
val cube = power 3
...cube(x1)...cube(x2)...etc.
General or Specialized Code?

• By making code general, it can be applied to many different problems.

• Code specialized to a particular problem usually runs faster.

• For example:
  - "power" vs. "cube"
  - "matrix multiply" vs. "3x3 matrix multiply"

• Idea: general-purpose programs that generate special-purpose code
Partial Evaluation

- Program transformation:
  - Takes the code for a program and some inputs
  - Returns code for a program to take the rest of the inputs and finish the computation.
  - That is, creates a specialized version of the program for the given inputs
  - The hope is that the specialized version will be faster/smaller/better than simply calling the original program with all the inputs.
Programs as Data and Behavior

• We will denote program code as \( \text{prog} \).
• We denote the meaning of this by \( \{ \text{prog} \} \)
  - That is, as a function from inputs to outputs.
  - So \( \{ \text{prog} \}(x) \) refers to the result of running the code \( \text{prog} \) and supplying it the input \( x \).

• We will sometimes write \( \{ \text{prog} \}_L \) to emphasize that we are viewing it as a program in language \( L \).
Formalizing Partial Evaluation

• Suppose the program $p$ in language $s$ takes two inputs:

$$[p]_s(m, n) = \text{output}$$

• If $\text{mix}$ is (code for) a partial evaluator then

$$[\text{mix}](p, m) = p^m$$

where

$$[p^m]_s(n) = \text{output} = [p]_s(m, n)$$

• That is, if

$$[[\text{mix}](p, m)]_s(n) = [p]_s(m, n)$$
Applications of Partial Evaluation

• Ray tracing
  - Fix the scene, repeatedly compute information about light rays.

• Neural networks
  - Fix the network topology, repeatedly simulate to train

• Scientific computing
  - Fix the layout of the circuit being simulated, ...
  - Fix the number of bodies whose orbit is being calculated, ...

• String search
  - Fix the string being sought, ...
# Interpreters and Compilers

- We say `int` is an interpreter for language $S$ if, given any program $\text{source}$ (written in $S$), we have
  \[
  \text{output} = [\text{source}]_S(\text{input}) \\
  = [\text{int}]_L(\text{source}, \text{input})
  \]

- We say $\text{comp}$ is a compiler from $S$ to $T$ if
  \[
  \text{output} = [\text{source}]_S(\text{input}) \\
  = [[\text{compiler}]_L(\text{source}) ]_T(\text{input})
  \]
  i.e.,
  \[
  \text{target} = [\text{compiler}]_L(\text{source})
  \]
  and
  \[
  [\text{source}]_S(\text{input}) = [\text{target}]_T(\text{input})
  \]
Compiling via Partial Evaluation

• Suppose we have an interpreted program $source$ which we run on many inputs.
  - That is, we are running the interpreter many times with one input ($source$) unchanging.
  - Why not use partial evaluation?

Put $target := \text{mix}(int, source)$
then $output = [source]_S(input)$
  $= [int]_L(source, input)$
  $= [[\text{mix}(int, source)]_L(input)$
  $= [target]_L(input)$
Compiling via Partial Evaluation

• Suppose we want to compute
  \[ \text{mix}\{\text{int,source}\} \]
  for many source programs.
  - That is, we are running \text{mix} many times with one input (\text{int}) unchanging.
  - Why not use partial evaluation?

  Put \texttt{compiler} := \text{mix}\{(\text{mix,\text{int}})\}

  then \texttt{target} = \text{mix}\{(\text{int,source})\}
  = \left(\text{mix}\{(\text{mix,\text{int}})\}\right)(\text{source})
  = \text{[compiler]}(\text{source})
Compiling via Partial Evaluation

- Suppose we want to compute 
\[ [mix](mix, int) \]
for many different interpreters.
  - That is, we are running \texttt{mix} many times with one input \texttt{(mix)} unchanging.
  - Why not use partial evaluation?

Put \texttt{cogen := [mix](mix, mix)}
then \texttt{compiler = [mix](mix, int)}
\[ = [[mix](mix, mix)](int) \]
\[ = [cogen](source) \]
Run-Time Code Generation

• Sometimes the early inputs to code aren't known until a program is running
  - Determined by user input or configuration file
  - Nested loops
    • Values in outer loop fixed while inner loop executes

• Run-Time Code Generation (RTCG)
  - Dynamically extending a program with new code (machine or bytecode)
  - Descendant of self-modifying code