EFOP-3.6.2-16-2017-00013

Comparing STG and GRIN

Péter Podlovics, Csaba Hruska, Andor Pénzes

Eötvös Loránd University (ELTE),
Budapest, Hungary

Haskell meetup

Overview

Spineless Tagless G-machine

STG examples

STG demonstration

Graph Reduction Intermediate Notation

GRIN examples

GRIN demonstration

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

- higher order functional language

Graph Reduction Intermediate Notation

High level overview

Spineless Tagless G-machine

- higher order functional language
- execution of lambda calculus

Graph Reduction Intermediate Notation

High level overview

Spineless Tagless G-machine

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

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

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

Spineless Tagless G-machine

STG overview

STG overview

STG overview

STG overview

STG overview

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

STG overview

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

STG overview

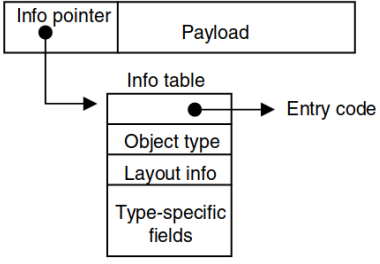
```
and x y = case x of
  True -> case y of
    True -> True
    y' -> False
  x' -> False
```

STG overview

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

STG overview

STG overview



STG overview

STG overview

STG overview

```
case * of {...}
```

STG overview

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

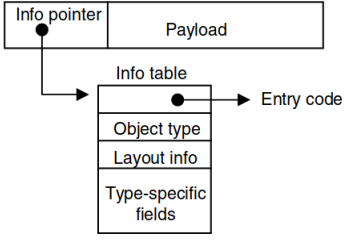
STG overview

```
case * of {...}

Update x *

* x y z
```

STG overview

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 examples

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
```

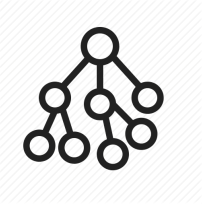
STG demonstration

Graph Reduction Intermediate Notation

GRIN overview

GRIN overview

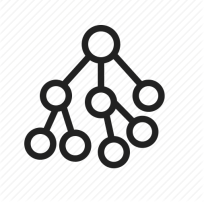
GRIN overview



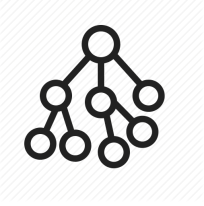
GRIN overview

GRIN overview

- C-node

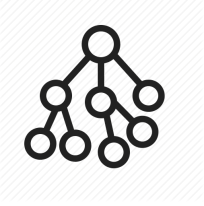


GRIN overview



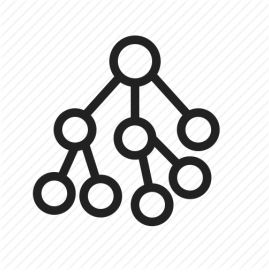
- C-node
- F-node

GRIN overview

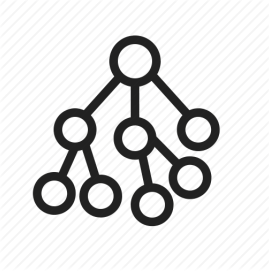


- C-node
- F-node
- P-node

GRIN overview

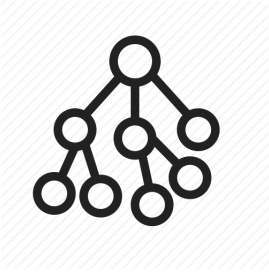


GRIN overview



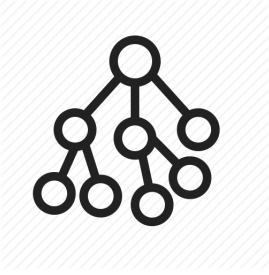
- store

GRIN overview



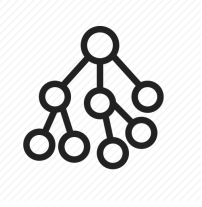
- store
- fetch

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 examples

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)
 ptr   <- store one
 thunk <- pure (Fid ptr)
 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)
 ptr   <- store one
 thunk <- pure (Fid ptr)
 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 <- store (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 <- store (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 =
 <...>

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 <- store (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
```

GRIN demonstration

Consequences of the execution models

STG

GRIN

- closures:

Consequences of the execution models

STG

- closures:
  - represented by heap objects

GRIN

Consequences of the execution models

STG

- closures:
  - represented by heap objects
  - they need special treatment

GRIN

Consequences of the execution models

STG

- closures:
  - represented by heap objects
  - they need special treatment
  - generic data layout

GRIN

Consequences of the execution models

STG

- closures:
  - represented by heap objects
  - they need special treatment
  - generic data layout
  - not representable in a register

GRIN

Consequences of the execution models

STG

- closures:
  - represented by heap objects
  - they need special treatment
  - generic data layout
  - not representable in a register
- execution stack:

GRIN

Consequences of the execution models

STG

- closures:
  - represented by heap objects
  - they need special treatment
  - generic data layout
  - not representable in a register
- execution stack:
  - custom stack

GRIN

Consequences of the execution models

STG

- closures:
  - represented by heap objects
  - they need special treatment
  - generic data layout
  - not representable in a register
- execution stack:
  - custom stack
  - custom calling convention for LLVM

GRIN

Consequences of the execution models

STG

- closures:
  - represented by heap objects
  - they need special treatment
  - generic data layout
  - not representable in a register
- execution stack:
  - custom stack
  - custom calling convention for LLVM

GRIN

- closures:

Consequences of the execution models

STG

- closures:
  - represented by heap objects
  - they need special treatment
  - generic data layout
  - not representable in a register
- execution stack:
  - custom stack
  - custom calling convention for LLVM

GRIN

- closures:
  - only data, no builtins

Consequences of the execution models

STG

- closures:
  - represented by heap objects
  - they need special treatment
  - generic data layout
  - not representable in a register
- execution stack:
  - custom stack
  - custom calling convention for LLVM

GRIN

- closures:
  - only data, no builtins
  - standard optimizations work

Consequences of the execution models

STG

- closures:
  - represented by heap objects
  - they need special treatment
  - generic data layout
  - not representable in a register
- execution stack:
  - custom stack
  - custom calling convention for LLVM

GRIN

- closures:
  - only data, no builtins
  - standard optimizations work
  - custom data layout (C-style tagged union)

Consequences of the execution models

STG

- closures:
  - represented by heap objects
  - they need special treatment
  - generic data layout
  - not representable in a register
- execution stack:
  - custom stack
  - custom calling convention for LLVM

GRIN

- closures:
  - only data, no builtins
  - standard optimizations work
  - custom data layout (C-style tagged union)
  - can be put into registers

Consequences of the execution models

STG

- closures:
  - represented by heap objects
  - they need special treatment
  - generic data layout
  - not representable in a register
- execution stack:
  - custom stack
  - custom calling convention for LLVM

GRIN

- closures:
  - only data, no builtins
  - standard optimizations work
  - custom data layout (C-style tagged union)
  - can be put into registers
- execution stack:

Consequences of the execution models

STG

- closures:
  - represented by heap objects
  - they need special treatment
  - generic data layout
  - not representable in a register
- execution stack:
  - custom stack
  - custom calling convention for LLVM

GRIN

- closures:
  - only data, no builtins
  - standard optimizations work
  - custom data layout (C-style tagged union)
  - can be put into registers
- execution stack:
  - standard C execution model

Consequences of the execution models

STG

- closures:
  - represented by heap objects
  - they need special treatment
  - generic data layout
  - not representable in a register
- execution stack:
  - custom stack
  - custom calling convention for LLVM

GRIN

- closures:
  - only data, no builtins
  - standard optimizations work
  - custom data layout (C-style tagged union)
  - can be put into registers
- execution stack:
  - standard C execution model
  - we get LLVM for free

EFOP-3.6.2-16-2017-00013

THANK YOU FOR YOUR ATTENTION!