GHC's Optimizer

Sylvain HENRY

GHC DevX – Learning Call
10 August 2023

GHC's optimizations

- Core optimizations: this presentation
- STG optimizations
- Cmm optimizations
- Asm optimizations
- Runtime optimizations (e.g. record selector opt in GC)

Other optimizations could be covered later.

Table of Contents

Core AST

```
data Expr b
  = Var      Id
  | Lit      Literal
  | App      (Expr b) (Arg b)
  | Lam      b (Expr b)
  | Let      (Bind b) (Expr b)
  | Case     (Expr b) b Type [Alt b]
  | Cast     (Expr b) CoercionR
  | Tick     CoreTickish (Expr b)
  | Type     Type
  | Coercion Coercion
```

References:

- 2022 - "Into the Core: Squeezing Haskell into ~~9~~ 10 constructors" (SPJ)

Core AST

```haskell
data Expr b
  = Var    Id
  | Lit    Literal
  | App    (Expr b) (Arg b)
  | Lam    b (Expr b)
  | Let    (Bind b) (Expr b)
  | Case   (Expr b) b Type [Alt b]
  | Cast   (Expr b) CoercionR
  | Tick   CoreTickish (Expr b)
  | Type   Type
  | Coercion Coercion
```

```haskell
data Alt b
  = Alt AltCon [b] (Expr b)

data AltCon
  = DataAlt DataCon
  | LitAlt  Literal
  | DEFAULT

data Bind b
  = NonRec b (Expr b)
  | Rec [(b, (Expr b))]

type Id          = Var
data Var         = ...
data Coercion    = ...
data Type        = ...
```

Rant: Core's 10 constructors

- Could we have less than 10 constructors?

Rant: Core's 10 constructors

- Could we have less than 10 constructors? Yes! One constructor!

```
data Expr = Expr Dynamic -- contains a Core datacon, promise

data Var = Var Id
data Lit = Lit Literal
...
```

Rant: Core's 10 constructors

- Could we have less than 10 constructors? Yes! One constructor!

```
data Expr = Expr Dynamic -- contains a Core datacon, promise

data Var = Var Id
data Lit = Lit Literal
...
```

- Hey! You're cheating! You've lost type safety!

Rant: Core's 10 constructors

- Could we have less than 10 constructors? Yes! One constructor!

```
data Expr = Expr Dynamic -- contains a Core datacon, promise

data Var = Var Id
data Lit = Lit Literal
...
```

- Hey! You're cheating! You've lost type safety!
- Yes, but it's already quite lost! Look at 'Id' (IdDetails, IdInfo):
  - We need shotgun parsing to handle ad-hoc cases:
    - Primops, data-con workers & wrappers, class ops, record selectors, covars, dfuns...
  - Sometimes they work the same, sometimes they don't. Good luck!

Rant: Core's 10 constructors

- Could we have less than 10 constructors? Yes! One constructor!

```
data Expr = Expr Dynamic -- contains a Core datacon, promise

data Var = Var Id
data Lit = Lit Literal
...
```

- Hey! You're cheating! You've lost type safety!
- Yes, but it's already quite lost! Look at 'Id' (IdDetails, IdInfo):
  - We need shotgun parsing to handle ad-hoc cases:
    - Primops, data-con workers & wrappers, class ops, record selectors, covars, dfuns...
  - Sometimes they work the same, sometimes they don't. Good luck!
- Take-away: Core optimization in practice is much trickier than it looks (e.g. in papers)
  - The compiler doesn't help much (cf shotgun parsing)
  - E.g. it doesn't tell you that you forgot to handle 'keepAlive#' or 'seq#' primops properly in your optimization pass or analysis
    - At best: missed optimization
    - At worst: bug!

Table of Contents

Core optimization pipeline

- Pipeline stages (a.k.a. CoreToDo)
  - Simplifier (many local transformations)
  - Worker-wrapper
  - Float-in and float-out
  - Full-laziness
  - + many other passes and analyses
- Phases: initial ("gentle"), 2, 1, 0, final*
  - Some rules and unfoldings (inlining) only enabled in some phases
  - Users can only specify 0-2 in pragmas
  - Final phase run repeatedly

References:

- GHC.Core.Opt.Pipeline
- Note [Compiler phases] in GHC.Types.Basic

Simple optimiser: simpler, alternative optimization pipeline

- Performs:
  - Occurrence analysis
  - Beta-reduction
  - Inlining
  - Case of known constructor
  - Dead code elimination
  - Coercion optimisation
  - Eta-reduction
  - ... more? (documentation is terrible)
- Used to simplify statically defined unfoldings, etc.
- Pure function: no stats, no dumps, etc.
- Do not confound "simple" with "simplifier"

References:

- GHC.Core.SimpleOpt

Simplifier

- Performs a bunch of local transformations
- Several simplifier iterations per phase of the optimization pipeline
  - Until fixpoint or N iterations (4 by default, set with ‘-fmax-simplifier-iterations‘)

References:

- GHC.Core.Opt.Simplify.*
  - "Utils" module contains optimizations too! e.g. see mkCase
- 1995 - "Compilation by transformation in non-strict functional languages" (Santos' thesis)

Simplifier: Santos' thesis 1/2

section	transformation	before	after
3.1	beta reduction	$(\lambda v.e)x$	$e[x/v]$
3.2.1	dead code removal	`let` v `=` e_v `in` e	e
3.2.2	inlining	`let` v `=` e_v `in` e	`let` v `=` e_v `in` $e[e_v/v]$
3.2.3	constructor reuse	`let` v `=` $C\ v_1 \ldots v_n$ `in let` w `=` $C\ v_1 \ldots v_n$ `in` e	`let` v `=` $C\ v_1 \ldots v_n$ `in let` w `=` v `in` e
3.3.1	case reduction	`case` $C_i\ v_1 \ldots v_n$ `of` $\ldots;\ C_i\ w_1 \ldots w_n$ `->` $e_i; \ldots$	$e_i[v_1/w_1 \ldots v_n/w_n]$
3.3.2	case elimination	`case` v_1 `of` v_2 `->` e	$e[v_1/v_2]$
3.3.3	case merging	`case` v `of` alt_1 `->` e_1 $\ldots$ d `-> case` v `of` alt_m `->` e_m $\ldots$	`case` v `of` alt_1 `->` e_1 $\ldots$ alt_m `->` $e_m[v/d]$ $\ldots$
3.3.5	default binding elimination	`case` v_1 `of` $\ldots;\ v_2$ `->` e	`case` v_1 `of` $\ldots;\ v_2$ `->` $e[v_1/v_2]$
3.4.1	let float from app	`(let` v `=` e_v `in` e`)` x	`let` v `=` e_v `in` $e\ x$
3.4.2	let float from let	`let` v `=let` w `=` e_w `in` e_v `in` e	`let` w `=` e_w `in let` v `=` e_v `in` e
3.4.3	let float from case scrutinee	`case (let` v `=` e_v `in` e`) of` $\ldots$	`let` v `=` e_v `in case` e `of` $\ldots$

Simplifier: Santos' thesis 2/2

3.5.1	case float from app	(case e_c of alt_1 -> e_1 ... alt_n -> e_n) v	case e_c of alt_1 -> e_1 v ... alt_n -> e_n v
3.5.2	case float from case (case of case)	case (case e_c of alt_{c1} -> e_{c1} ... alt_{cm} -> e_{cm}) of alt_1 -> e_1 ... alt_n -> e_n	case e_c of alt_{c1} -> case e_{c1} of alt_1 -> e_1 ... alt_n -> e_n ... alt_{cm} ->case e_{cm} of alt_1 -> e_1 ... alt_n -> e_n
3.5.3	case float from let	let v = case e_c of alt_{c1} -> e_{c1} ... alt_{cm} -> e_{cm} in e	case e_c of alt_{c1} -> let v = e_{c1} in e ... alt_{cm} -> let v = e_{cm} in e
3.6.1	let to case	let v = e_v in e	case e_v of v -> e
3.6.2	unboxing let to case	let v = e_v in e	case e_v of $C\ v_1 \ldots v_n$ -> let v = $C\ v_1 \ldots v_n$ in e
3.7.2	eta expansion	e	$\lambda x.e\ x$

Case of known constructor

- Also called "case reduction" (Santos)
- case C a b of { ...; C x y $\rightarrow$ e ;... } $\Longrightarrow$ e[a/x,b/y]
- Also applies to variable scrutinees which we know to be bound to a datacon
  - case C a b of v { .. case v of ... C x y $\rightarrow$ e ...}
  - let v = C a b in .. case v of ... C x y $\rightarrow$ e ...
- Made trickier by datacon wrappers that inline late... but for which we want to apply this optimization early
  - Inline wrapper on the fly. Some wrinkles (see Notes)

References:

- 1995 - "Compilation by transformation in non-strict functional languages" (Santos' thesis)

Let-floating: float-in, float-out (full-laziness)

- Float-in
  - float let-bindings closer to their use sites
  - reduce allocation scope
- Float-out (full-laziness)
  - float let-bindings towards the top-level
  - allow other optimizations to fire (without lets in their way)
  - increase sharing: risk of space leaks
  - static-forms (cd StaticPointers) always floated-out to the top-level
- Need to be careful with sharing, laziness, work duplication...

References:

- GHC.Core.Opt.{FloatIn,FloatOut}
- 1996 - "Let-floating: moving bindings to give faster programs" (SPJ et al)
- 1997 - "A transformation-based optimiser for Haskell" (SPJ, Santos)
- 2011 - (static pointers) "Towards Haskell in the Cloud" (SPJ et al)

Occurrence analyzer

Occurrence analyzer does much more than what it says on the tin!

- Occurrence analysis
- Dead let-binding elimination
- Strongly-Connected Component (SCC) analysis for let-bindings
- Loop-breaker selection in recursive let-bindings
- Join points detection
- Binder-swap

Occurrence analysis & dead let-binding elimination

- bottom-up traversal of an expression to annotate each variable binding with its usage:
  - how many times: 0, 1 (in different code paths or not), >1
  - in which context: in a lambda abstraction, in one-shot lambda...
- dead let-binding elimination
  - Done during the traversal
  - let b[dead] $=$ rhs in e $\Longrightarrow$ e
- Some accidental complexity for performance
  - ‘\x $\rightarrow$ \y $\rightarrow$ ... x ...‘ considered as ‘\x y $\rightarrow$ ... x ...‘ (x used once instead of inside a lambda; need to be careful with partial applications...)

References:

- 2002 - "Secrets of the GHC inliner" (SPJ, Marlow)
- GHC.Core.Opt.OccurAnal
  - Note [Dead code]

Binder-swap

```
(1)   case x of b { pi -> ri }
         ==>
      case x of b { pi -> ri[b/x] }

(2)   case (x |> co) of b { pi -> ri }
        ==>
      case (x |> co) of b { pi -> ri[b |> sym co/x] }
```

- Reduce number of occurrences of the scrutinee
- b has unfolding information in each alternative

Join point detection

- bottom-up traversal
  - track *always* tail-called variables (and their number of arguments)
  - update binding to say that it could become a join point
    - doesn't transform the binding itself
    - may need eta-expansion of the rhs and updates of the call sites (?)
    - Simplifier does the work of transforming identified let-bindings into join points
- Join points interact with occurrence analysis
  - Consider non-rec join points as if they were inlined, not as lets
    - Otherwise usage could be MultiOccs while it should be OneOcc (in several branches).
  - Only consider preexisting join points, not the candidates we discover in this pass.

References:

- Many notes in GHC.Core.Opt.OccurAnal
- 2017 - "Compiling without continuations" (SPJ, Maurer, Downen, Ariola)
  - Implementation doesn't fully follow the paper. See "join points" notes in GHC.Core

Dependency analysis

- Transform let-bindings into nest of:
  - Single let-binding (rec or non-rec)
  - Really recursive binding groups
- Select loop-breaker in recursive binding groups
  - Allow non-loop-breakers to be considered just like non-rec! Inline them, etc.
  - Loop-breaker selection: heuristics to rate bindings to find the least likely to be inlined
- Rules and unfoldings have to be taken into account
  - Rules' RHSs are considered as extra RHSs when doing dependency analysis
  - I.e. if we apply the rule, the free variables of the RHS should be well-scoped.

References:

- 2002 - "Secrets of the GHC inliner" (SPJ, Marlow)
- GHC.Core.Opt.OccurAnal
  - Note [Choosing loop breakers]
  - Note [Rules are extra RHSs]

Beta-reduction

- $(\backslash x \rightarrow e)\ b \Longrightarrow$ let $x = b$ in e
- Useful because let-binding ensures there is a rhs
  - With lambda application, we have to look outside
  - E.g. consider: $(\backslash x \rightarrow \backslash y \rightarrow \backslash z \rightarrow e)\ a\ b\ c$
  - Better to beta-reduce then float-out the let-binding

References:

- 2002 - "Secrets of the GHC inliner" (SPJ, Marlow)
- 1987 - "The implemetnation of functional programming languages" (SPJ)

Inlining

- Happens after occurrence analysis
  - Binding variables are annotated with occurrence info:
    - how many occurrences: 0, 1 (in different code paths or not), >1
    - in which context: in a lambda abstraction...
- pre-inline-unconditionally
  - Inline unoptimized RHS for bindings used once (and not in a lambda...)
- post-inline-unconditionally
  - Optimize E into E' in ‘let x = E in ...‘
  - Inline E' if
    - x isn't exported, nor a loop-breaker
    - E' is trivial
- call-site-inline (not done by the simple optimiser)
  - Just keep ‘let x = E' in ...‘
  - At each occurrence of ‘x‘, consider inlining it or not

References:

- 2002 - "Secrets of the GHC inliner" (SPJ, Marlow)

Coercion optimization

- Coercion: proof term that ‘foo $\sim$ bar‘ (this is its type)
  - Coercion ADT in GHC.Core.TyCo.Rep: Refl, Sym, Trans...
- Optimization
  - E.g. sym (sym c) $\Longrightarrow$ c
  - Can get much more complex
  - Especially with coercion roles for equality:
    - Nominal (Haskell type equality)
    - Representational ($\sim$ coercible equality, e.g. newtype)
    - Phantom (can always be made equal? Perhaps not with different kinds)

References:

- GHC.Core.Coercion.Opt
- 2007 - (coercions) "System F with Type Equality Coercions" (Sulzmann et al)
- 2011 - (roles) "Generative Type Abstraction and Type-level Computation" (Weirich et al)
- 2013 - (opt) "Evidence normalization in System FC" (SPJ, Vytiniotis)

Rewrite rules

- Rewrite an expression into another
  - User-provided rules (RULES pragmas, eg. foldr/build fusion)
  - Built-in rules (e.g. constant folding)
  - Compiler-generated rules (e.g. for specialisation)
- Add quite a lot of internal complexity
  - User-provided rules' LHS expressed using surface syntax. LHS lowered into Core.
  - Other optimisations then get in the way of rule application (worker wrapper, etc.) $\rightarrow$ phases
  - Rules RHS are alternate RHS for functions: they can mess up with dependency analysis and occurrence analysis.

References:

- 2001 - "Playing by the rules: rewriting as a practical optimisation technique" (SPJ et al)

Constant folding

- Rewrite rules for primops
  - Actually implemented as built-in rewrite rules
    - Some might be implemented with RULES: e.g. (+#) x 0# $\Longrightarrow$ x
    - Some can't: (+#) lit1 lit2 $\Longrightarrow$ lit3
  - I've added many more complex rules for nested expressions:
    - e.g. (x - lit1) - (y - lit2) $\Longrightarrow$ (lit2-lit1) + (x-y)

Constant folding

- Rewrite rules for primops
  - Actually implemented as built-in rewrite rules
    - Some might be implemented with RULES: e.g. (+#) x 0# $\Longrightarrow$ x
    - Some can't: (+#) lit1 lit2 $\Longrightarrow$ lit3
  - I've added many more complex rules for nested expressions:
    - e.g. (x - lit1) - (y - lit2) $\Longrightarrow$ (lit2-lit1) + (x-y)
- Rewrite rules for Integer/Natural/Bignat
  - I have been trying to only keep Bignat rules
    - So that only Bignat values are opaque (and ghc can unpack Int# or Bignat from Integer)
    - WIP because: rewrite rules + unlifted types = issues

Constant folding

- Rewrite rules for primops
  - Actually implemented as built-in rewrite rules
    - Some might be implemented with RULES: e.g. (+#) x 0# $\Longrightarrow$ x
    - Some can't: (+#) lit1 lit2 $\Longrightarrow$ lit3
  - I've added many more complex rules for nested expressions:
    - e.g. (x - lit1) - (y - lit2) $\Longrightarrow$ (lit2-lit1) + (x-y)
- Rewrite rules for Integer/Natural/Bignat
  - I have been trying to only keep Bignat rules
    - So that only Bignat values are opaque (and ghc can unpack Int# or Bignat from Integer)
    - WIP because: rewrite rules + unlifted types = issues
- Scrutinee constant folding
  - e.g. case x - 1 of y { 5 $\rightarrow$ ..; ..} $\Longrightarrow$ case x of y' { 6 $\rightarrow$ let y = 5 in ..; ..}
  - I've implemented these for my Variant package
    - case x - 1 of y { 0 $\rightarrow$ ..; _ $\rightarrow$ case y - 1 of z { 0 $\rightarrow$ ..; _ $\rightarrow$ case z - 1 of ...
    - $\Longrightarrow$ case x of { 0 $\rightarrow$ ..; 1 $\rightarrow$ ..; 2 $\rightarrow$ ..; .. }

Constant folding

- Rewrite rules for primops
  - Actually implemented as built-in rewrite rules
    - Some might be implemented with RULES: e.g. (+#) x 0# $\Longrightarrow$ x
    - Some can't: (+#) lit1 lit2 $\Longrightarrow$ lit3
  - I've added many more complex rules for nested expressions:
    - e.g. (x - lit1) - (y - lit2) $\Longrightarrow$ (lit2-lit1) + (x-y)
- Rewrite rules for Integer/Natural/Bignat
  - I have been trying to only keep Bignat rules
    - So that only Bignat values are opaque (and ghc can unpack Int# or Bignat from Integer)
    - WIP because: rewrite rules + unlifted types = issues
- Scrutinee constant folding
  - e.g. case x - 1 of y { 5 $\rightarrow$ ..; ..} $\Longrightarrow$ case x of y' { 6 $\rightarrow$ let y = 5 in ..; ..}
  - I've implemented these for my Variant package
    - case x - 1 of y { 0 $\rightarrow$ ..; _ $\rightarrow$ case y - 1 of z { 0 $\rightarrow$ ..; _ $\rightarrow$ case z - 1 of ...
    - $\Longrightarrow$ case x of { 0 $\rightarrow$ ..; 1 $\rightarrow$ ..; 2 $\rightarrow$ ..; .. }
- Rewrite rule for GHC.Magic.inline, seq, cstringLength, tagToEnum, etc.

References:

- GHC.Core.Opt.ConstantFold

Specialisation (type-classes)

- Idea
  - Find calls to polymorphic functions with type-class arguments
  - Generate specialized versions of the polymorphic functions and call them instead
- Works inside a module
- Also works for INLINABLE bindings from other modules
  - Disable with -fno-cross-module-specialize
- SPECIALIZE pragma
  - User directed specialization
  - Generate a rewrite rule to replace appropriate call sites

References:

- GHC.Core.Opt.Specialise (SPJ's notes, 1993)

Exitification

- Idea: allow inlining into "exit paths" of recursive functions by transforming them into join points

```
let t = foo bar
joinrec go 0     x y = t (x*x)
        go (n-1) x y = jump go (n-1) (x+y)
in ...
```

==>

```
let t = foo bar
join exit x = t (x*x)
joinrec go 0     x y = jump exit x
        go (n-1) x y = jump go (n-1) (x+y)
in ...
```

- Exit join points mustn't be inlined for this to work (ad-hoc)
- IMO it would be better to detect that t occurs once in an ExitJoinPoint in OccurAnal

References:

- GHC.Core.Opt.Exitify

Common subexpression (CSE)

- Idea: common up bindings with the same RHS / cases with the same scrutinee
- Issues:
  - Rules attached to one binding but not the other
  - (NO)INLINE attached to one binding but not the other
  - Bindings with stable unfoldings
    - RHSs look the same now, but inlined they would be different
    - But in some cases we don't care (e.g. worker-wrapper)...
  - Join points: can't cse join points in non tail position; musn't mess with join point arity by CSEing join points' lambdas...
  - Top-level unboxed strings: can't be variables... special case!
  - Ticks: we strip some ticks to CSE more but then it's bogus
  - Careful: CSE changes occurrence info and potentially demand info!
    - We zap them: we could probably merge them but it would require another pass to fix the occurrences...
- CSE self-recursive bindings by rewriting them as for use with 'fix'

References:

- GHC.Core.Opt.CSE

Liberate case

- Idea: unroll (inline) a recursive function once into itself
- When: when it scrutinizes a free variable before the recursive calls
- Why: in the inlined code, the free variable is already evaluated, split apart, etc.

```
   f = \ t -> case v of
                V a b -> a : f t

=> the inner f is replaced.

   f = \ t -> case v of
                 V a b -> a : (letrec
                               f = \ t -> case v of
                                            V a b -> a : f t
                              in f) t
=> Simplify

  f = \ t -> case v of
                 V a b -> a : (letrec
                               f = \ t -> a : f t
                              in f t)
```

References:

- GHC.Core.Opt.LiberateCase

SpecConst: specialise over constructors

- Idea: specialize a function depending on arguments' constructors

```
last :: [a] -> a
last [] = error "last"
last (x : []) = x
last (x : xs) = last xs
```

$\Longrightarrow$

```
last [] = error "last"
last (x:xs) = last' x xs
  where
    last' x [] = x
    last' x (y:ys) = last' y ys
```

References:

- GHC.Core.Opt.SpecConst
- 2007 - "Call-pattern specialisation for Haskell programs" (SPJ)

Demand analysis / strictness analysis

- Idea: annotate functions with the way they use their arguments
- E.g. not at all, strictly, etc.
- Also sub-demands for fields of datacons
- Why: caller can pass arguments by value, worker-wrapper, etc.

References:

- GHC.Core.Opt.DmdAnal
- 2017 - "Theory and practice of demand analysis in Haskell (draft)" (SPJ et al)

CPR analysis

- CPR: Constructed Product Result
- Idea: if a function always build a result "C a b c", just return a, b, and c and let the caller allocate C (if needed)

References:

- GHC.Core.Opt.CprAnal
- 2004 - "Constructed Product Result analysis for Haskell" (SPJ et al)
- Sebastian Graf

Worker-wrapper

- Idea: after demand analysis and CPR analysis, split functions into wrapper/worker
- Wrapper
  - evaluates and unboxes strict arguments
  - call worker
  - allocate CPR results
- Worker
  - Perform the useful work
- The hope is that the wrapper get inlined to be optimized

References:

- GHC.Core.Opt.WorkWrap
- previous papers about demand analysis and CPR analysis

Call arity, eta-reduction, eta-expansion

- Idea: find the "real" arity of a function
- E.g. f a b = \c → e
  - f doesn't do any work until applied to at least 3 arguments

References:

- GHC.Core.Opt.CallArity
- 2016 - Joachim Breitner's doctoral thesis

SAT: Static Argument Transformation

- Idea: avoid passing always the same arg to recursive calls

```
foo f n = case n of
  0 -> []
  _ -> f n : foo f (n-1)

==>

foo f n =
  let foo' = case n of
             0 -> []
             _ -> f n : foo' (n-1)
  in foo' n
```

- Side-effect: ‘foo‘ may no longer be recursive and may be inlined!
- Now always allocate a closure (e.g. "foo'")

References:

- GHC.Core.Opt.StaticArgs
- Santos' thesis

Remarks on Core optimizations

- Fragile because some semantic constructs are hidden
  - As primop application: seq, keepAlive
  - As let-bindings: join points, unlifted/unboxed lets
- Quite intricate
  - Doing some misoptimization because we know there is something else happening later in the pipeline to fix it
  - Rely on mutable state: demand-info, occur-info...
  - Mix optimization for performance:
    - E.g. specialisation triggers rule rewriting