Fixing 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*
- Is this variable evaluated at least once?
  - *Strictness*: Str ::= *S* | *L*
  - *S*trict (Yes!)
  - *L*azy (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)
```

Strictness Analysis

- Provides lower bounds on *evaluation cardinality*
- Is this variable evaluated at least once?
  - *Strictness*: Str ::= $S \mid L$
  - *S*trict (Yes!)
  - *L*azy (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 on its arguments?
- *Strictness type*: $\mathsf{StrType} = \langle \mathsf{FVs} \rightharpoonup \mathsf{Str}, \mathsf{Str}^* \rangle$

- 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
    (error "💣") -- L
```

- 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
```

- 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*
  - StrTrans $= \mathbb{N} \to$ 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
```

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

```
let f x = -- f₂ :: ⟨[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 problem, solved by worklist algorithm

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

Data-flow Graph for Strictness Analysis

- Allocate nodes to break recursion
  - One top-level node
  - One node per pair of (let binding, incoming arity)
- Initialise worklist to top-level node
- Initialise nodes with $\bot$

```
let f 0 = const 0
    f 1 = id
    f n = f (n `mod` 2)
in f x y
```
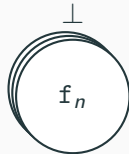
$\Longrightarrow$

$(\texttt{<root>}_0)$

Data-flow Graph for Strictness Analysis

- Allocate nodes to break recursion
  - One top-level node
  - One node per pair of (let binding, incoming arity)
- Initialise worklist to top-level node
- Initialise nodes with $\perp$

```
let f 0 = const 0
    f 1 = id
    f n = f (n `mod` 2)
in f x y
```
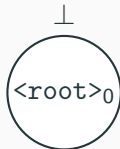
$\implies$

$\langle\text{root}\rangle_0$

f_n

- Allocate nodes to break recursion
  - One top-level node
  - One node per pair of (let binding, incoming arity)
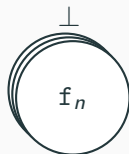- Initialise worklist to top-level node
- Initialise nodes with $\bot$

```
let f 0 = const 0
    f 1 = id
    f n = f (n `mod` 2)
in f x y
```

$\Longrightarrow$

$<\text{root}>_0$

f_n

Worklist: $\{<\text{root}>_0\}$

7

Data-flow Graph for Strictness Analysis

- Allocate nodes to break recursion
  - One top-level node
  - One node per pair of (let binding, incoming arity)
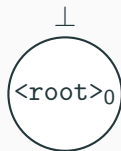- Initialise worklist to top-level node
- Initialise nodes with $\bot$

```
let f 0 = const 0
    f 1 = id
    f n = f (n 'mod' 2)
in f x y
```

$\implies$

Worklist: $\{\texttt{<root>}_0\}$

```
let f 0 = const 0
    f 1 = id
    f n = f (n 'mod' 2)
in f x y
```

$\Longrightarrow$

Worklist: $\{$<root>$_0\}$

```
let f 0 = const 0
    f 1 = id
    f n = f (n 'mod' 2)
in f x y
```

$\Longrightarrow$

Worklist: $\{\}$

```
let f 0 = const 0
    f 1 = id
    f n = f (n `mod` 2)
in f x y
```

$\Longrightarrow$

Worklist: $\{\}$

```
let f 0 = const 0
    f 1 = id
    f n = f (n 'mod' 2)
in f x y
```

$\Longrightarrow$

Worklist: $\{\}$

```
let f 0 = const 0
    f 1 = id
    f n = f (n 'mod' 2)
in f x y
```

$\Longrightarrow$

$\bot$

$\bot = \langle [], [S, S] \rangle$

<root>$_0$ ⟶ f_2

Worklist: $\{\}$

```
let f 0 = const 0
    f 1 = id
    f n = f (n 'mod' 2)
in f x y
```

$\Longrightarrow$

$\perp$

$\perp = \langle [], [S, S] \rangle$

$\texttt{<root>}_0 \longrightarrow f_2$

Worklist: $\{f_2\}$

```
let f 0 = const 0
    f 1 = id
    f n = f (n ‘mod‘ 2)
in f x y
```

$\Longrightarrow$

Worklist: $\{f_2\}$

```
let f 0 = const 0
    f 1 = id
    f n = f (n 'mod' 2)
in f x y
```

$\Longrightarrow$

$\langle\, [x \mapsto S, y \mapsto L]\,, []\,\rangle \qquad \langle [], [S, L]\rangle$

Worklist: $\{f_2\}$

Data-flow Graph for Strictness Analysis

```
let f 0 = const 0
    f 1 = id
    f n = f (n 'mod' 2)
in f x y
```

$\Longrightarrow$

$\langle\, [x \mapsto S, y \mapsto L]\,, []\,\rangle$ $\langle [], [S, L]\rangle$

$\langle\text{root}\rangle_0 \longrightarrow f_2$

Worklist: $\{f_2\}$

```
let f 0 = const 0
    f 1 = id
    f n = f (n `mod` 2)
in f x y
```

$\Longrightarrow$

$\langle\, [x \mapsto S, y \mapsto L]\, , []\, \rangle \qquad \langle [], [S, L] \rangle$

Worklist: $\{\}$

```
let f 0 = const 0
    f 1 = id
    f n = f (n ‘mod‘ 2)
in f x y
```

$\Longrightarrow$

$\langle\, [x \mapsto S, y \mapsto L]\,, []\,\rangle$ $\langle [], [S, L]\rangle$

$\langle\text{root}\rangle_0$ → f_2

Worklist: $\{\}$

```
let f 0 = const 0
    f 1 = id
    f n = f (n 'mod' 2)
in f x y
```

$\Longrightarrow$

$\langle\, [x \mapsto S, y \mapsto L]\,, []\rangle \quad \langle [], [S, L]\rangle$

$\langle\text{root}\rangle_0 \longrightarrow f_2$

Worklist: $\{\}$

```
let f 0 = const 0
    f 1 = id
    f n = f (n 'mod' 2)
in f x y
```

$\Longrightarrow$

$\langle\, [\mathrm{x} \mapsto S, \mathrm{y} \mapsto L]\,, [] \rangle \qquad \langle [], [S, L] \rangle$

$\langle \text{root} \rangle_0 \longrightarrow f_2$
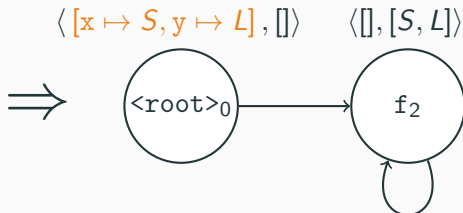
Worklist: $\{\}$

Implementation

- Hide iteration strategy behind `TransferFunction` monad
- Data-flow nodes `k`, denoting lattice `v`
- Single 'impure' primitive `dependOn`

```haskell
data TransferFunction k v a
instance Monad (TransferFunction k v)

dependOn
  :: Ord k
  => k
  -> TransferFunction k v (Maybe v)
```

Implementation

- `DataFlowProblem` assigns `TransferFunction` and `ChangeDetector` to nodes

```haskell
type ChangeDetector k v
  = k -> v -> v -> Bool

data DataFlowProblem k v
  = DFP
  { transfer       :: k -> TransferFunction k v v
  , detectChanges :: k -> ChangeDetector k v
  }
```

- `fixProblem` solves data-flow problems
- Specification as `DataFlowProblem`
- Implements fixed-point iteration strategy
  - Can use worklist algorithm, starting from a specified root set

```
fixProblem
  :: Ord k
  => DataFlowProblem k v
  -> Set k
  -> Map k v
```

Applied to Strictness Analysis

- Denote expressions by their strictness transformer
- Model points of strictness transformer separately
- Instantiate as
  `DataFlowProblem (ExprNode, Arity) StrType`
- `ExprNode`: Totally ordered, allocated as needed
  - Dictates priority in worklist
  - Performance depends on suitable priorities

Comparison to `hoopl`

- `hoopl` (Ramsey et al. 2010) works on CFGs
  - Our data-flow graph is much less restrictive
  - Edges implicit in DSL
- Designed for imperative languages
- 'Operational' rather than 'denotational'
  - Analysis along control-flow rather than data-flow
- Makes (join-semi)lattice explicit
  - TODO
- Also includes a solution for transformations

Discussion

- ✓ Decouple analysis logic from iteration logic by a graph-based approach
- ✗ Coupling not as painful as it would be in imperative programs
- ✓ Still obscures intent, even obstructs ideas
- ✓ 'Hacks' such as caching of analysis results as in Peyton Jones et al. (2006, §9.2) between iterations for free
- ✗ Unclear how performance is affected
- ✗ Can only shine if shared concerns are actually extracted from a number of analyses

Conclusion

- Pitched an interesting idea that came out of my thesis
- Separate *specification* of the data-flow problem from computing its *solution*
- Graph-based abstraction, single solver
- Future Work:
    1. Monotone maps with partially-ordered keys[2]
    2. Polish API, make a package
    3. Testdrive and measure it in GHC

---

[2] https://github.com/sgraf812/pomaps/

Bibliography

📄 Peyton Jones, Simon, Peter Sestoft, and John Hughes (2006).
   Demand Analysis. URL: https://www.microsoft.com/en-
   us/research/publication/demand-analysis/.

📄 Ramsey, Norman, João Dias, and Simon Peyton Jones (2010).
   "Hoopl: A Modular, Reusable Library for Dataflow Analysis and
   Transformation". In: *Proceedings of the Third ACM Haskell
   Symposium on Haskell*. Haskell '10. Baltimore, Maryland, USA:
   ACM, pp. 121–134. ISBN: 978-1-4503-0252-4. DOI:
   10.1145/1863523.1863539. URL:
   http://doi.acm.org/10.1145/1863523.1863539.

Backup

Example

```
let f 0 = const 0
    f 1 = id
    f n =
      const (f (n `mod` 2) 4)
in f x y
```

$\Longrightarrow$

(<root>_0) (f_n)

Example

```
let f 0 = const 0
    f 1 = id
    f n =
      const (f (n 'mod' 2) 4)
in seq (f x) (f x y)
```

$\implies$ (<root>_0) $\quad$ (f_n)

```
let f 0 = const 0
    f 1 = id
    f n =
      const (f (n 'mod' 2) 4)
in seq (f x) (f x y)
```

$\Longrightarrow$
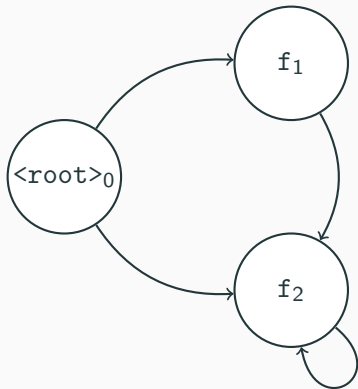
- TransferFunction is a State monad around WorklistState

```
data TransferFunction node lattice a
  = TFM (State (WorklistState node lattice ) a)
  deriving (Functor, Applicative, Monad)
```

Threading annotated expressions

- Annotated `CoreExpr`s are the reason why we do this!
- Thread it through all nodes:
  `DataFlowProblem (ExprNode, Arity) (StrType, CoreExpr)`
- Complicates change detection
  - Expressions follow AST structure
  - Possibly change when strictness type did not
  - `ChangeDetector` has to check set of changed dependencies
- Str $::= S \mid L$ not enough for annotating functions
  - Str $::= S^n \mid L$ with arity $n \in \mathbb{N}$
  - '`f` was called at least once, with at least n arguments'
- ... Or do it as the Demand Analyser does: Assume manifest arity for annotation
  - Be careful not to inline unsaturated wrappers!

```
let f x =
    let g y =
        if odd y
          then g (y - 1)
          else x
    in if even x
        then g x
        else f (3*x + 1)
in f 7
```