Graph Reduction Intermediate Notation

A compiler backend for lazy functional languages

Péter Podlovics

Faculty of Informatics MSc
Eötvös Loránd University

Bolyai informatics seminar, 2018

Overview

Why functional?

- Declarativeness

  pro: can program on a higher abstraction level

- Composability

  pro: can easily piece together smaller programs

  con: results in a lot of function calls

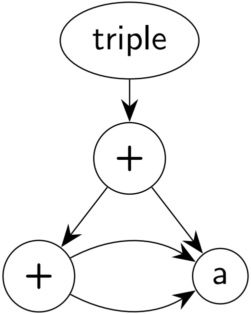
- Functions are first class citizens

  pro: higher order functions

  con: unknown function calls

Implementation of functional languages

Graph reduction



Laziness

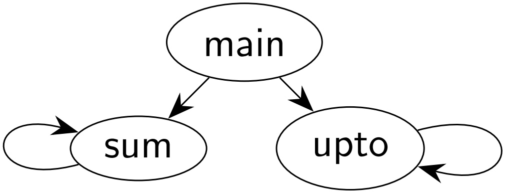
```
sum 0 [1,2,3]
sum (0 + 1) [2,3]
sum ((0 + 1) + 2) [3]
sum (((0 + 1) + 2) + 3) []
((0 + 1) + 2) + 3
(1 + 2) + 3
3 + 3
6
```

A small functional program

```
main = sum (upto 0 10)

upto from to
  | from > to = []
  | otherwise = from : upto (from+1) to

sum []     = 0
sum (x:xs) = x + sum xs
```

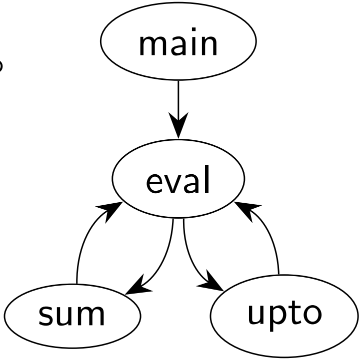
Strict control flow

```
main = sum (upto 0 10)

upto m n to
  | m > n = []
  | otherwise = m : upto (m+1) to

sum []     = 0
sum (x:xs) = x + sum xs
```



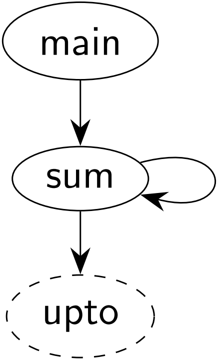
Lazy control flow

```
main = sum (upto 0 10)

upto m n to
  | m > n = []
  | otherwise = m : upto (m+1) to

sum []     = 0
sum (x:xs) = x + sum xs
```



Optimized lazy control flow

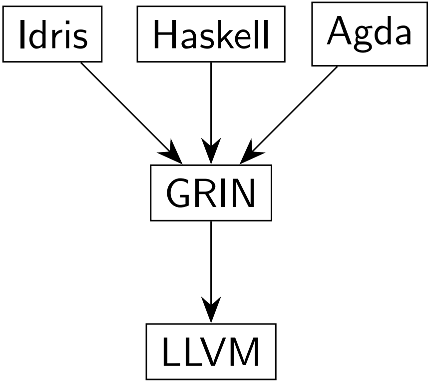
```
main = sum (upto 0 10)

upto m n to
  | m > n = []
  | otherwise = m : upto (m+1) to

sum []     = 0
sum (x:xs) = x + sum xs
```



Goals

- We need to handle laziness
- We need to optimize across functions
- Accomplish both of these for all functional languages

Graph Reduction Intermediate Notation



Properties

- Designed for the computer
- Simple syntax, and semantics
- Untyped, but we use a typed version (for LLVM)
- First order language
- Monadic structure
- Singe Static Assignment property
- Explicit laziness
- Global `eval` (generated)
- No unknown function calls

Syntax

```
grinMain =
  n <- pure (CTwo 0 1)
  p <- store n
  x <- fetch p
  (CTwo a b) <- pure x
  pure a
```

```
eval q =
  v <- fetch q
  case v of
    (CInt x'1)    -> pure v
    (CNil)        -> pure v
    (CCons y ys)  -> pure v
    (Fupto a b) ->
      w <- upto a b
      update q w
      pure w
    (Fsum c) ->
      z <- sum c
      update q z
      pure z
```

Semantics

- C, F, P nodes
- Only basic values and pointers can be in nodes
- Functions cannot return pointers
  - More register usage is exposed
  - The caller can decide whether the return value should be put onto the heap
- `store`, `fetch`, `update`
- Control flow can only diverge and merge at case expressions

Laziness in GRIN

```
upto m n =
  (CInt m') <- eval m
  (CInt n') <- eval n
  b' <- _prim_int_gt m' n'
  if b' then
    pure (CNil)
  else
    m1' <- _prim_int_add m' 1
    m1  <- store (CInt m1')
    p   <- store (Fupto m1 n)
    pure (CCons m p)
```

Eval inlining

- `eval` is an ordinary GRIN function
- It is generated for each individual program
- It enumerates all possible node patterns
- Inlining it results in an enormous code explosion
- It is also quite wasteful
- ... why do we inline it then?

Sparse case optimization

```
<m0>
v <- eval l
case v of
  CNil        -> <m1>
  CCons x xs  -> <m2>
```

$\xRightarrow{v \in \{\text{CCons}\}}$

```
<m0>
v <- eval l
case v of
  CCons x xs -> <m2>
```

Producer name introduction

```
(CInt k) <- pure (CInt 5)

case (CNode x y) of
  <alternatives>

p <- store (CWord 128)
```

$\xRightarrow{\text{for all}}$

```
n0 <- pure (CInt 5)
(CInt k) <- pure n0

n1 <- pure (CNode x y)
case n1 of
  <alternatives>

n2 <- pure (CWord 128)
p <- store n2
```

Dead data elimination

```
<m0>
n <- pure (CPair a b)
(CPair x y) <- pure n
<m1>
```

$\xrightarrow{\text{x is dead}}$

```
<m0>
n <- pure (CPair b)
(CPair y) <- pure n
<m1>
```

Analysis types

- Whole program analysis

  The entire program is subject to the analysis

- Interprocedural program analysis

  The analysis is performed across functions

- Context insensitive program analysis
  - Information is not propagated back to the call site

Compiled data flow analysis

- Analyzing the syntax tree has an interpretation overhead
- We can work around this by "compiling" our analysis into and executable program
- The compiled abstract program is independent of the AST
- It can be executed in a different context (ie.: another program or GPU)
- After run (iteratively), it produces the result of the given analysis

Heap-points-to

```
grinMain =
  a0 <- pure 5
  n0 <- pure (CNil)
  p0 <- store n0
  n1 <- pure (CCons a0 p0)
  r  <- case n1 of
    (CNil) ->
      pure (CNil)
    (CCons x xs) ->
      xs' <- fetch xs
      pure xs'
  pure r
```

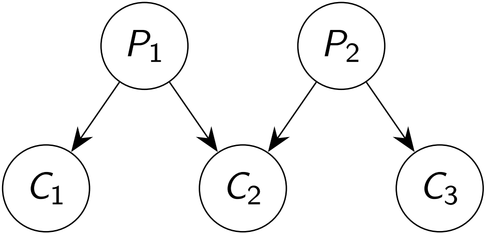
```
Heap
0 -> {CNil[]}
Env
a0   -> {T_Int64}
n0   -> {CNil[]}
n1   -> {CCons[{T_Int64},{0}]}
p0   -> {0}
r    -> {CNil[]}
x    -> {T_Int64}
xs   -> {0}
xs'  -> {CNil[]}
Function
grinMain :: {CNil[]}
```

Created-by

```
grinMain =
  a0 <- pure 5
  n0 <- pure (CNil)
  p0 <- store n0
  n1 <- pure (CCons a0 p0)
  r  <- case n1 of
    (CNil) ->
      pure (CNil)
    (CCons x xs) ->
      xs' <- fetch xs
      pure xs'
  pure r
```

```
Producers
a0      -> {}
n0      -> {CNil{n0}}
n1      -> {CCons{n1}}
p0      -> {}
r       -> {CNil{n0}}
x       -> {}
xs      -> {}
xs'     -> {CNil{n0}}
```

Producers and consumers



Liveness

```
grinMain =
  a0 <- pure 5
  n0 <- pure (CNil)
  p0 <- store n0
  n1 <- pure (CCons a0 p0)
  r <- case n1 of
    (CNil) ->
      pure (CNil)
    (CCons x xs) ->
      xs' <- fetch xs
      pure xs'
  pure r
```

```
Heap
0        -> {CNil[]}
Env
a0       -> DEAD
n0       -> {CNil[]}
n1       -> {CCons[DEAD,LIVE]}
p0       -> LIVE
r        -> {CNil[]}
x        -> DEAD
xs       -> LIVE
xs'      -> {CNil[]}
Function
grinMain :: {CNil[]}
```

Summary

- Compiling functional programs has its own challenges
- We can make it easier by introducing a new IR
- We can perform elaborate dataflow analyses on the IR, then …
- By transforming the code to a more manageable format, we can utilize the already existing infrastructure of LLVM