Partial Evaluation

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Objectives

You should be able to...

- Explain the difference between Interpreters and Compilers mathematically
- Annotate a program according to the expression binding times
- Explain the difference between online and offline partial evaluation
- Specialize a simple program according to its static input
- Describe the three Futamura projections

An Interpreter

Notations

- Let $\mathcal{S}$ be a language.
- Let M be a program in language $\mathcal{S}$.
- Let lower case letters be values in $\mathcal{S}$.
- An $\mathcal{S}$-interpreter is a program I such that

$$I(M, s, d) \to x$$

- An $\mathcal{S}$-partial evaluator is a program

$$P(M, s) \to M_s$$

such that

$$M_s(d) = M(s, d)$$

Some examples

$$P(\texttt{printf}, \texttt{"\%s"}) \to \texttt{puts}$$

$$P(\texttt{pow(n,x)}, 2) \to \lambda\texttt{x . x * x}$$

$$P()$$

Basic Operation

Online

- Like `eval`, but distinguishes between “known” and “unknown” values.
- Expressions that have all known sub-expressions are *specialized*.
- Everything else is *residualized*.
- More aggressive, but can cause instability.

Offline

- A preprocessor called a *binding time analyser* annotates the source program.
  - Everything that is known for sure is marked as known.
  - Everything else is marked as unknown.
- The partial evaluator then follows the annotations.
- Can lose opportunity to specializes, but more stability.

BTA Example

- We underline the things that are known.
- We start with the input `n`.
- We annotate the “leaves”
- If all subexpressions are known, so is the expression.
- The parial evaluator will compute anything that’s underlined.
- It will unroll functions that the inputs are partially known.

```
pow n x =
  if n > 0
    then x * pow (n-1) x
    else 1
```

BTA Example

- We underline the things that are known.
- We start with the input `n`.
- We annotate the “leaves”
- If all subexpressions are known, so is the expression.
- The parial evaluator will compute anything that’s underlined.
- It will unroll functions that the inputs are partially known.

```
pow n x =
  if n > 0
    then x * pow (n-1) x
    else 1
```

BTA Example

- We underline the things that are known.
- We start with the input `n`.
- We annotate the “leaves”
- If all subexpressions are known, so is the expression.
- The parial evaluator will compute anything that’s underlined.
- It will unroll functions that the inputs are partially known.

```
pow n x =
  if n >0
    then x * pow (n-1) x
    else 1
```

BTA Example

- We underline the things that are known.
- We start with the input `n`.
- We annotate the “leaves”
- If all subexpressions are known, so is the expression.
- The parial evaluator will compute anything that’s underlined.
- It will unroll functions that the inputs are partially known.

```
pow n x =
   if n >0
     then x * pow (n-1) x
     else 1
```

Binding Time Analyzer

```
data AnnExp = AIntExp _
            | AVarExp String Bool
            | AOpExp String AnnExp AnnExp
     ...
bta :: Exp -> BEnv -> AnnExp
bta (IntExp i) env = IntExp i
bta (VarExp s) env = AVarExp s bt
  where bt = case H.lookup s env of
               Just b -> b
               Nothing -> False
bta (OpExp e1 e2) env =
  let ae1 = bta e1 env
      ae2 = bta e2 env
   in AOpExp ae1 ae2 (isKnown ae1 && isKnown ae2)
```

The First Futamura Projection

$$P(I,S) \mapsto I_S$$
where $I_S(D) = I(S,D)$

Compilation

- We have fed an interpreter to our parial evaluator.
- The result is I_S... this is a compiled program!
- I_S usually runs 4–10 times faster than $I(S,P)$.

The Second Futamura Projection

$$P(P,I) \mapsto P_I$$
where $P_I(S) = P(I,S)$
and $P(I,S)(D) = I_S(D) = I(S,D)$

Producing a Compiler

- Notice what P_I actually does.
- We wrote an interpeter, and got a compiler...
- ... for *free*.

The Third Futamura Projection

$P(P,P) \mapsto P_P$
where $P_P(I) = P(P,I)$
and $P(P,I)(S) = P_I(S) = P(I,S)$
and $P(I,S)(D) = I_S(D) = I(S,D)$

Compiler Generator

- Well, maybe not entirely free. It costs something to run $P(P,I)$.
- But, we can specialize P to run these, so that P_P is faster.
- This is called a *code generator* or *compiler generator*.