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

```
send  : connection -> (data -> unit)
```

  - Efficiencies possible by reusing results of earlier stages

```
val send_to_c  = send c
....send_to_c(msg1)...send_to_c(msg2)...
```

Today's topics

- Two techniques supporting staged computation
  - Partial Evaluation
  - Run-time Code Generation

Consider this code

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

...power(x1,3)...power(x2,3)...etc.
```

Version 2

```
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.
```

Version 3

```
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.
```

Version 4

```
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
 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 **`prog`**.
- We denote the meaning of this by $[\![\mathbf{prog}]\!]$
  - That is, as a function from inputs to outputs.
  - So $[\![\mathbf{prog}]\!]\mathbf{(x)}$ refers to the result of running the code **`prog`** and supplying it the input **`x`**.

- We will sometimes write $[\![\mathbf{prog}]\!]_{\mathbf{L}}$ 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:

$$[\![\mathtt{p}]\!]_{\mathtt{S}}(\mathtt{m},\mathtt{n}) = \mathtt{output}$$

- If **mix** is (code for) a partial evaluator then

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

where

$$[\![\mathtt{p}^{\mathtt{m}}]\!]_{\mathtt{S}}(\mathtt{n}) = \mathtt{output} = [\![\mathtt{p}]\!]_{\mathtt{S}}(\mathtt{m},\mathtt{n})$$

- That is, if

$$[\![\, [\![\mathtt{mix}]\!](\mathtt{p},\mathtt{m})]\!]_{\mathtt{S}}(\mathtt{n}) = [\![\mathtt{p}]\!]_{\mathtt{S}}(\mathtt{m},\mathtt{n})$$

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 **`source`** (written in **`S`**), we have

$$\texttt{output} = [\![\texttt{source}]\!]_S(\texttt{input})$$
$$= [\![\texttt{int}]\!]_L(\texttt{source},\texttt{input})$$

- We say **`comp`** is a compiler from **`S`** to **`T`** if

$$\texttt{output} = [\![\texttt{source}]\!]_S(\texttt{input})$$
$$= [\![\, [\![\texttt{compiler}]\!]_L(\texttt{source})\,]\!]_T(\texttt{input})$$

i.e., $\texttt{target} = [\![\texttt{compiler}]\!]_L(\texttt{source})$

and $[\![\texttt{source}]\!]_S(\texttt{input}) = [\![\texttt{target}]\!]_T(\texttt{input})$

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 $\texttt{target} := [\![\texttt{mix}]\!](\texttt{int},\texttt{source})$

then
$$
\begin{aligned}
\texttt{output} &= [\![\texttt{source}]\!]_S(\texttt{input}) \\
&= [\![\texttt{int}]\!]_L(\texttt{source},\texttt{input}) \\
&= [\![[\![\texttt{mix}]\!](\texttt{int},\texttt{source})]\!]_L(\texttt{input}) \\
&= [\![\texttt{target}]\!]_L(\texttt{input})
\end{aligned}
$$

Compiling via Partial Evaluation

- Suppose we want to compute

  $[\![$**`mix`**$]\!]$**`(int,source)`**

  for many source programs.
  - That is, we are running **`mix`** many times with one input (**`int`**) unchanging.
  - Why not use partial evaluation?

  Put **`compiler := `**$[\![$**`mix`**$]\!]$**`(mix,int)`**

  then **`target = `**$[\![$**`mix`**$]\!]$**`(int,source)`**

  `= `$[\![[\![$**`mix`**$]\!]$**`(mix,int)`**$]\!]$**`(source)`**

  `= `$[\![$**`compiler`**$]\!]$**`(source)`**

Compiling via Partial Evaluation

- Suppose we want to compute

  $[\![$**`mix`**$]\!]$**`(mix,int)`**

  for many different interpreters.
  - That is, we are running **`mix`** many times with one input (**`mix`**) unchanging.
  - Why not use partial evaluation?

  Put **`cogen := `**$[\![$**`mix`**$]\!]$**`(mix,mix)`**

  then **`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