GRIN

An Alternative Haskell Compiler Backend

Csaba Hruska, Andor Penzes

Thomas Johnsson, Urban Boquist

Graph Reduction Intermediate Notation

Graph reduction

GRIN intermediate language

Heap Points-To Analysis

LLVM backend for GRIN

GHC frontend for GRIN

Graph reduction

Graph reduction

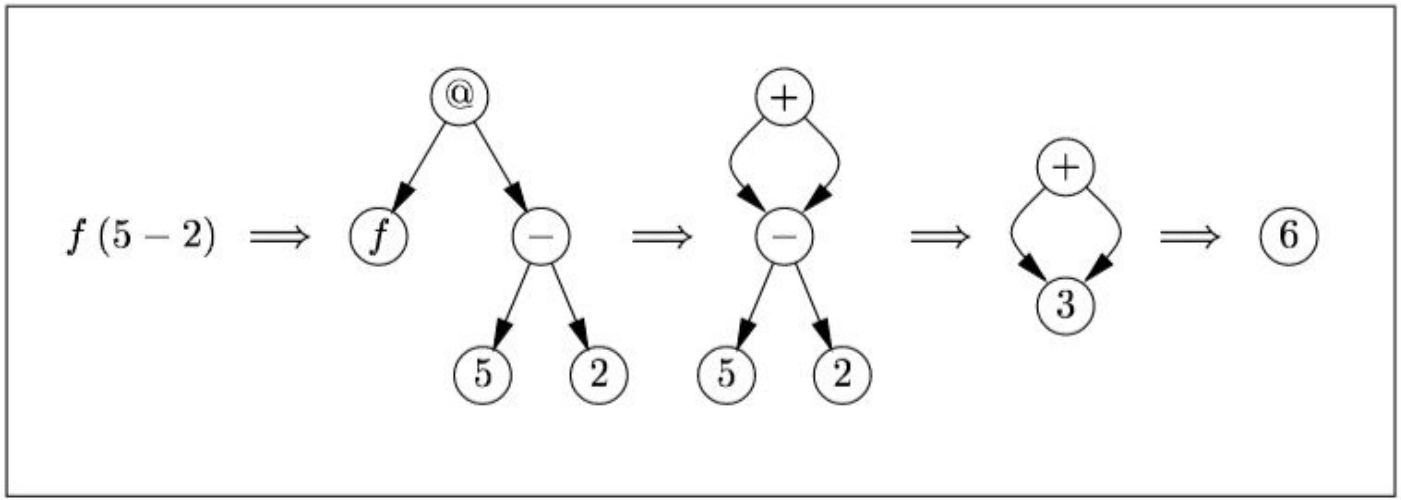


Figure 1.1: A simple example of graph reduction.

Graph reduction

- Initial implementations of graph reduction were in the form of interpreters
- Compiled graph reduction via abstract machines
- G-machine
- Spineless G-machine: Avoid creating apply nodes
- Spineless Tagless G-machine: The tag was eliminated, a code pointer used instead
- Graph Reduction Intermediate Notation: Attack the indirect calls

Graph reduction

Evaluation of a suspended function application

- STG: The function to call is unknown at compile time, the evaluation of a closure will result in some kind of indirect call, through a pointer. This makes optimisations harder.
- GRIN: forbids all unknown calls and uses control flow analysis to approximate the possible targets.

The GRIN Language

GRIN Syntax

Haskell

```
main = print $ sum [1..100000]
```

```
main =
  n13 <- sum 0 1 100000
  _prim_int_print n13
sum n29 n30 n31 =
  b2 <- _prim_int_gt n30 n31
  case b2 of
    #True  -> pure n29
    #False -> n18 <- _prim_int_add n30 1
              n28 <- _prim_int_add n29 n30
              sum n28 n18 n31
```

GRIN Syntax

Haskell

```
main = print (sum 0 (upto 1 10000))

upto :: Int -> Int -> [Int]
upto m n = if m > n
    then []
    else m : upto (m+1) n

sum :: Int -> [Int] -> Int
sum n []       = n
sum n (x:xs) = sum (n+x) xs
```

```
main = t1 <- store (CInt 1)
       t2 <- store (CInt 10000)
       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)
             #False -> m1' <- _prim_int_add m' 1
                       m1 <- store (CInt m1')
                       p <- store (Fupto m1 n)
                       pure (CCons m p)
sum l = l2 <- eval l
        case l2 of
          (CNil)       -> pure (CInt 0)
          (CCons x xs) -> (CInt x') <- eval x
                          (CInt s') <- sum xs
                          ax' <- _prim_int_add x' s'
                          pure (CInt ax')
```

```
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
```

GRIN Syntax

Variables

- Static single assignment

Primitive operations

- **Pure**: Monadic pure
- **Function Call**: Saturated function calls.
- **Store**: Save a node value to a heap, returns a heap location.
- **Fetch**: Loads a node from the given location, returns the loaded value.
- **Update**: Updates the given location with a given node value, returns unit.

Compositional operations

- **Case**: branches on a value, primitive or node.
- **Bind**: Monadic bind
- **Block**: Groups a monadic expression, serves extending the lhs of the bind.

No looping construct, only recursive function calls.

```
upto m n = (CInt m') <- eval m
           (CInt n') <- eval n
           b' <- _prim_int_gt m' n'
           case b' of
             #True  -> pure (CNil)
             #False -> m1' <- _prim_int_add m' 1
                       m1 <- store (CInt m1')
                       p <- store (Fupto m1 n)
                       pure (CCons m p)
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
```

GRIN Semantics

GRIN semantics

- Grin is a strict first order language.
- All values are primitive values or in weak head normal form.
- Suspended computation represented as F-Nodes instead of closures

GRIN types

- **Primitive types**: Int, Double, Bool, ...
- **Locations**: User can not create value of the location type. It is created via the store operation
- **Constant tag node types**
  - Tag C/F/P
  - Name
  - List of values of simple type or location

GRIN monad: State monad with heap as underlying state

- **Pure**: Monadic pure
- **Bind**: binds a result of a computation to a pattern and runs the rest of the computation
- **Store**: Creates a new location on the heap, calculates the node value and stores it on created new location.
- **Fetch**: The parameter of the fetch is a variable which value is a location, looks up the value from the heap.
- **Update**: First parameter a variable which is a location, the second one is a node value, overwrites the location with the value.

GRIN branching: The **case** can be in the position of LHS and RHS of the bind, values to match against can be a primitive value, or a node value or a variable.

- **Value**:Primitive or node value, or a variable of such kind.
- **Alternatives**: Patterns that introduce a literal, a node with variables, excludes variable tag nodes.
- **Default alternative:** It is selected when nothing else matters.

Heap Points-To result

```
main = t1 <- store (CInt 1)
       t2 <- store (CInt 10000)
       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)
             #False -> m1' <- _prim_int_add m' 1
                       m1 <- store (CInt m1')
                       p <- store (Fupto m1 n)
                       pure (CCons m p)
sum l = l2 <- eval l
        case l2 of
          (CNil)       -> pure (CInt 0)
          (CCons x xs) -> (CInt x') <- eval x
                          (CInt s') <- sum xs
                          ax' <- _prim_int_add x' s'
                          pure (CInt ax')
```

$$
\begin{array}{lll}
\texttt{t1} \to \{\,1\,\} & \texttt{m'} \to \{\,BAS\,\} & \texttt{l2} \to \{\,\texttt{CNil}[\,],\texttt{CCons}[\{1,5\},\{6\}]\,\} \\
\texttt{t2} \to \{\,2\,\} & \texttt{n'} \to \{\,BAS\,\} & \texttt{x} \to \{\,1,5\,\} \\
\texttt{t3} \to \{\,3\,\} & \texttt{b'} \to \{\,BAS\,\} & \texttt{xs} \to \{\,6\,\} \\
\texttt{t4} \to \{\,4\,\} & \texttt{m1'} \to \{\,BAS\,\} & \texttt{x'} \to \{\,BAS\,\} \\
\texttt{r'} \to \{\,BAS\,\} & \texttt{m1} \to \{\,5\,\} & \texttt{s'} \to \{\,BAS\,\} \\
\texttt{m} \to \{\,1,5\,\} & \texttt{p} \to \{\,6\,\} & \texttt{ax'} \to \{\,BAS\,\} \\
\texttt{n} \to \{\,2\,\} & \texttt{l} \to \{\,3,6\,\} &
\end{array}
$$

Fig. 9. Abstract environment of the analysis result.

$$
\begin{array}{lll}
1 \to \{\,\texttt{CInt}[\{BAS\}]\,\} & & \text{shared} \\
2 \to \{\,\texttt{CInt}[\{BAS\}]\,\} & & \text{shared} \\
3 \to \{\,\texttt{Fupto}[\{1\},\{2\}]\,\} & & \text{unique} \\
4 \to \{\,\texttt{Fsum}[\{3\}]\,\} & & \text{unique} \\
5 \to \{\,\texttt{CInt}[\{BAS\}]\,\} & & \text{shared} \\
6 \to \{\,\texttt{Fupto}[\{5\},\{2\}]\,\} & & \text{unique}
\end{array}
$$

Fig. 11. Abstract heap of the analysis result (*with* sharing analysis).

Optimisations

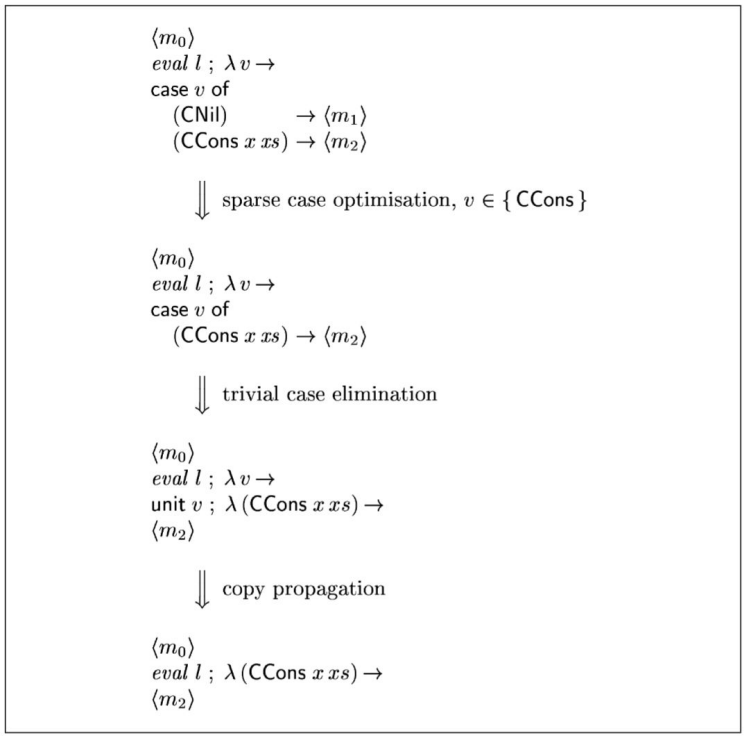


Figure 4.25: Sparse case optimisation.

Optimisations

$$
\begin{array}{l}
\langle m_0 \rangle \\
\mathit{foo}\ a_1\ \ldots\ a_n\ ;\ \lambda\,(\mathsf{CInt}\ y') \rightarrow \\
\langle m_1 \rangle
\end{array}
$$

$$\Downarrow \text{ Unbox } \mathit{foo} \mapsto \mathit{foo}'$$

$$
\begin{array}{l}
\langle m_0 \rangle \\
(\mathit{foo}'\ a_1\ \ldots\ a_n\ ;\ \lambda\, x' \rightarrow \\
\ \mathsf{unit}\ (\mathsf{CInt}\ x') \\
)\ ;\ \lambda\,(\mathsf{CInt}\ y') \rightarrow \\
\langle m_1 \rangle
\end{array}
$$

$$\Downarrow \text{ Bind normalisation + copy propagation}$$

$$
\begin{array}{l}
\langle m_0 \rangle \\
\mathit{foo}'\ a_1\ \ldots\ a_n\ ;\ \lambda\, x' \rightarrow \\
\langle m_1 \rangle \left[x'/y' \right]
\end{array}
$$

Figure 4.21: Unboxing of function return values.

$$
\begin{array}{l}
\mathit{foo}\ x\ y = \langle m_0 \rangle \\
\qquad\qquad \mathsf{fetch}\ y\ ;\ \lambda\,(\mathsf{Fbar}\ r\ s) \rightarrow \\
\qquad\qquad \langle m_1 \rangle
\end{array}
$$

$$\Downarrow \text{ arity raising}$$

$$
\begin{array}{l}
\mathit{foo}\ x\ a\ b = \langle m_0 \rangle \\
\qquad\qquad\ \mathsf{unit}\ (\mathsf{Fbar}\ a\ b)\ ;\ \lambda\,(\mathsf{Fbar}\ r\ s) \rightarrow \\
\qquad\qquad\ \langle m_1 \rangle
\end{array}
$$

Figure 4.33: The result of arity raising on a closure argument.

Optimisations

Local code transformations

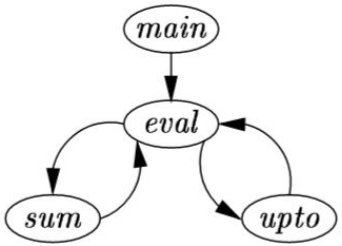
- Evaluated Case Elimination
- Trivial Case Elimination
- Update Elimination
- Copy Propagation
- Constant Propagation
- Dead Procedure Elimination
- Dead Parameter Elimination
- Case Copy Propagation

Whole program analysis based transformations

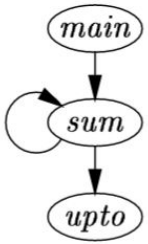
- Sparse Case Optimisation
- Dead Variable Elimination
- Dead Data Elimination
- Common Subexpression Elimination
- Case Hoisting
- Generalized Unboxing
- Arity Raising

Whole Program Analysis

Control Flow Analysis for lazy evaluation



Control Flow Graph:
GRIN - Strict



Control Flow Graph:
Thunks - Lazy

```
main = t1 <- store (CInt 1)
       t2 <- store (CInt 10000)
       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)
             #False -> m1' <- _prim_int_add m' 1
                       m1 <- store (CInt m1')
                       p <- store (Fupto m1 n)
                       pure (CCons m p)
sum l = l2 <- eval l
        case l2 of
          (CNil)       -> pure (CInt 0)
          (CCons x xs) -> (CInt x') <- eval x
                          (CInt s') <- sum xs
                          ax' <- _prim_int_add x' s'
                          pure (CInt ax')
```

Lazy control flow issue:
Compile-time prediction of run-time heap values

```
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
```

Control Flow Analysis: Heap Points-To

- Run-time heap can not be predicted, but can be approximated.
- Halting problem workaround: bounded heap.

Abstract Interpretation: program evaluation with alternative semantics.

Control Flow Analysis = Pointer Analysis with Abstract Interpretation

Heap Points-To Semantics:

- Tracking of node tag values through variables and bounded heap.
- Ignoring everything else.

Heap Points-To: Abstract Machine

Abstract Value:

- Node: Tag + arity sized vector of Abstract Values
- Pointer: Index of an Abstract Location
- Basic Abstract Symbol: Literals are mapped to the BAS symbol

Abstract Heap:

- Bounded set of abstract location { 1.. total number of stores in the program AST }.
- Each abstract location is associated to a single store operation in the program AST.
- Each abstract location stores a set of abstract values.

Abstract Environment:

- Mapping all variables to a set of abstract values.
- Mapping all functions to a set of abstract values that the function can return.

Heap Points-To Evaluation

Abstract GRIN evaluation design decisions (precision vs efficiency):

- No branching, every path is executed
- No function call, every function is executed
- Call site context-insensitive

Finding solution:

- Iterative solver
- Fixpoint conditions: monotonic, finite
- Initial state: empty heap, empty environment

The abstract program can be seen as data-flow graph

- Vertices: Variables and Heap locations
- Edges: Abstract operations

Heap Points-To Evaluation

HPT Abstract value operations:

- Create: constructs a constant value, i.e. `{BAS}` or `{CInt[]}`
- Merge: merges two values, i.e. `{CCons[{1},{6}]} + {CCons[{5},{6}]} = {CCons[{1,5},{6}]}`
- Project: projects the content of a given tag (or node item)
  I.e. `proj CCons from {CNil[], CCons[{1},{6}]} = {CCons[{1},{6}]}`
  `proj CInt from {CNil[]} = {}`

Abstract evaluation of GRIN constructs:

- Create: *pure value*, i.e. `pure (CNil)` evaluates to `{CNil[]}`
- Merge: *pure var*, *bind var*, *fetch*, *update*, *store*, i.e.
  `t1 <- store (CInt 1)` is evaluated as:
  - `heapValueOf(locationIndexOf(store)) += {CInt[{BAS}]}`
  - `t1 += locationIndexOf(store)`
- Project: *bind pattern*, *case*, i.e `(CInt a) <- pure x` evaluated as: `a += proj CInt from x`

Heap Points-To result

```
main = t1 <- store (CInt 1)
       t2 <- store (CInt 10000)
       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)
             #False -> m1' <- _prim_int_add m' 1
                       m1 <- store (CInt m1')
                       p <- store (Fupto m1 n)
                       pure (CCons m p)
sum l = l2 <- eval l
        case l2 of
          (CNil)       -> pure (CInt 0)
          (CCons x xs) -> (CInt x') <- eval x
                          (CInt s') <- sum xs
                          ax' <- _prim_int_add x' s'
                          pure (CInt ax')
```

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

Fig. 10. Abstract heap of the analysis result (*without* sharing analysis).

$$
\begin{array}{lll}
\texttt{t1} \rightarrow \{\,1\,\} & \texttt{m'} \rightarrow \{\,BAS\,\} & \texttt{l2} \rightarrow \{\, \texttt{CNil}[\,], \texttt{CCons}[\{1,5\},\{6\}] \,\} \\
\texttt{t2} \rightarrow \{\,2\,\} & \texttt{n'} \rightarrow \{\,BAS\,\} & \texttt{x} \rightarrow \{\,1,5\,\} \\
\texttt{t3} \rightarrow \{\,3\,\} & \texttt{b'} \rightarrow \{\,BAS\,\} & \texttt{xs} \rightarrow \{\,6\,\} \\
\texttt{t4} \rightarrow \{\,4\,\} & \texttt{m1'} \rightarrow \{\,BAS\,\} & \texttt{x'} \rightarrow \{\,BAS\,\} \\
\texttt{r'} \rightarrow \{\,BAS\,\} & \texttt{m1} \rightarrow \{\,5\,\} & \texttt{s'} \rightarrow \{\,BAS\,\} \\
\texttt{m} \rightarrow \{\,1,5\,\} & \texttt{p} \rightarrow \{\,6\,\} & \texttt{ax'} \rightarrow \{\,BAS\,\} \\
\texttt{n} \rightarrow \{\,2\,\} & \texttt{l} \rightarrow \{\,3,6\,\} &
\end{array}
$$

Fig. 9. Abstract environment of the analysis result.

Data-flow based type inference

Improve Heap Points-To analysis with better representation of literals

- Replace BAS with multiple Type Symbols (Int, Word, Float, etc.)
- The number of builtin types is finite

```
Location
  0   -> {CInt[T_Int64]}
  1   -> {CInt[T_Int64]}
  2   -> {Fupto[{0},{1}]}
  3   -> {Fsum[{2}]}
  4   -> {CInt[T_Int64]}
  5   -> {Fupto[{4},{1}]}
```

```
Variable
  a    -> {0,4}
  ax'  -> T_Int64
  b    -> {1}
  b'   -> T_Bool
  c    -> {2}
  l    -> {2,5}
  l2   -> {CCons[{0,4},{5}]
          ,CInt[T_Int64]
          ,CNil[]}
  ...
```

```
Function
  _prim_int_add   :: T_Int64 -> T_Int64 -> T_Int64
  _prim_int_gt    :: T_Int64 -> T_Int64 -> T_Bool
  _prim_int_print :: T_Int64 -> T_Unit
  eval :: {0,1,2,3,4,5}
       -> {CCons[{0,4},{5}]
          ,CInt[T_Int64]
          ,CNil[]}
  grinMain :: T_Unit
  sum  :: {2,5} -> {CInt[T_Int64]}
  upto :: {0,4} -> {1}
       -> {CCons[{0,4},{5}],CNil[]}
```

Compiled Abstract Interpretation

Abstract Interpretation has interpretational overhead by the host language.

Instead of direct interpretation, the GRIN program can be compiled to data-flow instructions.

Generic data-flow “virtual” machine

- The data-flow machine is capable to calculate any finite state.
- Operates on Int Sets that is efficient representation of finite values.
- Data-flow instructions can project or merge Int Sets.
- Instruction order does not have effect on the result.

Data-flow instruction evaluation

- Interpreted (convenient for debugging)
- Compiled (efficient, i.e. LLVM JIT, GPGPU, etc.)

Data-Flow Analysis Pipeline

Each Data-flow analysis is a code generator

- Compiles Data-flow program from GRIN AST
- Generates a mapping from analysis domain to integer values (finite <-> finite)

Usage:

- Compile GRIN to data-flow program with the analysis code generator
- Run data-flow machine
- Read-back the result using the domain mapping

Data-Flow Analysis

Analysis	Heap Points-To	Sharing	Created-By
Optimization	Sparse Case Optimization	Update Elimination	Dead Data Elimination
Description	Removes impossible cases	Removes useless updates	Removes unused data fields
Tracked property	Node tag	Variable linearity (bool)	Variable name of the value origin

E.g.: Created-By + Dead Data Elimination can transform the representation of List to Nat.

```
main = print $ length [1..1000]
```

replaces `(CCons x xs)` with `(CCons' xs)`

LLVM Backend for GRIN

GRIN to LLVM

LLVM features:

- Interprocedural optimizations
- Interprocedural register allocator
- Efficient machine code generator
- Supports many CPU architectures
- Excellent tooling
- Typed intermediate language
  (int, float, struct, array, pointer, ...)
- Bit-width precise types
- Vector types and operations (SIMD)
- Arbitrary number of virtual registers

Both GRIN and LLVM language is:
first order, strict, SSA, explicit memory operations

Issue: GRIN supports sum types, LLVM does not.

With the result of Heap Points-To analysis
it's easy to translate sum types to LLVM structs as tagged unions.

Variables are mapped to LLVM registers, other language constructs also have straightforward translation.

Garbage Collector Support:

- Urban Boquist's PhD thesis explains the details.
- Can be implemented with LLVM Stack Maps.

Demo: GRIN → LLVM → x64

GRIN

```
grinMain =
  t1 <- store (CInt 1)
  t2 <- store (CInt 100000)
  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 = ...
```

- Thunks on heap
- Heap operations

GRIN optimized

```
grinMain =
  n13 <- sum 0 1 100000
  _prim_int_print n13

sum n29 n30 n31 =
  b2 <- _prim_int_gt n30 n31
  case b2 of
    #True ->
      pure n29
    #False ->
      n18 <- _prim_int_add n30 1
      n28 <- _prim_int_add n29 n30
      sum n28 n18 n31
```

- Removed memory operations
- Transformed to strict loop

LLVM x64 codegen

```
grinMain:                               # @grinMain
# BB#0:                                 # %grinMain.entry
    movabsq    $5000050000, %rdi        # imm = 0x12A06B550
    jmp    _prim_int_print              # TAILCALL
```

- Constant folding
- Compile time evaluation

GHC Frontend for GRIN

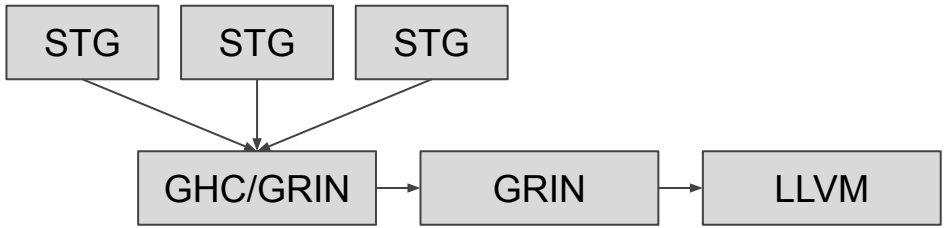
Haskell to GRIN

GRIN requires whole program compilation
Issue: GHC compiles modules

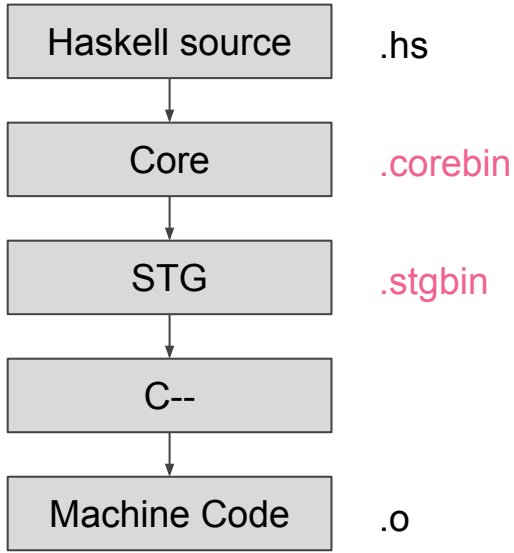
Modified GHC pipeline:

- Module compile phase: save core and stg
- Link phase: call an external core/stg linker with the program's core/stg modules

GHC/GRIN: STG to GRIN compiler



GHC module compilation pipeline



GHC/GRIN Status

Working:

- Modified GHC with whole program compilation support
- Sample project (with stack)

Current work:

- STG to GRIN translation (WIP)

Future work:

- Support GHC primops
  - GRIN interpreter (debug tool)
  - LLVM codegen
- Expose LLVM types and primops to Haskell
- Benchmark real-world programs

Questions

https://github.com/grin-tech/grin