Solving Data-flow Problems in Syntax Trees

Sebastian Graf

September 9, 2017

Karlsruhe Institute of Technology

Introduction

- My master's thesis[1]: Call Arity vs. Demand Analysis
  - Result: Usage Analysis generalising Call Arity
  - Precision of Call Arity without co-call graphs
- Requirements led to complex analysis order
- *Specification* of data-flow problem decoupled from its *solution*

[1] https://pp.ipd.kit.edu/uploads/publikationen/graf17masterarbeit.pdf

Strictness Analysis

- Provides lower bounds on *evaluation cardinality*
- Which variables are evaluated at least once?
  - *S* Strict (Yes!)
  - *L* Lazy (Not sure)
- Enables call-by-value, unboxing

```haskell
main = do
  let  x = ... -- S
  let  y = ... -- S
  let  z = ... -- L
  print (x + if odd y then y else z)
```

Strictness Analysis

- Provides lower bounds on *evaluation cardinality*
- Which variables are evaluated at least once?
  - *S* Strict (Yes!)
  - *L* Lazy (Not sure)
- Enables call-by-value, unboxing

```
main = do
  let !x = ... -- S
  let !y = ... -- S
  let  z = ... -- L
  print (x + if odd y then y else z)
```

GHC's Demand Analyser

- Performs strictness analysis (among other things)
- Fuels Worker/Wrapper transformation
- Backward analysis
    - Which strictness does an expression place on its free variables?
    - Which strictness does a function place its arguments?
- *Strictness type*: $\mathsf{StrType} = \langle \mathsf{FVs} \rightharpoonup \mathsf{Str}, \mathsf{Str}^* \rangle$

Strictness Signatures

- Looks at the right-hand side of `const` before the `let` body!
- *Unleashes* strictness type of `const`'s RHS at call sites

```
let const a b = a -- const :: ⟨[], [S, L]⟩
in const
     y              -- S
     (fac 1000)     -- L
```

Call Context Matters

- Whole expression is strict in `z`
- Only digests `f` for manifest arity 1, can't look under lambda
- `f` is called with 2 arguments

```
let f x = -- f :: ⟨[z ↦ L], [S]⟩
        if odd x
          then \y -> y*z
          else \y -> y+z
in f 1 2
```

Call Context Matters

- Whole expression is strict in `z`
- Only digests `f` for manifest arity 1, can't look under lambda
- `f` is called with 2 arguments

```
let f x = -- f :: ⟨[z ↦ L], [S]⟩
      if odd x
        then \y -> y*z
        else \y -> y+z
in seq (f 1) 42
```

Call Context Matters

- Whole expression is strict in `z`
- Only digests `f` for manifest arity 1, can't look under lambda
- `f` is called with 2 arguments

```
let f x = -- f :: ⟨[z ↦ L], [S]⟩
      if odd x
        then \y -> y*z
        else \y -> y+z
in f 1 2
```

Call Context Matters

- Solution: Analyse RHS when incoming arity is known
- Formally: Finite approximation of *strictness transformer*
  - $\mathrm{StrTrans} = \mathbb{N} \to \mathrm{StrType}$
- Exploit laziness to memoise results?

```
let f x = -- f₁ :: ⟨[z ↦ L], [S]⟩
      if odd x
        then \y -> y*z
        else \y -> y+z
in f 1 2
```

Call Context Matters

- Solution: Analyse RHS when incoming arity is known
- Formally: Finite approximation of *strictness transformer*
  - $\text{StrTrans} = \mathbb{N} \to \text{StrType}$
- Exploit laziness to memoise results?

```
let f x = -- f_2 :: ⟨[z ↦ S], [S, S]⟩
      if odd x
        then \y -> y*z
        else \y -> y+z
in f 1 2
```

Recursion

- Exploit laziness to memoise approximations?
- ✗ Recursion leads to termination problems
- Rediscovered fixed-point iteration, detached from the syntax tree
- Leads to data-flow network, solved by worklist algorithm

```
let fac n =
      if n == 0
        then 1
        else n * fac (n-1)
in fac 12
```

Example

- Allocate one top-level node + one node per `let` binding

```
let even 0 = True
    even n = odd (n-1)
    odd 0 = False
    odd n = even (n-1)
in even 12
```

$\Rightarrow$

End