

# **A more flexible code generator for GHC**

**João Dias**

**(joint work with Norman Ramsey and Simon Peyton Jones)**

**dias@cs.tufts.edu**

**Department of Computer Science  
Tufts University**

# GHC 6.10's code gen is inflexible

**Hard to**

- **change compiler internals**
- **generate better code**

# Adding flexibility creates opportunities

## Opportunities to:

- change compiler internals
- generate better code

# Adding flexibility creates opportunities

## Opportunities to:

- change compiler internals
- generate better code

## Tried-and-true techniques:

- new IR
- compositional

# GHC's code gen bridges a great gap

Core ( $\lambda$ -calc.)



LIR



asm

# GHC's code gen bridges a great gap

**Core ( $\lambda$ -calc.)**



**LIR**



**asm**

**Core:**

**Nested functions**

**Local variables**

**Implicit stack**

**Implicit heap**

# GHC's code gen bridges a great gap

Core ( $\lambda$ -calc.)



LIR



asm

Core:

Nested functions

Local variables

Implicit stack

Implicit heap

LIR:

~~Nested Functions~~

Local variables, registers

Low-level imperative instructions

~~Call and return~~ Tail calls only

Explicit HP, heap addresses

Explicit SP, stack addresses

GC info

Fixed set of local variables

# **LIR invariants are a straitjacket**

## **Fixed set of local variables**

- **hamstrings optimization**

## **Explicit stack and tail calls only:**

- **complicates translation from Core**
- **makes hand-written run-time code ugly**

# **LIR invariants are a straitjacket**

## **Fixed set of local variables**

- **hamstrings optimization**

## **Explicit stack and tail calls only:**

- **complicates translation from Core**
- **makes hand-written run-time code ugly**

## **Need IR with:**

- **unlimited set of local variables**
- **implicit stack**
- **calls**

# New code gen provides helpful invariants

**Core**



**CFG**



**CFG**



**CFG**



**CFG**



**LIR**



**Asm**

# New code gen provides helpful invariants

**Core**



**CFG**



**CFG**



**CFG**



**CFG**



**LIR**



**Asm**

**optimize**

- Top-level functions**
- Local variables, registers**
- Low-level imperative instructions**
- Call and return**
- Explicit HP, heap addresses**
- Unlimited set of local variables**

# New code gen provides helpful invariants

Core



CFG



CFG



CFG



CFG



LIR



Asm

optimize

Top-level functions

Local variables, registers

Low-level imperative instructions

Call and return

Explicit HP, heap addresses

Unlimited set of local variables

Explicit SP, stack addresses

Fixed set of local variables

# New code gen provides helpful invariants

Core



CFG



CFG



CFG



CFG



LIR



Asm

optimize

Top-level functions

Local variables, registers

Low-level imperative instructions

Call and return

Explicit HP, heap addresses

Unlimited set of local variables

Explicit SP, stack addresses

Fixed set of local variables

~~Call and return~~ Tail calls only

# New code gen provides helpful invariants

Core



CFG



CFG



CFG



CFG



LIR



Asm

optimize

Top-level functions  
Local variables, registers  
Low-level imperative instructions  
Call and return  
Explicit HP, heap addresses  
Unlimited set of local variables

Explicit SP, stack addresses  
Fixed set of local variables

~~Call and return~~ Tail calls only

GC info

# Dataflow optimization is our big hammer

## Dataflow optimization

- analysis computes dataflow facts on edges
- transformation rewrites graph

## Dataflow engine makes optimization easy

- used throughout code generator

# HOOPL makes dataflow easy

You define:

- **lattice** (fact type,  $\perp$ ,  $\sqcup$ ,  $\sqcap$ )
- **transfer functions** relate dataflow facts
- **rewrite functions** replace graph nodes

# HOOPL makes dataflow easy

You define:

- lattice (fact type,  $\perp$ ,  $\sqcup$ ,  $\sqcap$ )
- transfer functions relate dataflow facts
- rewrite functions replace graph nodes

We:

- set up and solve recursion equations
- rewrite graph where possible

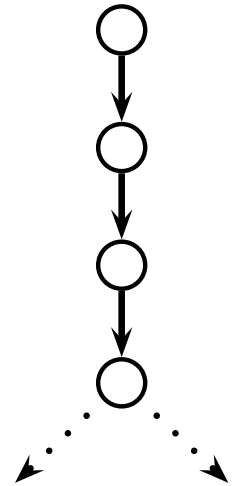
# CFG is purely applicative

Control-flow graph based on Huet's zipper  
(Ramsey and Dias 2005)

Speculative transformation is trivial

Basic block:

- First node (label)
- Middle nodes (assignments)
- Last node (control-transfers)



# Example: Liveness analysis

## Simple lattice: sets of live variables

```
type Live = VarSet
```

```
bottom = emptyVarSet
```

```
join    = unionVarSets
```

```
changed new old = size old < size new
```

# Middle nodes fold over uses, defs

Middle nodes:



```
middleLiveness :: Middle -> Live -> Live  
middleLiveness m = addUsed m . remDefd m
```

# Middle nodes fold over uses, defs

Middle nodes:



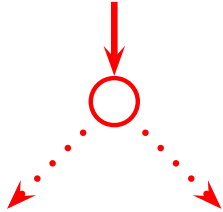
```
middleLiveness :: Middle -> Live -> Live
middleLiveness m = addUsed m . remDefd m
```

Implementation folds over variables:

```
addUsed :: UserOfLocalVars    a => a -> Live -> Live
remDefd :: DefinerOfLocalVars a => a -> Live -> Live
addUsed a live = foldVarsUsed extendVarSet live a
remDefd a live = foldVarsDefd delFromVarSet live a
```

# Last nodes get live sets from labels

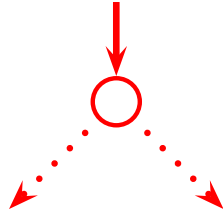
Last nodes:



```
lastLiveness :: Last -> (Label -> Live) -> Live
lastLiveness l = addUsed l . remDefd l . lastLiveOut l
```

# Last nodes get live sets from labels

Last nodes:



```
lastLiveness :: Last -> (Label -> Live) -> Live
lastLiveness l = addUsed l . remDefd l . lastLiveOut l

lastLiveOut :: Last -> (Label -> Live) -> Live
lastLiveOut l env = last l
  where
    last (Branch succ)           = env succ
    last (CondBranch _ t f)     = unionVarSets (env t)
                                     (env f)
    last (Switch _ tbl)        = unionManyVarSets $
                                     map env (catMaybes tbl)
    last (Call { })            = emptyVarSet
```

# Running analysis takes 1 call

## Running the liveness analysis:

```
liveness g = zdfFpFacts soln
  where soln = zdfSolveBwd "liveness" liveLattice
                liveTransfers g
```

# Optimization = Analysis + Rewrite

## Assignments to dead variables become empty graphs:

```
deadRewrites = BackwardRewrites nothing
              middleRemoveDeads nothing

where
  nothing _ _ = Nothing
  middleRemoveDeads (Assign x _) live
    | not (x `elemVarSet` live) = Just emptyGraph
  middleRemoveDeads _ _ = Nothing
```

# Optimization = Analysis + Rewrite

## Running the optimization:

```
removeDeadAssignments g = zdfFpContents result
  where result = zdfRewriteBwd "dead-assignment elim"
                    liveLattice liveTransfers
                    deadRewrites g
```

# Now, we have opportunities

## Machine-independent, low-level optimization

- simple optimizations in place

## More opportunities abound:

- optimizations: constant folding, PRE, etc.
- apply register allocator to entire procedure

Ample targets for hackathons!