EFOP-3.6.2-16-2017-00013

European Union

Comparing STG and GRIN

Péter Podlovics, Csaba Hruska, Andor Pénzes

Eötvös Loránd University (ELTE),
Budapest, Hungary

Haskell meetup

SZÉCHENYI 2020

Hungarian Government

European Union
European Social
Fund

INVESTING IN YOUR FUTURE

Overview

Codes

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

High level overview

Spineless Tagless
G-machine

Graph Reduction
Intermediate Notation

High level overview

Spineless Tagless
G-machine

Graph Reduction
Intermediate Notation

- higher order
  functional language

High level overview

Spineless Tagless
G-machine

Graph Reduction
Intermediate Notation

- higher order functional language
- execution of lambda calculus

High level overview

Spineless Tagless
G-machine

Graph Reduction
Intermediate Notation

- higher order functional language
- execution of lambda calculus
- implicit operational semantics

High level overview

Spineless Tagless
G-machine

- higher order functional language
- execution of lambda calculus
- implicit operational semantics
- efficient code generation

Graph Reduction
Intermediate Notation

High level overview

Spineless Tagless
G-machine

- higher order functional language
- execution of lambda calculus
- implicit operational semantics
- efficient code generation

Graph Reduction
Intermediate Notation

- first order imperative language

High level overview

Spineless Tagless
G-machine

- higher order functional language
- execution of lambda calculus
- implicit operational semantics
- efficient code generation

Graph Reduction
Intermediate Notation

- first order imperative language
- unified back end for functional languages

High level overview

Spineless Tagless G-machine

- higher order functional language
- execution of lambda calculus
- implicit operational semantics
- efficient code generation

Graph Reduction Intermediate Notation

- first order imperative language
- unified back end for functional languages
- explicit operational semantics

High level overview

Spineless Tagless
G-machine

- higher order functional language
- execution of lambda calculus
- implicit operational semantics
- efficient code generation

Graph Reduction
Intermediate Notation

- first order imperative language
- unified back end for functional languages
- explicit operational semantics
- aggressive code optimization

STG overview

λx

STG overview

λx

STG overview

λx

STG overview

λx

STG overview

λx

```
and :: Bool -> Bool -> Bool
and True True = True
and _    _    = False
```

STG overview

λx

```
and True True = True
and _    _    = False
```

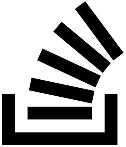
STG overview

λx

```
and x y = case x of
  True -> case y of
    True -> True
    y' -> False
  x' -> False
```

STG overview

λx

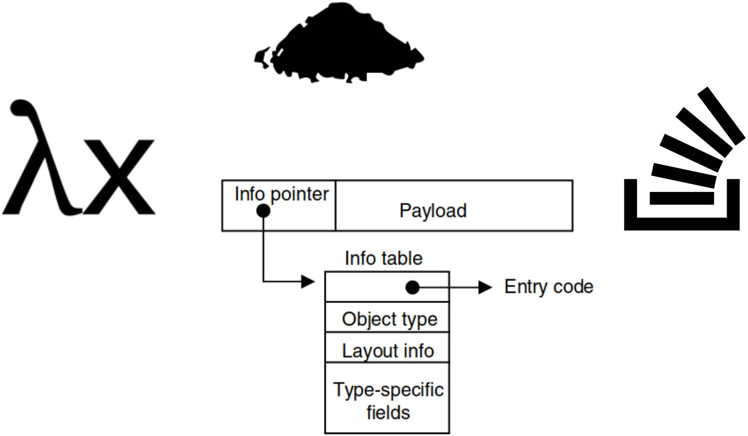


```
and = \x y -> case x of
  True -> case y of
    True -> True
    y' -> False
  x' -> False
```

STG overview

λx

STG overview



STG overview

λx

STG overview

λx

`case * of {...}`

STG overview

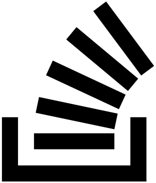
λx

`case * of {...}`

`Update x *`

STG overview

λx



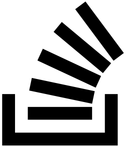
`case * of {...}`

`Update x *`

`* x y z`

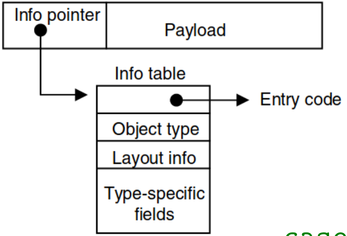
STG overview

λx



STG overview

```
and = \x y -> case x of
 True -> case y of
  True -> True
   y' -> False
 x' -> False
```




```
case * of {...}
Update x *
* x y z
```

STG id-add

```
id = \x -> x
```

STG id-add

```
id = \x -> x

zero = \ -> Int# 0#;
one  = \ -> Int# 1#;
```

STG id-add

```
id = \x -> x

zero = \ -> Int# 0#;
one  = \ -> Int# 1#;

add = \x y -> case x of
  Int# x' -> case y of
    Int# y' -> case +# x' y' of
      r -> Int# r;
    badInt -> Error_min badInt;
  badInt -> Error_min badInt;
```

STG id-add

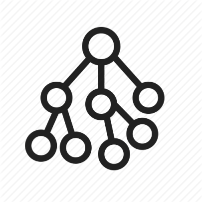
```
id = \x -> x

zero = \ -> Int# 0#;
one  = \ -> Int# 1#;

add = \x y -> case x of
  Int# x' -> case y of
    Int# y' -> case +# x' y' of
      r -> Int# r;
    badInt -> Error_min badInt;
  badInt -> Error_min badInt;

main = \ -> let add_one = \ -> add one
            in id add_one zero
```

STG id-add

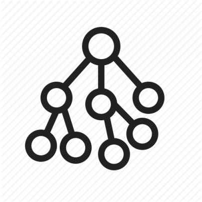
```
id = \x -> x

zero = \ -> Int# 0#;
one  = \ -> Int# 1#;

add = \x y -> case x of
  Int# x' -> case y of
    Int# y' -> case +# x' y' of
      r -> Int# r;
    badInt -> Error_min badInt;
  badInt -> Error_min badInt;

main = \ => let add_one = \ -> add one
            in id add_one zero
```

GRIN overview

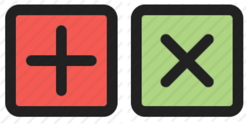
GRIN overview

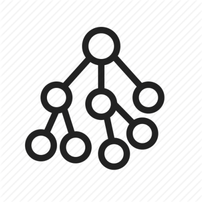


GRIN overview

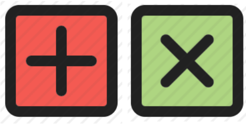


GRIN overview

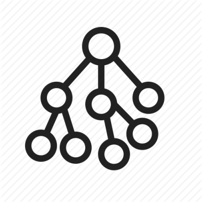




GRIN overview

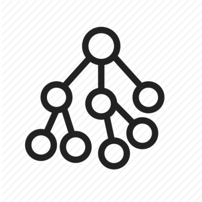


- C-node

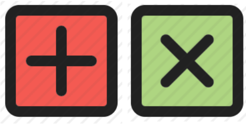


GRIN overview

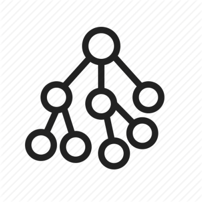
- C-node
- F-node



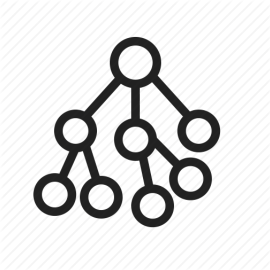
GRIN overview



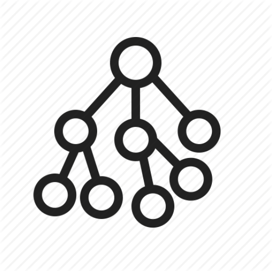
- C-node
- F-node
- P-node



GRIN overview

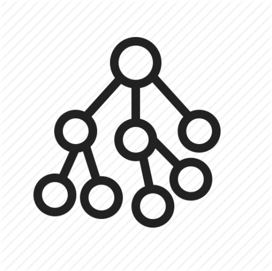


GRIN overview



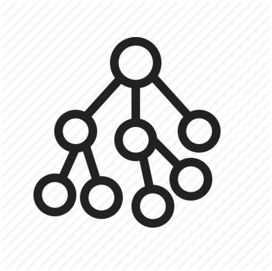
- store

GRIN overview



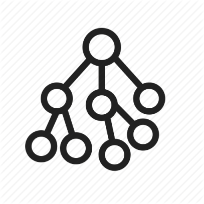
- store
- fetch

GRIN overview

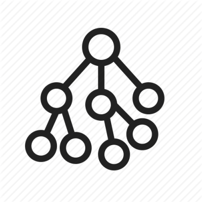


- store
- fetch
- update

GRIN overview

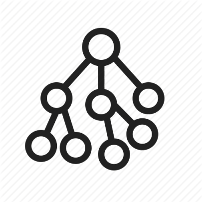


GRIN overview



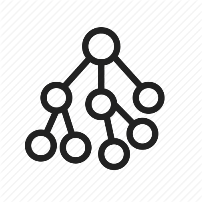
- eval

GRIN overview



- eval
- apply

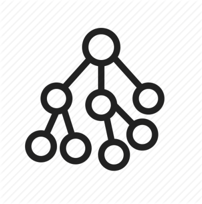
GRIN overview



- eval
- apply
- analyses

GRIN overview

- C-node
- F-node
- P-node



- store
- fetch
- update

- eval
- apply
- analyses

GRIN id

```
id x.0 =
 x.0' <- eval x.0
 pure x.0'
```

GRIN id

```
id x.0 =
 x.0' <- eval x.0
 pure x.0'

eval p =
 v <- fetch p
 case v of
  (CInt _n) -> pure v
  (Fid x.1) ->
   r.id <- id x.1
   update p r.id
   pure r.id
```

GRIN id

```
id x.0 =
 x.0' <- eval x.0
 pure x.0'

eval p =
 v <- fetch p
 case v of
  (CInt _n) -> pure v
  (Fid x.1) ->
   r.id <- id x.1
   update p r.id
   pure r.id
```

```
id_one =
 one     <- pure (CInt 1)
 one_ptr <- store one
 thunk   <- pure (Fid one)
 pure thunk
```

GRIN id

```
id x.0 =
 x.0' <- eval x.0
 pure x.0'

eval p =
 v <- fetch p
 case v of
  (CInt _n) -> pure v
  (Fid x.1) ->
   r.id <- id x.1
   update p r.id
   pure r.id
```

```
id_one =
 one     <- pure (CInt 1)
 one_ptr <- store one
 thunk   <- pure (Fid one)
 pure thunk
```

```
grinMain =
 (CInt k) <- eval id_one
 _prim_int_print k
```

GRIN add

```
add x y =
 (CInt x') <- eval x
 (CInt y') <- eval y
 r <- _int_add x' y'
 pure (CInt r)
```

GRIN add

```
add x y =
 (CInt x') <- eval x
 (CInt y') <- eval y
 r <- _int_add x' y'
 pure (CInt r)

eval p =
 v <- fetch p
 case v of
  (CInt _n) -> pure v
  (Fadd x.1 y.1) ->
   r.add <- add x.1 y.1
   update p r.add
   pure r.add
```

GRIN add

```
add x y =
 (CInt x') <- eval x
 (CInt y') <- eval y
 r <- _int_add x' y'
 pure (CInt r)

eval p =
 v <- fetch p
 case v of
  (CInt _n) -> pure v
  (Fadd x.1 y.1) ->
   r.add <- add x.1 y.1
   update p r.add
   pure r.add
```

```
add_one =
 one <- store (CInt 1)
 pure (P1_add one)
```

GRIN add

```
add x y =
 (CInt x') <- eval x
 (CInt y') <- eval y
 r <- _int_add x' y'
 pure (CInt r)

eval p =
 v <- fetch p
 case v of
  (CInt _n) -> pure v
  (Fadd x.1 y.1) ->
   r.add <- add x.1 y.1
   update p r.add
   pure r.add
```

```
add_one =
 one <- store (CInt 1)
 pure (P1_add one)

grinMain =
 zero <- (CInt 0)
 suc <- add_one
 apply suc zero
```

GRIN add

```
add x y =
 (CInt x') <- eval x
 (CInt y') <- eval y
 r <- _int_add x' y'
 pure (CInt r)

eval p =
 v <- fetch p
 case v of
  (CInt _n) -> pure v
  (Fadd x.1 y.1) ->
   r.add <- add x.1 y.1
   update p r.add
   pure r.add
```

```
add_one =
 one <- store (CInt 1)
 pure (P1_add one)

grinMain =
 zero <- (CInt 0)
 suc <- add_one
 apply suc zero

apply f u =
 case f of
  (P2_add) ->
   pure (P1_add u)
  (P1_add z) -> add z u
```

GRIN add

```
add x y =
 (CInt x') <- eval x
 (CInt y') <- eval y
 r <- _int_add x' y'
 pure (CInt r)

eval p =
 v <- fetch p
 case v of
  (CInt _n) -> pure v
  (P2_add) -> pure v
  (P1_add _x) -> pure v
  (Fadd x.1 y.1) ->
   r.add <- add x.1 y.1
   update p r.add
   pure r.add
```

```
add_one =
 one <- store (CInt 1)
 pure (P1_add one)

grinMain =
 zero <- (CInt 0)
 suc <- add_one
 apply suc zero

apply f u =
 case f of
  (P2_add) ->
   pure (P1_add u)
  (P1_add z) -> add z u
```

GRIN id-add

```
                        -- id (add 1) 0 ?
id q =
 q' <- eval q
 pure q'

add x y =
 (CInt x') <- eval x
 (CInt y') <- eval y
 r <- _int_add x' y'
 pure (CInt r)

eval p = ...

apply f u = ...
```

GRIN id-add

```
id q =
 q' <- eval q
 pure q'

add x y =
 (CInt x') <- eval x
 (CInt y') <- eval y
 r <- _int_add x' y'
 pure (CInt r)

eval p = ...

apply f u = ...
```

```
-- id (add 1) 0 ?
grinMain =
 zero <- store (CInt 0)
 one  <- store (CInt 1)

 add_1 <- store (P1_add one)
 thunk <- store (Fid add_1)

 id_add_1 <- eval thunk
 r <- apply id_add_1 zero

 (CInt r) <- pure r
 _prim_int_print r
```

Introduction

Why functional?

- Declarativeness

  pro: can program on a higher abstraction level

- Composability

  pro: can easily piece together smaller programs

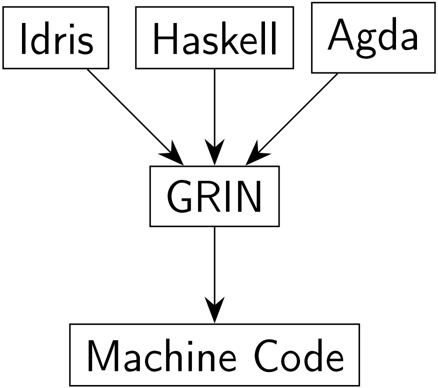
  con: results in a lot of function calls

- Functions are first class citizens

  pro: higher order functions

  con: unknown function calls

Graph Reduction Intermediate Notation

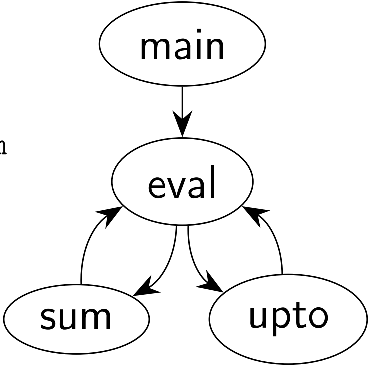


Front end code

```
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

```
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

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

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

Extending Heap points-to

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

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

```
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 I.

```
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
```

Dead data elimination II.

```
data Bin : Nat -> Type where
  N : Bin 0
  O : {n : Nat} -> Bin n -> Bin (2*n + 0)
  I : {n : Nat} -> Bin n -> Bin (2*n + 1)
```

Dead data elimination II.

```
data Bin : Nat -> Type where
  N : Bin 0
  O : {n : Nat} -> Bin n -> Bin (2*n + 0)
  I : {n : Nat} -> Bin n -> Bin (2*n + 1)

binToNat : Bin n -> Nat
binToNat N = 0
binToNat (O {n} _) = 2*n
binToNat (I {n} _) = 2*n + 1
```

Applications

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

What do we need?

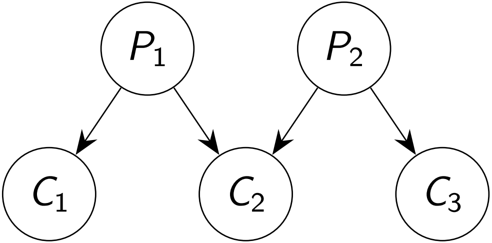
- Producers & consumers
- Detect dead fields
- Connect consumers to producer
- Remove or transform dead fields

Created-by

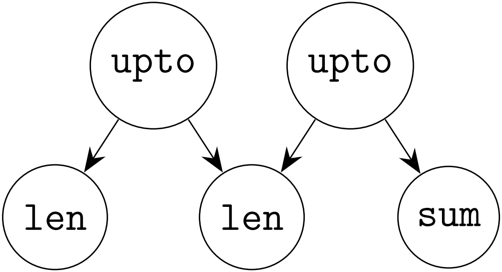
```
null xs =
 y <- case xs of
  (CNil) ->
   a <- pure (CTrue)
   pure a
  (CCons z zs) ->
   b <- pure (CFalse)
   pure b
 pure y
```

Var	Producers
xs	$\mathit{CNil}[\ldots], \mathit{CCons}[\ldots]$
a	$\mathit{CTrue}[\mathrm{a}]$
b	$\mathit{CFalse}[\mathrm{b}]$
y	$\mathit{CTrue}[\mathrm{a}], \mathit{CFalse}[\mathrm{b}]$

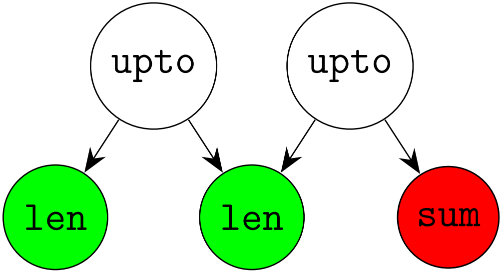
Producers and consumers



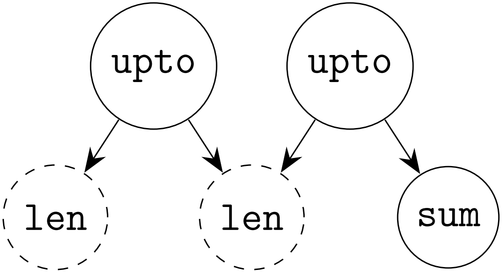
Producers and consumers



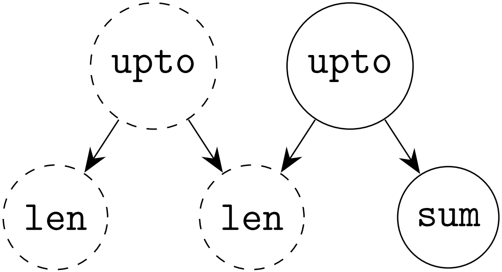
Producers and consumers



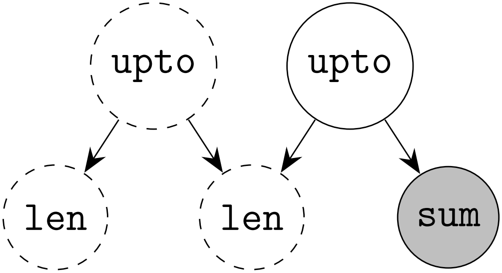
Producers and consumers



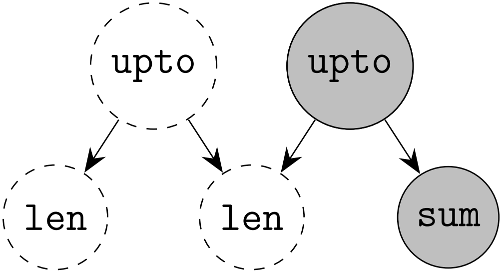
Producers and consumers



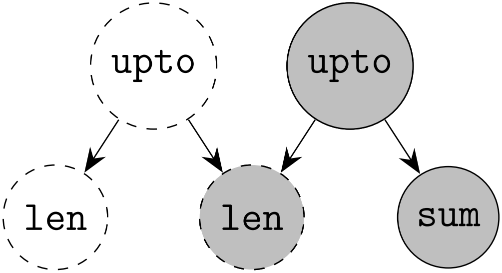
Producers and consumers



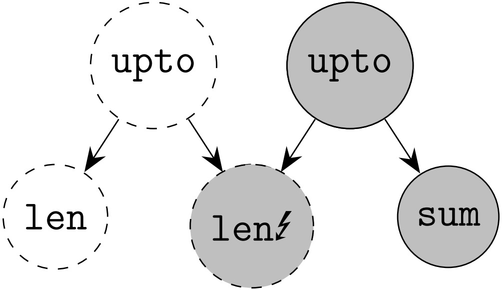
Producers and consumers



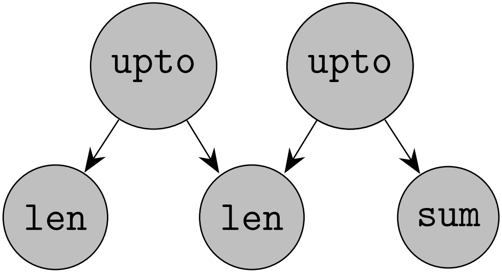
Producers and consumers



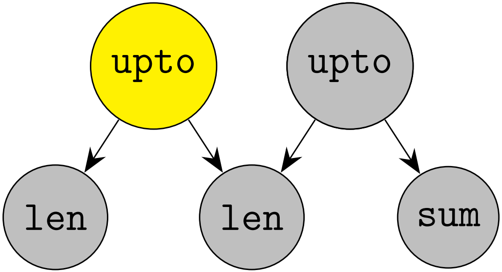
Producers and consumers



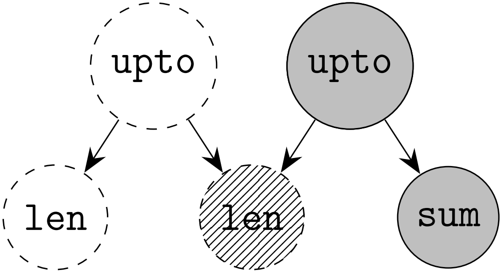
Producers and consumers



Producers and consumers



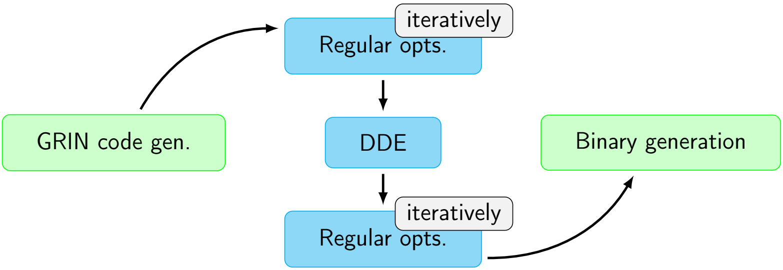
Producers and consumers



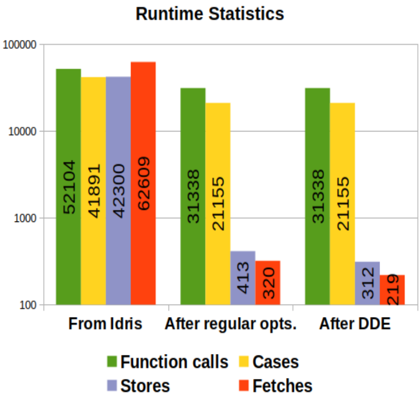
Results

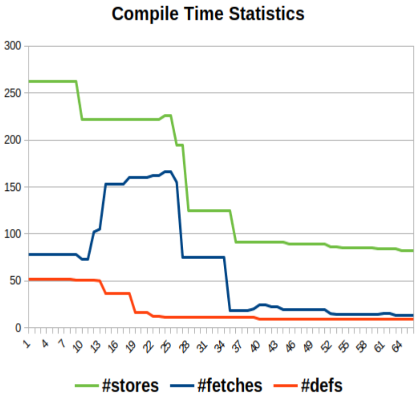
Setup

- Small Idris code snippets from:
  Type-driven Development with Idris by Edwin Brady
- Both interpreted GRIN code and executed binaries
- Compile- & runtime measurements



Length - GRIN statistics

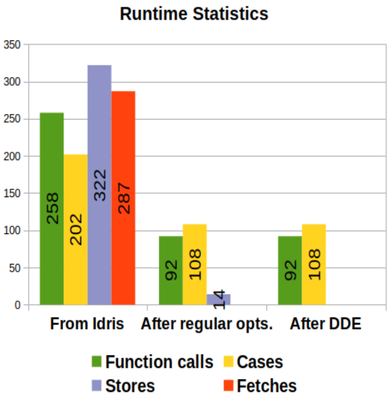


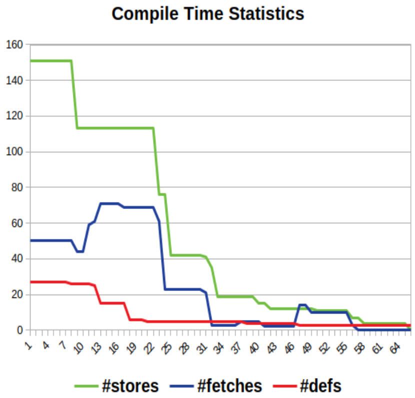


Length - CPU binary statistics

Stage	Size	Instructions	Stores	Loads
`normal-O0`	23928	769588	212567	233305
`normal-O3`	23928	550065	160252	170202
`regular-opt`	19832	257397	14848	45499
`dde-O0`	15736	256062	14243	45083
`dde-O3`	15736	284970	33929	54555

Exact length - GRIN statistics

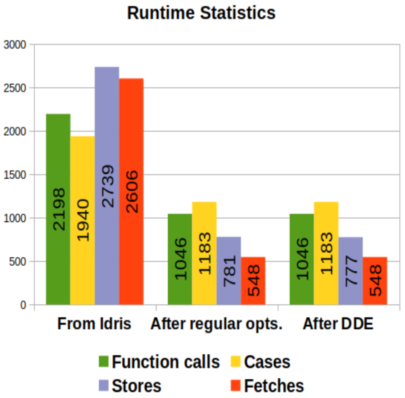


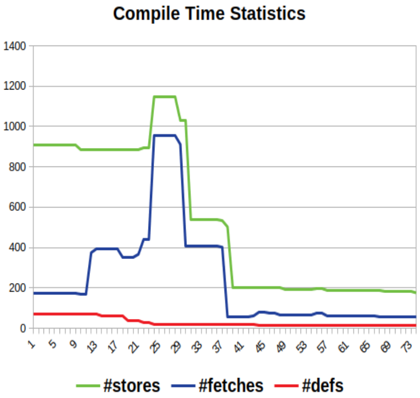


Exact length - CPU binary statistics

Stage	Size	Instructions	Stores	Loads
`normal-O0`	18800	188469	14852	46566
`normal-O3`	14704	187380	14621	46233
`regular-opt`	10608	183560	13462	45214
`dde-O0`	10608	183413	13431	45189
`dde-O3`	10608	183322	13430	44226

Type level functions - GRIN statistics

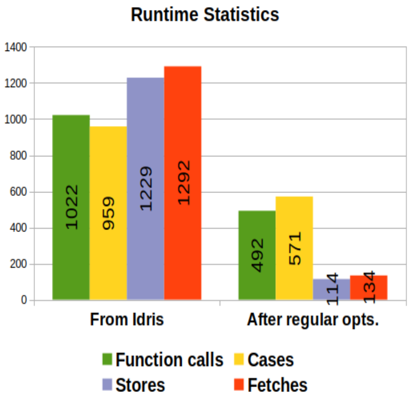


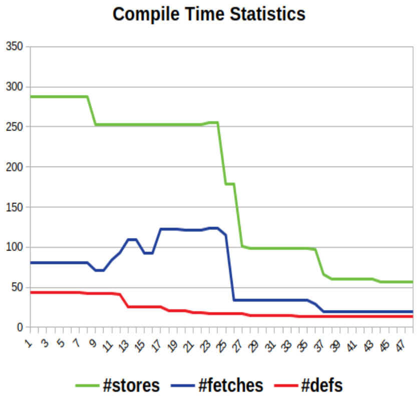


Type level functions - CPU binary statistics

Stage	Size	Instructions	Stores	Loads
`normal-O0`	65128	383012	49191	86754
`normal-O3`	69224	377165	47556	84156
`regular-opt`	36456	312122	34340	71162
`dde-O0`	32360	312075	34331	70530
`dde-O3`	28264	309822	33943	70386

Reverse - GRIN statistics





Reverse - CPU binary statistics

Stage	Size	Instructions	Stores	Loads
`normal-O0`	27112	240983	25018	58253
`normal-O3`	31208	236570	23808	56617
`regular-opt-O0`	14824	222085	19757	53125
`regular-opt-O3`	14824	220837	19599	52827

Conclusions

- Dead Data Elimination:
  - is demanding on resources
  - can completely transform data structures
  - can trigger further transformations
  - can considerably reduce binary size
- Regular optimizations:
  - GRIN works well for dependently-typed languages as well
  - the optimized GRIN code is significantly more efficient
  - the GRIN optimizations are orthogonal to the LLVM optimizations

EFOP-3.6.2-16-2017-00013

THANK YOU FOR YOUR ATTENTION!

Sparse case optimization

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

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

```
<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