EFOP-3.6.2-16-2017-00013

European Union

A modern look at GRIN

an optimizing functional language back end

Péter Podlovics, Csaba Hruska

Eötvös Loránd University (ELTE),
Budapest, Hungary

STCS-2019

SZÉCHENYI 2020

HUNGARIAN GOVERNMENT

European Union
European Social
Fund

INVESTING IN YOUR FUTURE

Overview

Introduction

Why functional?

- Declarativeness

  pro: can program on a higher abstraction level

- Composability

  pro: can easily piece together smaller programs

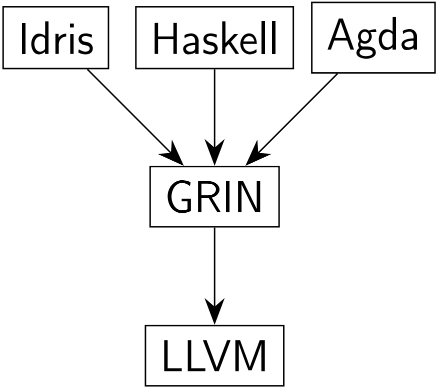
  con: results in a lot of function calls

- Functions are first class citizens

  pro: higher order functions

  con: unknown function calls

Graph Reduction Intermediate Notation

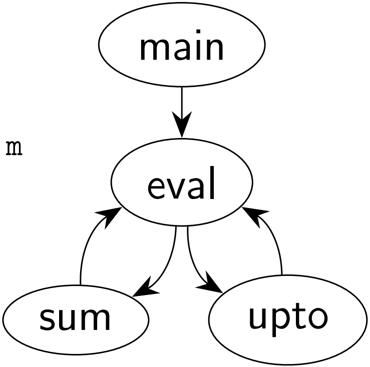


Front end code

```
main = sum (upto 0 10)

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

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

Front end code

```haskell
main = sum (upto 0 10)

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

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



GRIN code

```
grinMain =
  t1 <- store (CInt 1)
  t2 <- store (CInt 10)
  t3 <- store (Fupto t1 t2)
  t4 <- store (Fsum t3)
  (CInt r) <- eval t4
  _prim_int_print r
```

```
eval p =
  v <- fetch p
  case v of
    (CInt n)       -> pure v
    (CNil)         -> pure v
    (CCons y ys)   -> pure v
    (Fupto a b) ->
      zs <- upto a b
      update p zs
      pure zs
    (Fsum c) ->
      s <- sum c
      update p s
      pure s
```

Transformation machinery

- Inline calls to `eval`
- Run dataflow analyses:
  - Heap points-to analysis
  - Sharing analysis
- Run transformations until we reach a fixed-point:
  - Sparse Case Optimization
  - Common Subexpression Elimination
  - Generalized Unboxing
  - etc ...

Extensions

Extending Heap points-to

$$
\begin{aligned}
1 &\rightarrow \{\, \texttt{CInt}[\{BAS\}] \,\} \\
2 &\rightarrow \{\, \texttt{CInt}[\{BAS\}] \,\} \\
3 &\rightarrow \{\, \texttt{Fupto}[\{1\}, \{2\}], \texttt{CNil}[\,], \texttt{CCons}[\{1, 5\}, \{6\}] \,\} \\
4 &\rightarrow \{\, \texttt{Fsum}[\{3\}], \texttt{CInt}[\{BAS\}] \,\} \\
5 &\rightarrow \{\, \texttt{CInt}[\{BAS\}] \,\} \\
6 &\rightarrow \{\, \texttt{Fupto}[\{5\}, \{2\}], \texttt{CNil}[\,], \texttt{CCons}[\{1, 5\}, \{6\}] \,\}
\end{aligned}
$$

Extending Heap points-to

$$
\begin{aligned}
1 &\to \{\, \texttt{CInt}[\{BAS\}] \,\} \\
2 &\to \{\, \texttt{CInt}[\{BAS\}] \,\} \\
3 &\to \{\, \texttt{Fupto}[\{1\}, \{2\}], \texttt{CNil}[\,], \texttt{CCons}[\{1, 5\}, \{6\}] \,\} \\
4 &\to \{\, \texttt{Fsum}[\{3\}], \texttt{CInt}[\{BAS\}] \,\} \\
5 &\to \{\, \texttt{CInt}[\{BAS\}] \,\} \\
6 &\to \{\, \texttt{Fupto}[\{5\}, \{2\}], \texttt{CNil}[\,], \texttt{CCons}[\{1, 5\}, \{6\}] \,\}
\end{aligned}
$$

$$\mathit{BAS} \in \{\mathsf{Int64}, \mathsf{Float}, \mathsf{Bool}, \mathsf{String}, \mathsf{Char}\}$$

Extending Heap points-to

$$
\begin{aligned}
1 &\rightarrow \{\, \texttt{CInt}[\{BAS\}] \,\} \\
2 &\rightarrow \{\, \texttt{CInt}[\{BAS\}] \,\} \\
3 &\rightarrow \{\, \texttt{Fupto}[\{1\},\{2\}], \texttt{CNil}[\,], \texttt{CCons}[\{1,5\},\{6\}] \,\} \\
4 &\rightarrow \{\, \texttt{Fsum}[\{3\}], \texttt{CInt}[\{BAS\}] \,\} \\
5 &\rightarrow \{\, \texttt{CInt}[\{BAS\}] \,\} \\
6 &\rightarrow \{\, \texttt{Fupto}[\{5\},\{2\}], \texttt{CNil}[\,], \texttt{CCons}[\{1,5\},\{6\}] \,\}
\end{aligned}
$$

$$BAS \in \{\mathsf{Int64}, \mathsf{Float}, \mathsf{Bool}, \mathsf{String}, \mathsf{Char}\}$$

```
indexArray# :: Array# a -> Int# -> (# a #)
newMutVar#  :: a -> s -> (# s, MutVar# s a #)
```

LLVM back end

```
grinMain =
 t1 <- store (CInt 1)
 t2 <- store (CInt 10)
 t3 <- store (Fupto t1 t2)
 t4 <- store (Fsum t3)
 (CInt r') <- eval t4
 _prim_int_print r'

upto m n =
 (CInt m') <- eval m
 (CInt n') <- eval n
 b' <- _prim_int_gt m' n'
 case b' of
   #True -> pure (CNil)

sum l = ...

eval p = ...
```

LLVM back end

```
grinMain =
 t1 <- store (CInt 1)
 t2 <- store (CInt 10)
 t3 <- store (Fupto t1 t2)
 t4 <- store (Fsum t3)
 (CInt r') <- eval t4
 _prim_int_print r'

upto m n =
 (CInt m') <- eval m
 (CInt n') <- eval n
 b' <- _prim_int_gt m' n'
 case b' of
   #True -> pure (CNil)

sum l = ...

eval p = ...
```

```
grinMain =
 n1 <- sum 0 1 10
 _prim_int_print n1

sum s lo hi =
 b <- _prim_int_gt lo hi
 if b then
  pure s
 else
  lo' <- _prim_int_add lo 1
  s' <- _prim_int_add s lo
  sum s' lo' hi
```

LLVM back end

```
grinMain =
 t1 <- store (CInt 1)
 t2 <- store (CInt 10)
 t3 <- store (Fupto t1 t2)
 t4 <- store (Fsum t3)
 (CInt r') <- eval t4
 _prim_int_print r'

upto m n =
 (CInt m') <- eval m
 (CInt n') <- eval n
 b' <- _prim_int_gt m' n'
 case b' of
   #True -> pure (CNil)

sum l = ...

eval p = ...
```

```
grinMain =
 n1 <- sum 0 1 10
 _prim_int_print n1

sum s lo hi =
 b <- _prim_int_gt lo hi
 if b then
  pure s
 else
  lo' <- _prim_int_add lo 1
  s' <- _prim_int_add s lo
  sum s' lo' hi
```

```
grinMain:
# BB#0:
  movabsq    $55, %rdi
  jmp     _prim_int_print
```

Dead Data Elimination

Dead data elimination

```
length : List a -> Nat
length Nil = Z
length (Cons x xs)
  = S (length xs)
```

$\xRightarrow{\text{DDE}}$

```
length p =
 xs <- fetch p
 case xs of
  (Cons ys) ->
   l1 <- length ys
   l2 <- _prim_int_add l1 1
   pure l2
  (Nil) ->
    pure 0
```

Applications

- Map $\rightarrow$ Set
- Type class dictionaries
- Type erasure for dependently typed languages

What do we need?

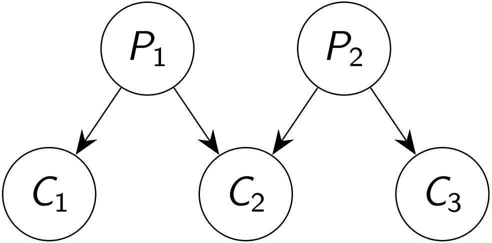
- Producers & consumers
- Detect dead fields
- Connect consumers to producer
- Remove or transform dead fields

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}}
```

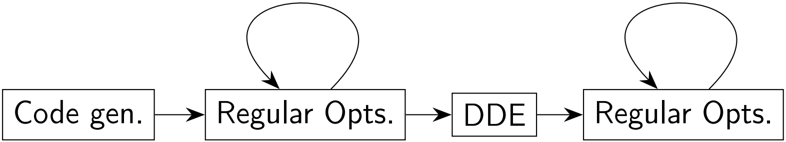
Producers and consumers



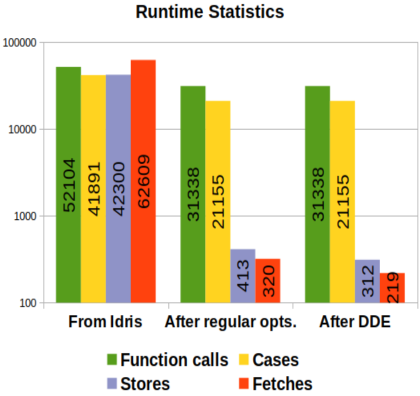
Results

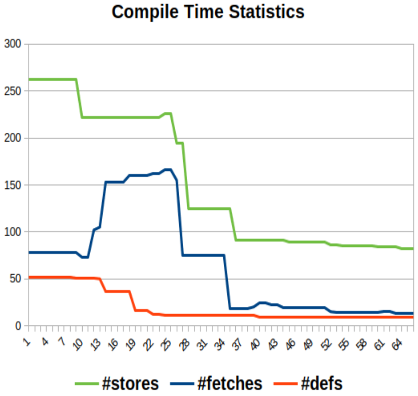
Setup

- Small Idris code snippets from:
  Type-driven Development with Idris by Edwin Brady
- Only interpreted code
- Compile- & runtime measurements
- Pipeline setup:

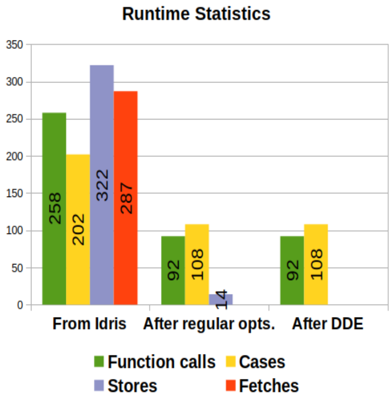


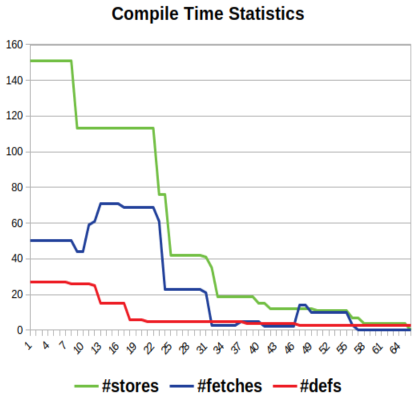
Length


Runtime Statistics
100000
10000
1000
100
52104
41891
42300
62609
31338
21155
413
320
31338
21155
312
219
From Idris
After regular opts.
After DDE
Function calls
Cases
Stores
Fetches

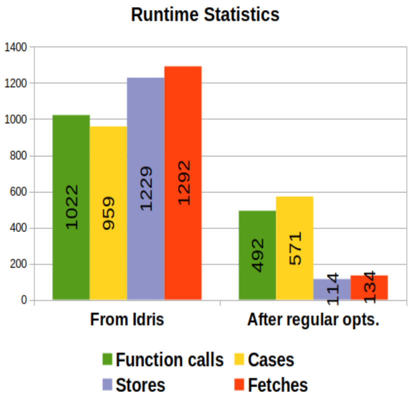

Compile Time Statistics
300
250
200
150
100
50
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64
#stores
#fetches
#defs

Exact length

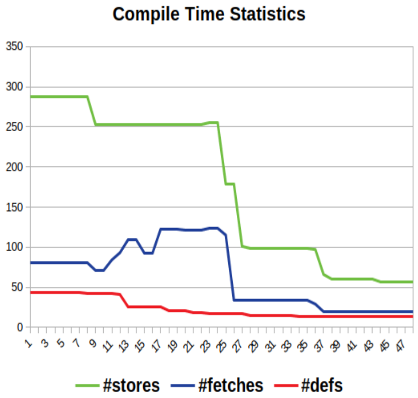




Reverse

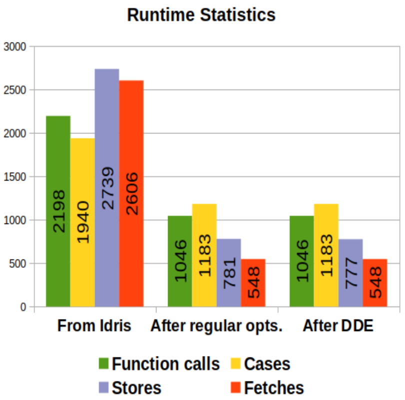


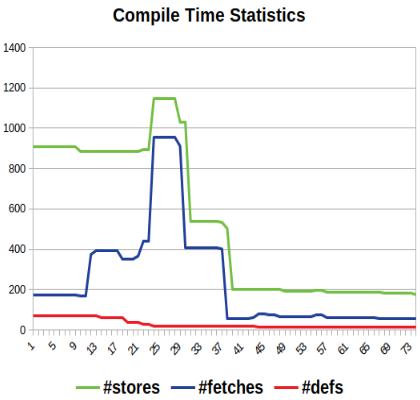
Runtime Statistics
1400
1200
1000
800
600
400
200
0
1022
959
1229
1292
492
571
114
134
From Idris
After regular opts.
Function calls
Cases
Stores
Fetches



Compile Time Statistics
350
300
250
200
150
100
50
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47
#stores
#fetches
#defs

Type level functions





Conclusions

- The optimizer works well:
  - the number of stores, fetches, function calls and pattern matches significantly decreased
  - the structure of the code resembles that of an imperative language
- Dead Data Elimination:
  - is a bit costly
  - is a specific optimization
  - can completely transform data structures
  - can trigger further transformations

EFOP-3.6.2-16-2017-00013

innovation informatics infocommunications

European Union

THANK YOU FOR YOUR ATTENTION!

SZÉCHENYI 2020

HUNGARIAN GOVERNMENT

European Union
European Social Fund

INVESTING IN YOUR FUTURE

Sparse case optimization

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

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

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

Compiled data flow analysis

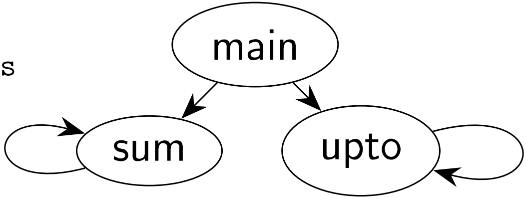
- Analyzing the syntax tree has an interpretation overhead
- We can work around this by "compiling" our analysis into an executable program
- The compiled abstract program is independent of the AST
- It can be executed in a different context (ie.: by another program or on GPU)
- After run (iteratively),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

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
```

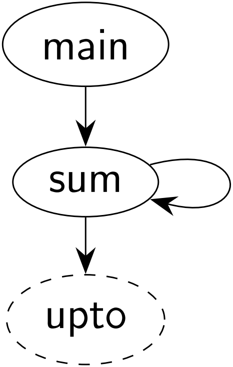
Strict control flow

```
main = sum (upto 0 10)

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

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



Optimized lazy control flow

```
main = sum (upto 0 10)

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

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

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

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)
```

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

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
```

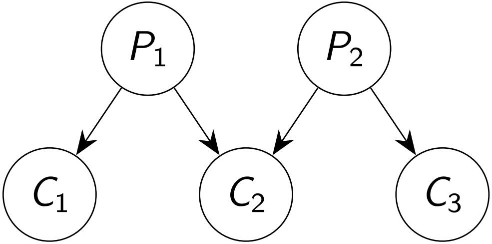
```
Heap
0 -> {CNil[]}
Env
a0  -> {T_Int64}
n0  -> {CNil[]}
n1  -> {CCons[{T_Int64},{0}]}
p0  -> {0}
r   -> {CNil[]}
x   -> {T_Int64}
xs  -> {0}
xs' -> {CNil[]}
Function
grinMain :: {CNil[]}
```

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[]}
```

Results

- eval inlinig impact on code size
- dead code elimination impact on code size
- dead code elimination impact on performance
- comparing intra- and interprocedural dead code elimination
- how costly they are?
- how the resulting codes differ?
- how should the transformations be ordered to minimize compilation time, and maximize performance?
- how costly are the analyses?
- how does the GRIN optimized code compare to GHC's?

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