Partial Evaluation

April 4, 2011
CS 132: Compiler Design
Staged Computation

• Many algorithms naturally can be divided into stages
  – Later stages exploit results of earlier stages for efficiency
  – Includes compile-time, link-time, various points at run-time

• 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)....
Specializing Code

• One instance of staged computation
  – Given "early" arguments, create code to take "late" arguments and does computation
  – General-purpose programs to generate special-purpose code
  – Hope that specialized version is faster/smaller/better.

• Useful when...
  – Speedup from using special-purpose code outweights cost of code-generation
  – Special-purpose code can be repeatedly re-used
Partial Evaluation

- Source-to-source program transformation:
  - Takes the code for a program and some inputs
  - Returns a program which takes the rest of the inputs and finishes the computation.
  - That is, creates a specialized version of the given program
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, ...
Example

• Consider the function:

```ml
fun append(xs : int list, ys : int list) = 
  if (xs=nil) then
    ys
  else
    (hd xs)::append(tl xs,ys)
```

• How would you specialize this code if
  – you knew \( xs = [1,2,3] \) ?
  – you knew \( ys = [1,2,3] \) ?
Implementations

- On-line partial evaluation
  - Generalized interpreter.
  - Environment contains values for some variables
  - For each expression, check whether there's enough information to do the computation
    - If so, return the resulting value
    - Otherwise, return code to compute the value.
Implementations

• Off-line partial evaluation
  – Runs in two stages
    • Prepass ("Binding-Time Analysis")
      – Determines which expressions will depend only on the "early" arguments
      – Annotate the code
    • Postpass
      – Runs through the annotated code
      – Does any computation annotated as static
      – Returns any computation annotated as dynamic
Another Example

```c
void miniprintf(char fmt[], int val[]) {
    int i = 0;
    while( *fmt != '\0' ) {
        if( *fmt != '%' )
            putchar(*fmt);
        else
            switch(*++fmt) {
                case 'd' : putint(val[i++]); break;
                case '%' : putchar('%');     break;
                default  : prterror(*fmt);   break;
            }
        fmt++;
    }
}
```
void miniprintf(char fmt[], int val[]) {
    int i = 0;
    while( *fmt != '\0' ) {
        if( *fmt != '%' )
            putchar(*fmt);
        else
            switch(*++fmt) {
                case 'd': putint(val[i++]); break;
                case '%': putchar('%'); break;
                default: prterror(*fmt); break;
            }
        fmt++;
    }
}
Specialization

• Specialize with `fmt = "<%d, %d>"`

```c
miniprintf_1(int val[])
{
    putchar( '<'    );
    putint ( val[0] );
    putchar( ','    );
    putint ( val[1] );
    putchar( '>'    );
}
```

• Example taken from Tempo system
  
Programs as Data and Behavior

- I will denote program code as $\text{prog}$.
- I 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 $x$.

- I will sometimes write $[\text{prog}]_L$ to emphasize that we are using the semantics of the language $L$ to determine the meaning of the code $\text{prog}$.
Formalizing Partial Evaluation

• Suppose the program $p$ in language $S$ takes two inputs:
  \[
  \llbracket p \rrbracket_S(m, n) = \text{output}
  \]

• If $\text{mix}$ is (code for) a partial evaluator then
  \[
  \llbracket \text{mix} \rrbracket(p, m) = p^m
  \]
  where
  \[
  \llbracket p^m \rrbracket_S(n) = \llbracket p \rrbracket_S(m, n) = \text{output}
  \]

• That is,
  \[
  \llbracket \llbracket \text{mix} \rrbracket(p, m) \rrbracket_S(n) = \llbracket p \rrbracket_S(m, n) = \text{output}
  \]
Interpreters and Compilers

• We say $\text{int}$ is an interpreter for language $S$ written in language $L$ 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 $T$ written in $L$ if

$$\text{output} = [\text{source}]_S(\text{input})$$

$$= [[[\text{comp}]]_L(\text{source})]_T(\text{input})$$

i.e.,

$$[[\text{comp}]]_L(\text{source}) = \text{target}$$

where

$$[\text{source}]_S(\text{input}) = [[\text{target}]]_T(\text{input})$$
Compiling via Partial Evaluation?

- Consider an interpreted program `source` running on many inputs.
  - Runs interpreter many times, one input (`source`) unchanging.
  - Why not use partial evaluation?

```
Put target := [mix](int, source)
then output = [source]_s(input)
    = [int]_L(source, input)
    = [[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 `mix` many times with one 
  input (`int`) unchanging.
- Why not use partial evaluation?

  Put  
  \[
  \text{compiler} := \text{mix}(\text{mix, int})
  \]
  then  
  \[
  \text{target} = \text{mix}(\text{int,source})
  = [[[\text{mix}] (\text{mix, int})]](\text{source})
  = [[\text{compiler}}](\text{source})
  \]
Compiling via Partial Evaluation?

• Suppose we want to compute

\[ [\text{mix}] (\text{mix}, \text{int}) \]

for many different interpreters.
  – That is, we are running \text{mix} many times with one input (\text{mix}) unchanging.
  – Why not use partial evaluation?

Put \text{cogen} := [\text{mix}] (\text{mix}, \text{mix})
then \text{compiler} = [\text{mix}] (\text{mix}, \text{int})
  = [[\text{mix}] (\text{mix}, \text{mix})](\text{int})
  = [\text{cogen}] (\text{int})
Critique of Partial Evaluation

• PE can work very well. But...
  – Tends to be very sensitive to the way a program is written
    • Literature on "binding-time improvements"
  – Still need work on "partially static" inputs
  – Hard to be sufficiently aggressive and still terminate on all inputs

• Partial evaluators exist for Scheme, C, Haskell, ...
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
General Approaches

• Run a compiler at run-time
  – Manipulate abstract syntax trees in program
  – Run the result through a compiler.

  – Advantages:
    • straightforward
    • probably best code generation
  – Disadvantages:
    • Big overhead for code generation
General Approaches

• Use templates
  – Pieces of machine code with "holes" for constants
  – At run-time make copies of the templates as needed while filling in the holes.

  – Advantages:
    • Cheaper to generate code

  – Disadvantages:
    • Permits few code optimizations
General Approaches

• Specialized compilers
  – Create code that directly generates the desired code at run-time.

  – Advantages:
    • Cheap to generate code
    • Permits more (local) optimizations

  – Disadvantages:
    • More complex to think about code that creates machine instructions.
Applications of RT CG

• Matrix multiplication (Fabius)
  – Particularly sparse matrices

• Operating systems
  – Specialize system calls like read for particular cases (Synthetix)
    • e.g., regular file on local disk with 8KB block size)
  – Generate code for packet filters (Fabius, DPF)

• Language implementations
  – JIT compilation