

Partial Evaluation for Haskell

Duncan Coutts

30th September 2004

Here is why partial evaluation is cool

- Take an interpreter for a simple imperative language

```
runProg :: Prog -> Env -> Value
```

- Take a program in that language

```
prog = [While (Var "x" :/=: Const 0)
        ["x" := Var "x" :-: Const 1]
        ,Return "x"]
```

- Partially evaluate the interpreter on the program...

```
$(partialEval runProg) prog
```

The outcome is that we have compiled the program into Haskell

```
let update_0 = \st_1 e_2 -> st_1 // [(0, e_2)]
    lookup_3 = \st_4 -> st_4 ! 0
    evalProg_5 = \initialState_6 -> evalStmts_7 (listArray (0,0) initialState_6)
    evalStmts_7 = \st_8 -> if evalExpr_9 st_8 /= 0
                            then evalStmts_10 st_8
                            else evalStmts_11 st_8
    evalStmts_11 = \st_12 -> evalExpr_13 st_12
    evalStmts_10 = \st_14 -> evalStmts_7 (update_0 st_14 (evalExpr_15 st_14))
    evalExpr_15 = \st_16 -> evalExpr_13 st_16 - evalExpr_17 st_16
    evalExpr_17 = \st_18 -> 1
    evalExpr_9 = \st_19 -> fromEnum $ (evalExpr_13 st_19 /= evalExpr_20 st_19)
    evalExpr_20 = \st_18 -> 0
    evalExpr_13 = \st_21 -> lookup_3 st_21
in evalProg_5
```

What is partial evaluation?

- Partial evaluation is program specialisation
- We take a general program and specialise it to particular values of one (or more) of its arguments. Hopefully the *residual* specialised program is faster than the original.
- A technique for splitting a program into two (or more) run time stages:

The stage 1 program consumes some input parameters and produces a stage 2 program. The stage 2 program consumes the rest of the input parameters and computes the result.

What is partial evaluation good for?

- Making a modular program run faster
- Saving you from having to write complicated program generators
- Generating compilers from interpreters
- Explaining certain kinds of program optimisation (unfolding, inlining, constant propagation)

Why Haskell?

- Partial evaluators exist for other languages
- Types and laziness present new challenges
- The many domain specific languages embedded in Haskell could benefit
- “Haskell is the language of choice for discriminating hackers”
(source: ICFP’04 programming contest)

The formal bit

This is just a simplified restatement of Kleene's S-m-n theorem.

Original general function:

$$[[f]] : S \rightarrow D \rightarrow X$$

The function specialised to some value of its first input:

$$[[f_s]] : D \rightarrow X$$

If it's a correct specialisation, the following had better hold:

$$[[f_s]] d = [[f]] s d \quad \forall s \in S, d \in D$$

The program `mix` calculates f_s from f and s :

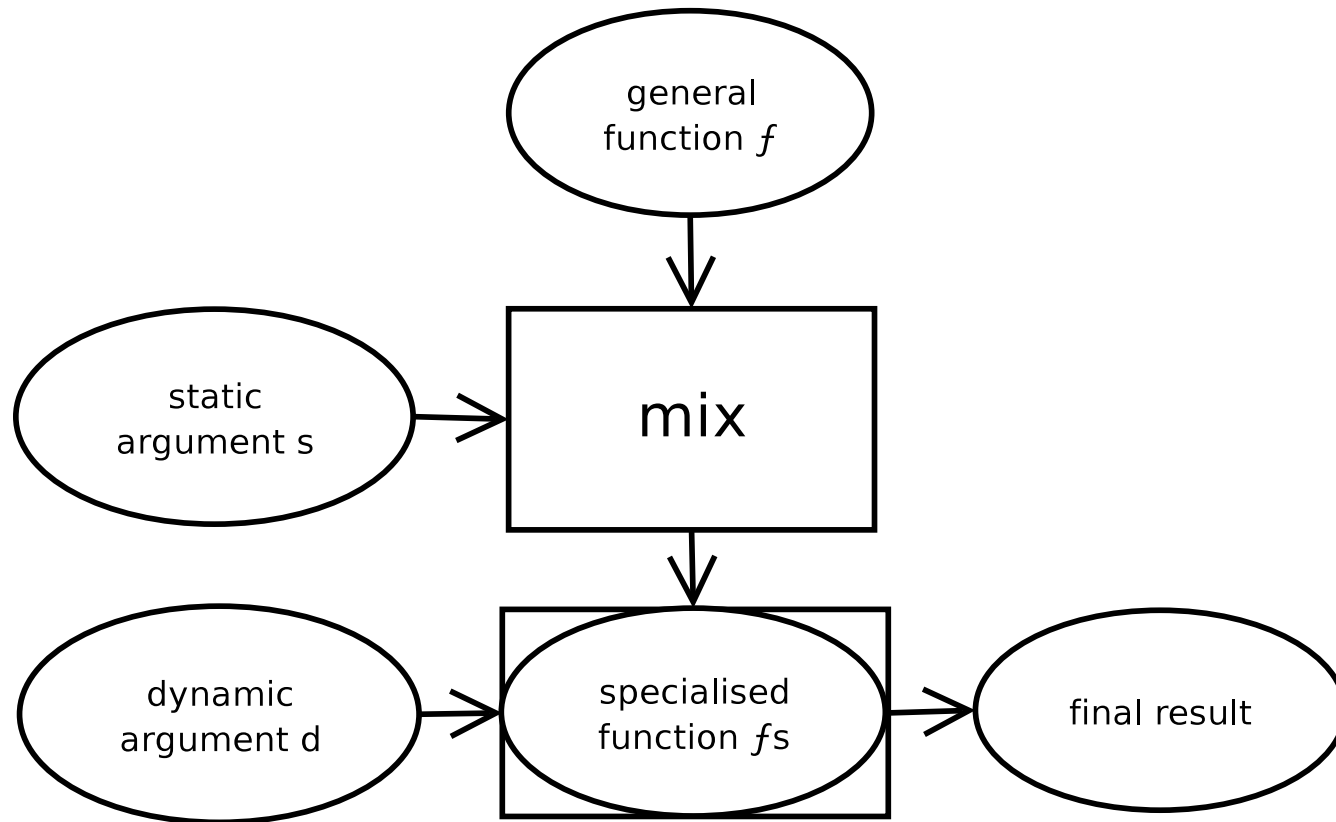
$$[[\text{mix}]] f s = f_s$$

Putting this together we get the mix equation:

$$[[[[\text{mix}]] f s]] d = [[f]] s d \quad \forall s \in S, d \in D$$

The picture of the mix equation

$$\llbracket \llbracket \text{mix} \rrbracket f s \rrbracket d = \llbracket f \rrbracket s d \quad \forall s \in S, d \in D$$



Example: the Ackermann function

```
ack m n
| m == 0      = n+1
| n == 0      = ack (m-1) 1
| otherwise   = ack (m-1) (ack m (n-1))
```

Specialising on the m parameter (e.g. $m = 3$ or 4) amounts to loop unrolling and gives approximately a double speedup.

Binding time annotations

- We indicate which bits of the program to run at compile time by annotating expressions as “static” or “dynamic”
- This can either be automatic or manual

```
ack m =  
  if m == 0 then dynamic(\n -> n+1)  
    else dynamic(\n ->  
      if n == 0 then static(ack (m-1)) 1  
        else static(ack (m-1)) (static(ack m) (n-1))
```

What is the type of mix?

- No obvious answer, but two possibilities

- $\text{mix} :: \text{Exp } (a \rightarrow b) \rightarrow a \rightarrow \text{Exp } b$

This is not feasible in Haskell because we need to do case analysis on the static argument a .

- $\text{mix} :: \text{Exp } (a \rightarrow b) \rightarrow \text{Exp } a \rightarrow \text{Exp } b$

This leads to problems of double encodings which result in huge space and time overheads.

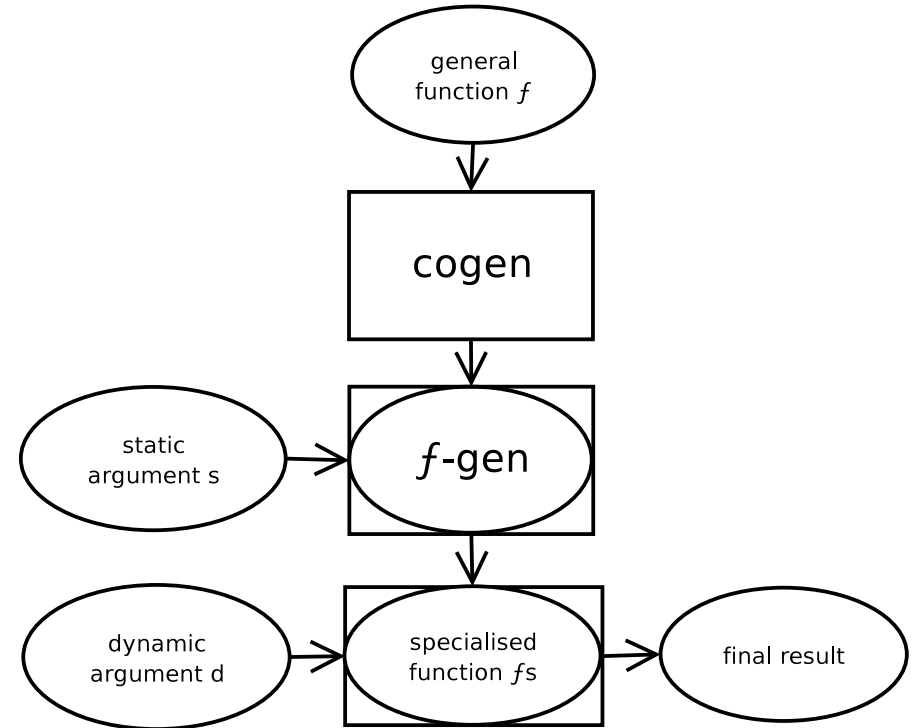
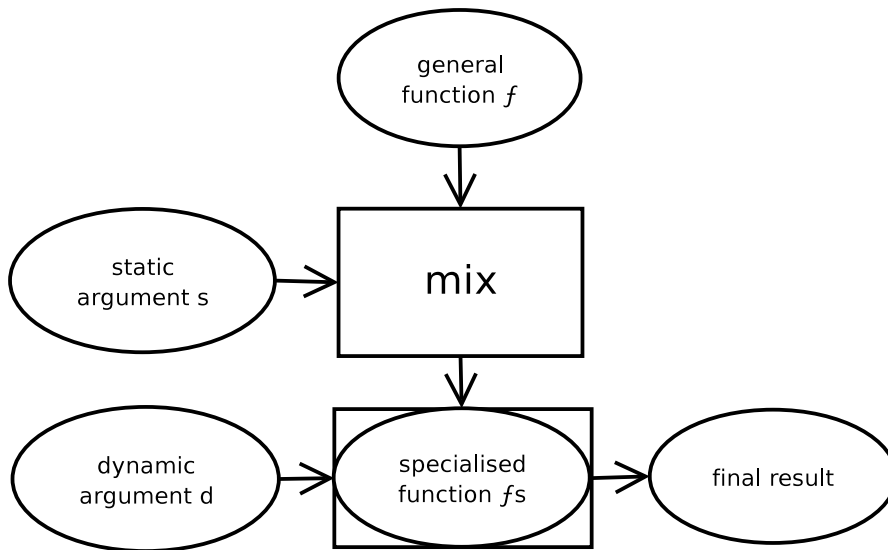
Interpreter style vs. compiler style partial evaluators

- We do not need to implement mix, there is an alternative
- mix is very much like an interpreter
- We can implement a partial evaluator in a compiler style, which (by historical accident) we call cogen
- cogen has a feasible type:
 $\text{cogen} :: \text{Exp } (a \rightarrow b) \rightarrow \text{Exp } (a \rightarrow \text{Exp } b)$
- $\text{cogen} = [|\text{mix}|] \text{mix mix}$

The picture for mix and cogen

$\text{mix} :: \text{Exp } (a \rightarrow b) \rightarrow a \rightarrow \text{Exp } b$
 $\llbracket \llbracket \text{mix} \rrbracket f s \rrbracket d = \llbracket f \rrbracket s d$

$\text{cogen} :: \text{Exp } (a \rightarrow b) \rightarrow \text{Exp } (a \rightarrow \text{Exp } b)$
 $\llbracket \llbracket \llbracket \text{cogen} \rrbracket f \rrbracket s \rrbracket d = \llbracket f \rrbracket s d$



Advantages of the compiler approach

- The type is feasible in Haskell
- No double encoding problems
- Can re-use an existing compiler (where as for `mix` we must embed an interpreter)
- A simple version can be implemented in approximately 100 lines of code using Template Haskell

A quick introduction to Template Haskell

- Template Haskell is a compile time meta programming language for Haskell
- Two extra language features `$(...)` and `[| ... |]`
- Allows us to write staged programs

```
ack m =  
  if m == 0 then [| \n -> n+1 |]  
  else [| \n ->  
    if n == 0 then $(ack (m-1)) 1  
    else $(ack (m-1)) ($(ack m) (n-1) |]
```

Template Haskell does not solve all the problems

- The previous example produces an infinite stage 2 program
- Correct staging is not easy and requires changing the program

```
ack m =
  if m == 0 then [| \n -> n+1 |]
  else [| let ack_m = \n ->
          if n == 0 then $(ack (m-1)) 1
          else $(ack (m-1)) (ack_m (n-1))
        in ack_m |]
```

Template Haskell does provide a good infrastructure

- Provides parser, pretty printer, abstract syntax
- `cogen` produces Template Haskell programs
- Allows partial evaluator to be implemented as a library rather than an external preprocessor

```
compiler = $(cogen [| interpreter |])  
program  = $(compiler progAST)
```

How does cogen work?

- In general partial evaluation works by duplicating chunks of code for each set of values of their *free static variables*
- Each chunk of code is run (possibly requiring more chunks to be specialised)
- All the chunks are put together in the residual program and in general they may be mutually recursive
- Cogen works by producing a program that performs this

Result of specialising the Ackerman function for $m = 3$

```
let ack_3 = \n -> if n == 0
                  then ack_2 1
                  else ack_2 (ack_3 (n - 1))
    ack_2 = \n -> if n == 0
                  then ack_1 1
                  else ack_1 (ack_2 (n - 1))
    ack_1 = \n -> if n == 0
                  then ack_0 1
                  else ack_0 (ack_1 (n - 1))
    ack_0 = \n -> n + 1
in ack_3
```

Something unsurprising

- We already write staged programs, e.g.

```
matchRegex :: String -> String -> Bool
matchRegex spec text = match (compile spec) text
```

```
compile :: String -> Regex
match :: Regex -> String -> Bool
```

- These sorts of programs specialise trivially

Future directions

- Improve support for partially static data
- Add support for semi-automatically inferring binding time annotations
- Extend to work for programs that manipulate higher-order static data
- Have a more precise story about laziness
- Apply the tool to Haskell EDSLs with the aim of demonstrating easier performance improvements than other techniques

Summary

- Partial evaluation is a useful and unifying technique
- Partial evaluation should be much easier than using other generative programming techniques
- The major issue is the lack of good tools - especially for typed languages

Questions?