EFOP-3.6.2-16-2017-00013

innovation informatics infocommunications

European Union

GRIN: Dead data elimination

in the context of dependently typed languages

Péter Podlovics, Csaba Hruska

Eötvös Loránd University (ELTE),
Budapest, Hungary

EUTypes-2019

SZÉCHENYI 2020

HUNGARIAN GOVERNMENT

European Union
European Social Fund

INVESTING IN YOUR FUTURE

Overview

Introduction

Extensions

Dead Data Elimination

Results

Introduction

Why functional?

- Declarativeness

  pro: can program on a higher abstraction level

- Composability

  pro: can easily piece together smaller programs

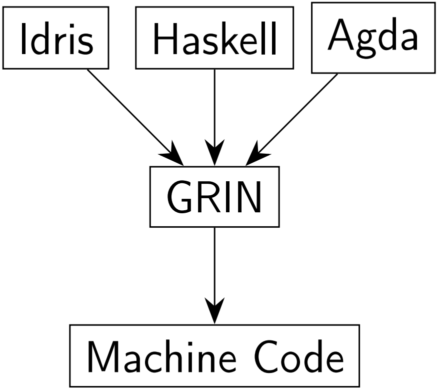
  con: results in a lot of function calls

- Functions are first class citizens

  pro: higher order functions

  con: unknown function calls

Graph Reduction Intermediate Notation

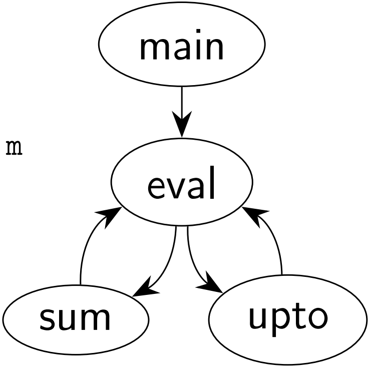


Front end code

```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
```

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

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

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

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)
```

$$\overset{\text{DDE}}{\Longrightarrow}$$

```
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 → Set
- Type class dictionaries
- Type erasure for dependently typed languages

What do we need?

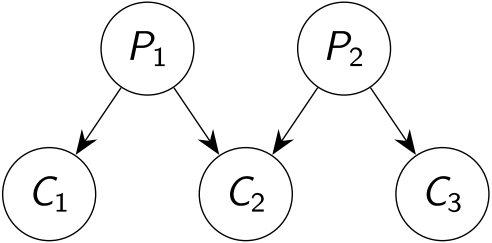
- Producers & consumers
- Detect dead fields
- Connect consumers to producer
- Remove or transform dead fields

Created-by

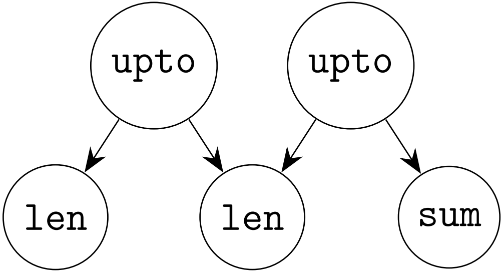
```
null xs =
 y <- case xs of
  (CNil) ->
   a <- pure (CTrue)
   pure a
  (CCons z zs) ->
   b <- pure (CFalse)
   pure b
 pure y
```

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

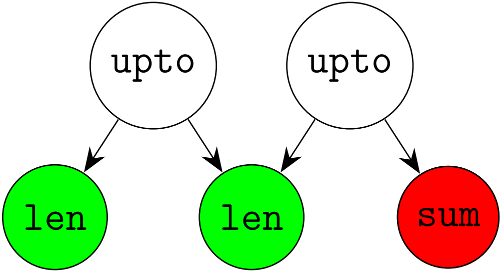
Producers and consumers



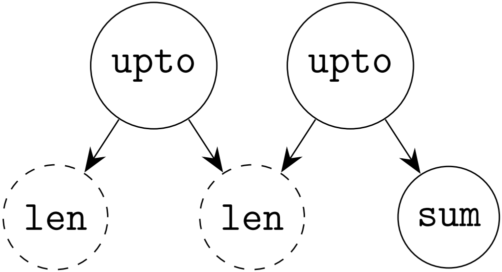
Producers and consumers



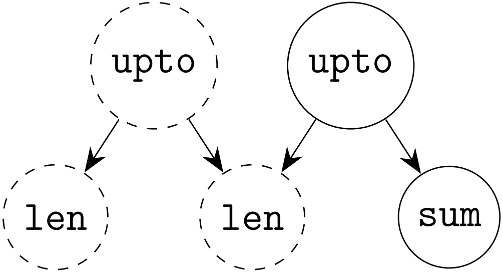
Producers and consumers



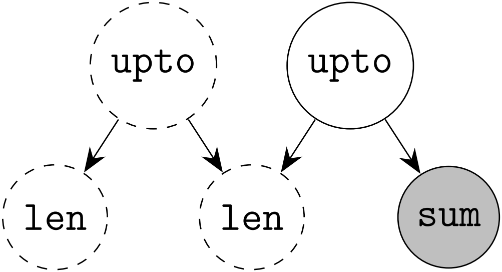
Producers and consumers



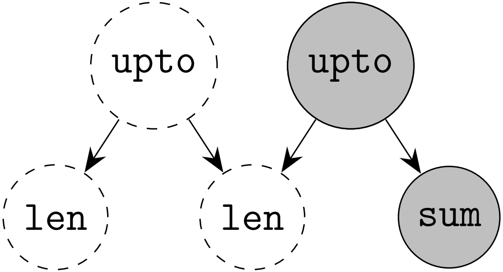
Producers and consumers



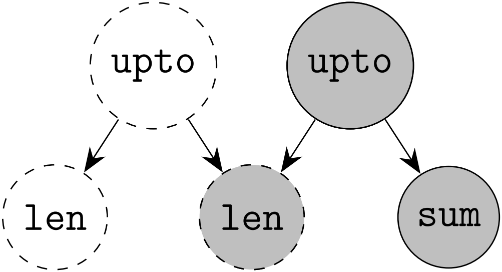
Producers and consumers



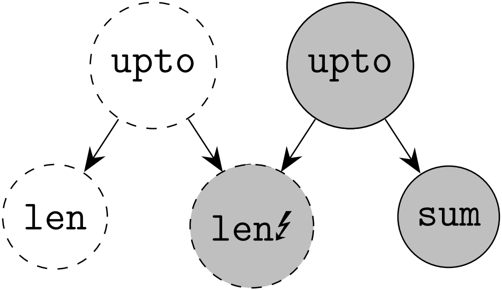
Producers and consumers



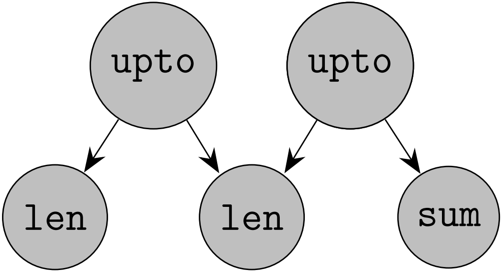
Producers and consumers



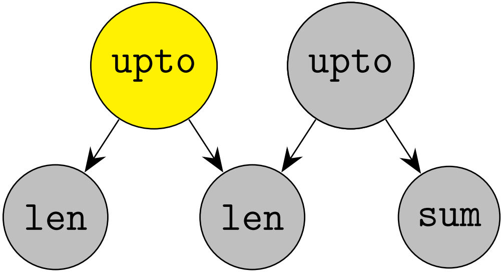
Producers and consumers



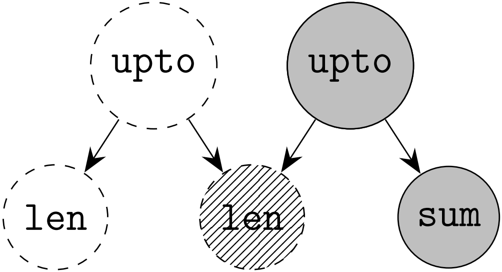
Producers and consumers



Producers and consumers



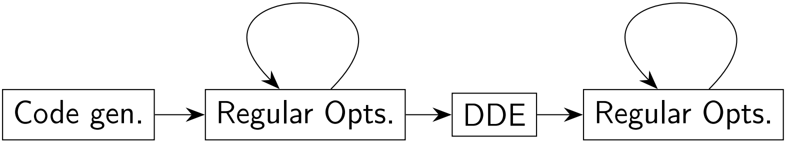
Producers and consumers



Results

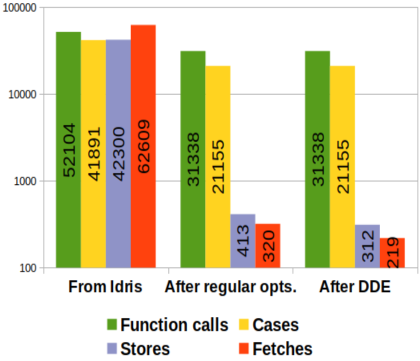
Setup

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

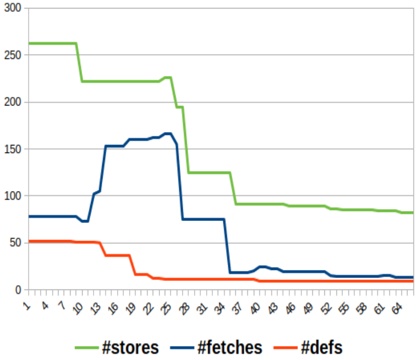


Length

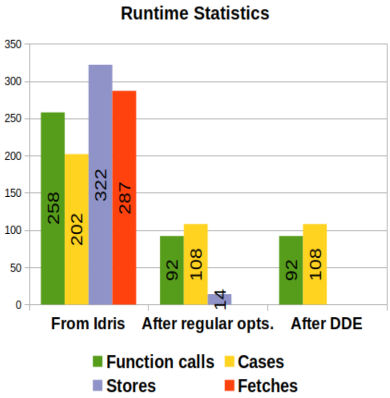
Runtime Statistics

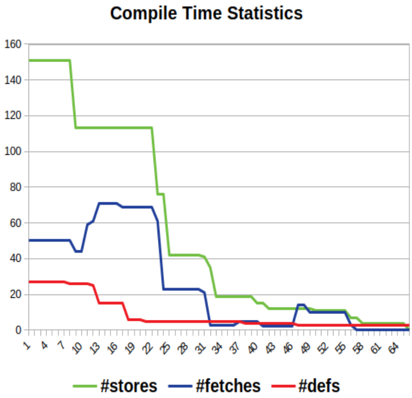


Compile Time Statistics

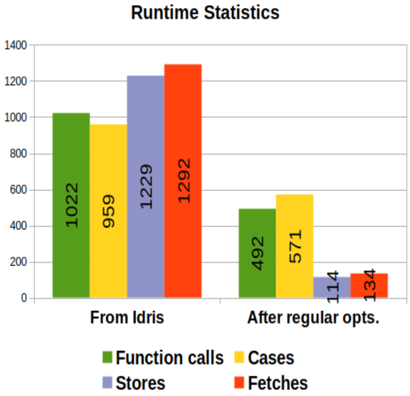


Exact length

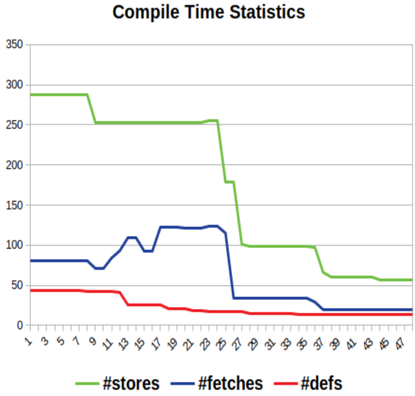

Runtime Statistics


Compile Time Statistics

Reverse

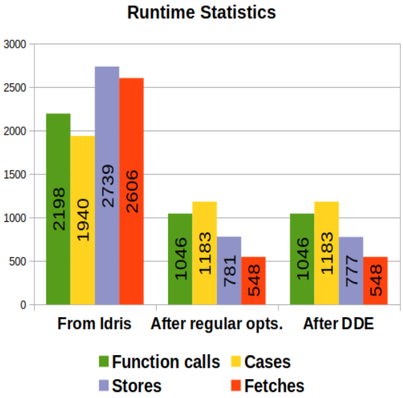


Runtime Statistics

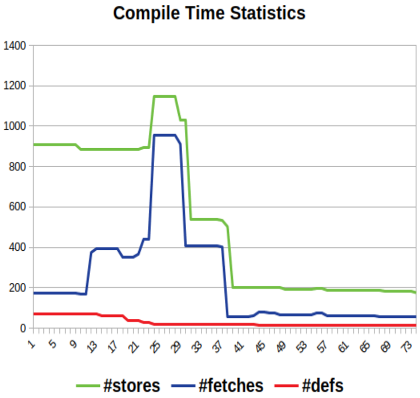


Compile Time Statistics

Type level functions



Runtime Statistics



Compile Time Statistics

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>
```

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

```
<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