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:

```
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

```
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++;
        }
      }
```

Binding Time Analysis

```
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>"`

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

- Example taken from Tempo system

  `http://www.irisa.fr/compose/tempo/`

Programs as Data and Behavior

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

- I will sometimes write $[\![\texttt{prog}]\!]_{\texttt{L}}$ to emphasize that we are using the semantics of the language `L` to determine the meaning of the code `prog`.

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

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

- That is,

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

Interpreters and Compilers

- We say `int` is an interpreter for language `S` written in language `L` if, given any program `source` (written in `S`), we have

$$\texttt{output} = [\![\texttt{source}]\!]_{\mathrm{S}}(\texttt{input})$$

$$= [\![\texttt{int}]\!]_{\mathrm{L}}(\texttt{source},\texttt{input})$$

- We say `comp` is a compiler from `S` to `T` written in `L` if

$$\texttt{output} = [\![\texttt{source}]\!]_{\mathrm{S}}(\texttt{input})$$

$$= [\![\,[\![\texttt{comp}]\!]_{\mathrm{L}}(\texttt{source})\,]\!]_{\mathrm{T}}(\texttt{input})$$

i.e., $[\![\texttt{comp}]\!]_{\mathrm{L}}(\texttt{source}) = \texttt{target}$

where $[\![\texttt{source}]\!]_{\mathrm{S}}(\texttt{input}) = [\![\texttt{target}]\!]_{\mathrm{T}}(\texttt{input})$

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

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

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

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

  Put $\texttt{compiler} := [\![\texttt{mix}]\!](\texttt{mix},\texttt{int})$

  then

$$\begin{aligned}\texttt{target} &= [\![\texttt{mix}]\!](\texttt{int},\texttt{source})\\ &= [\![\,[\![\texttt{mix}]\!]\ (\texttt{mix},\texttt{int})]\!]\ (\texttt{source})\\ &= [\![\texttt{compiler}]\!](\texttt{source})\end{aligned}$$

Compiling via Partial Evaluation?

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

  Put $\texttt{cogen} := [\![\texttt{mix}]\!]\ (\texttt{mix,mix})$

  then
  $$\begin{aligned}\texttt{compiler} &= [\![\texttt{mix}]\!]\ (\texttt{mix,int})\\ &= [\![\ [\![\texttt{mix}]\!]\ (\texttt{mix,mix})\ ]\!](\texttt{int})\\ &= [\![\texttt{cogen}]\!]\ (\texttt{int})\end{aligned}$$

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 RTCG

- 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