Into the Core
Squeezing Haskell into nine constructors

Simon Peyton Jones

Microsoft Research

September 2016

CACM Jan 2016

DOI:10.1145.2844112

Article development led by acmqueue
queue.acm.org

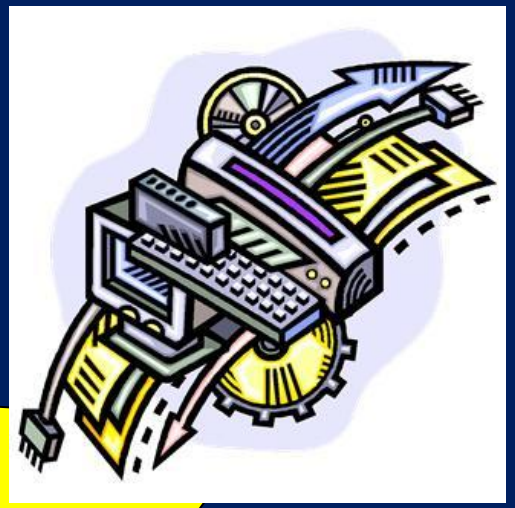
We need it, we can afford it, and the time is now.

BY PAT HELLAND

Immutability Changes Everything

GHC Haskell

A **rich** language



GHC Haskell

A very complicated and ill-defined language, with a long user manual, that almost no one understands completely

GHC is big and old

Module	Lines (1992)	Lines (2011)	Increase
Compiler			
Main	997	11,150	11.2
Parser	1,055	4,098	3.9
Renamer	2,828	4,630	1.6
Type checking	3,352	24,097	7.2
Desugaring	1,381	7,091	5.1
Core tranformations	1,631	9,480	5.8
STG transformations	814	840	1
Data-Parallel Haskell	—	3,718	—
Code generation	2913	11,003	3.8
Native code generation	—	14,138	—
LLVM code generation	—	2,266	—
GHCi	—	7,474	—
Haskell abstract syntax	2,546	3,700	1.5
Core language	1,075	4,798	4.5
STG language	517	693	1.3
C-- (was Abstract C)	1,416	7,591	5.4
Identifier representations	1,831	3,120	1.7
Type representations	1,628	3,808	2.3
Prelude definitions	3,111	2,692	0.9
Utilities	1,989	7,878	3.96
Profiling	191	367	1.92
Compiler Total	28,275	139,955	4.9
Runtime System			
All C and C-- code	43,865	48,450	1.10

Figure 1: Lines of code in GHC, past and present

GHC is big and old

Module	Lines (1992)	Lines (2011)	Increase
Compiler			
Main	997	11,150	11.2
Parser	1,055	4,098	3.9
Renamer	2,828	4,630	1.6
Type checking	3,352	24,097	7.2
Desugaring	1,381	7,091	5.1
Core tranformations	1,631		
STG transformati			
			2.3
	3,111	2,692	0.9
Utilities	1,989	7,878	3.96
Profiling	191	367	1.92
Compiler Total	28,275	139,955	4.9
Runtime System			
All C and C-- code	43,865	48,450	1.10

Figure 1: Lines of code in GHC, past and present

Question

how to stay sane?

How GHC works

Source language

Haskell

Massive language
Hundreds of pages of user manual
Syntax has dozens of data types
100+ constructors

Typecheck

Desugar

Typed intermediate language

Core

3 types,
15 constructors

Rest of GHC

A typed intermediate language

Haskell	Core (the typed IL)
Big	Small
Implicitly typed	Explicitly typed
Binders typically un-annotated \x. x && y	Every binder is type-annotated \(x:Bool). x && y
Type inference (complex, slow)	Type checking (simple, fast)
Complicated to specify just which programs will type-check	Very simple to specify just which programs are type-correct
Ad-hoc restrictions to make inference feasible	Very expressive indeed; simple, uniform

A typed intermediate language

\x. x && y	\(x:Bool). x && y
Type inference (complex, slow)	Type checking (simple, fast)
Complicated to specify just which programs will type-check	Very simple to specify just which programs are type-correct
Ad-hoc restrictions to make inference feasible	Very expressive indeed; simple, uniform

A typed intermediate language: why?

1. Small IL means that analysis, optimisation, and code generation, handle only a small language.
2. Type checker ("Lint") for Core is a very powerful internal consistency check on most of the compiler
   - Desugarer must produce well-typed Core
   - Optimisation passes must transform well-typed Core to well-typed Core
3. Design of Core is a powerful sanity check on crazy type-system extensions to source language. If you can desugar it into Core, it must be sound; if not, think again.

A typed intermediate language: why?

1. Small IL means that analysis, optimisati… de generation, handle only a smal…
2. Type checker (… ternal consistency check on most …
   - Desugarer mus…
   - Optimisation pa… ll-typed Core
3. Design of Core is… pe-system extensions to sou… into Core, it must be sound; if…

GHC is the only production compiler that remorselessly pursues this idea of a strongly-typed intermediate language

The design of Core is probably GHC's single most substantial technical achievement

Why should you care about Core?

- Because it is jolly interesting
- Because much of it would work for Erlang
- Because it shows theory and practice working together

WHAT SHOULD CORE BE LIKE?

What should Core be like?

- Start with lambda calculus. From "Lambda the Ultimate X" papers we know that lambda is super-powerful.
- But we need a TYPED lambda calculus
- Idea:
  - start with lambda calculus
  - sprinkle type annotations
- But:
  - Don't want to be buried in type annotations
  - Types change as you optimise

Example

```
compose :: (b->c) -> (a->b) -> a -> c
compose = λ(f:b->c). λ(g:a->b). λ(x:a).
          let tmp:b = g x
          in f tmp
```

- Idea: put type annotations on each binder (lambda, let), but nowhere else
- But: where are 'a' 'b' 'c' bound?
- And: unstable under transformation...

Example

```
compose :: (b->c) -> (a->b) -> a -> c
compose = λf:b->c. λg:a->b. λx:a.
            let tmp:b = g x
            in f tmp
```

```
neg :: Int -> Int
isPos :: Int -> Bool
```

```
compose isPos neg
=       (inline compose:
        f=isPos, g=neg)
    λx:a. let tmp:b = neg x
          in isPos tmp
```

- Now the type annotations are wrong
- Solution: learn from Girard and Reynolds!

System F

```
compose :: ∀abc. (b->c) -> (a->b) -> a -> c
compose = Λabc. λf:b->c. λg:a->b. λx:a.
          let tmp:b = g x
          in f tmp
```

- Idea: an explicit (big) lambda binds type variables

System F

```
compose :: ∀abc. (b->c) -> (a->b) -> a -> c
compose = Λabc. λf:b->c. λg:a->b. λx:a.
            let tmp:b = g x
            in f tmp
```

```
compose @Int @Int @Bool isPos neg
=   (inline compose:
     a=Int, b=Int, c=Bool, f=isPos, g=neg)
   λx:Int. let tmp:Int = neg x
           in isPos tmp
```

- Big lambdas are applied to types,
  just as little lambdas are applied to values
- Now the types stay correct!

The real "System F"

- In GHC, the IL is like what we've seen, plus:
  - Algebraic data type declarations

```
data Maybe a = Nothing | Just a
```

  - Data constructors in terms

```
λx:Int. Just (Just x)
```

  - Case expressions

```
case x of { Nothing -> 0; Just x -> x+1 }
```

  - Let expressions

```
let x:Int = 4 in x+x
```

Core: GHC's intermediate language

```
data Expr
  = Var       Var
  | Lit       Literal
  | App       Expr Expr
  | Lam       Var Expr  -- Both term and type lambda
  | Let       Bind Expr
  | Case      Expr Var Type [(AltCon, [Var], Expr)]
  | Type      Type       -- Used for type application

data Var = Id    Name Type  -- Term variable
         | TyVar Name Kind  -- Type/kind variable

type Kind = Type
data Type = TyVarTy  Var
    | LitTy    TyLit
    | AppTy    Type Type
    | TyConApp TyCon [Type]
    | FunTy    Type Type    -- Not really necy
    | ForAllTy Var Type
```

Core: GHC's intermediate language

```
  | Case     Expr Var Type [(AltCon, [Var], Expr)]
  | Type     Type       -- Used for type application

data Var = Id    Name Type  -- Term variable
         | TyVar Name Kind  -- Type/kind variable

type Kind = Type
data Type = TyVarTy  Var
   | LitTy    TyLit
   | AppTy    Type Type
   | TyConApp TyCon [Type]
   | FunTy    Type Type   -- Not really necy
   | ForAllTy Var Type
```

Core: GHC's intermediate language

```
data Expr
  = Var       Var
  | Lit       Literal
  | App       Expr Expr
  | Lam       Var Expr  -- Both term and type lambda
  | Let       Bind Expr
  | Case      Expr Var Type [(AltCon
  | Type      Type        -- Used

data Var = Id
         | T

type
data
    | L
    | AppTy     Type Type
    | TyConApp  TyCon [Type]
    | FunTy     Type Type    -- Not really necy
    | ForAllTy Var Type
```

26 years old and still tiny.
Bravo Girard & Reynolds!

What's good about System F

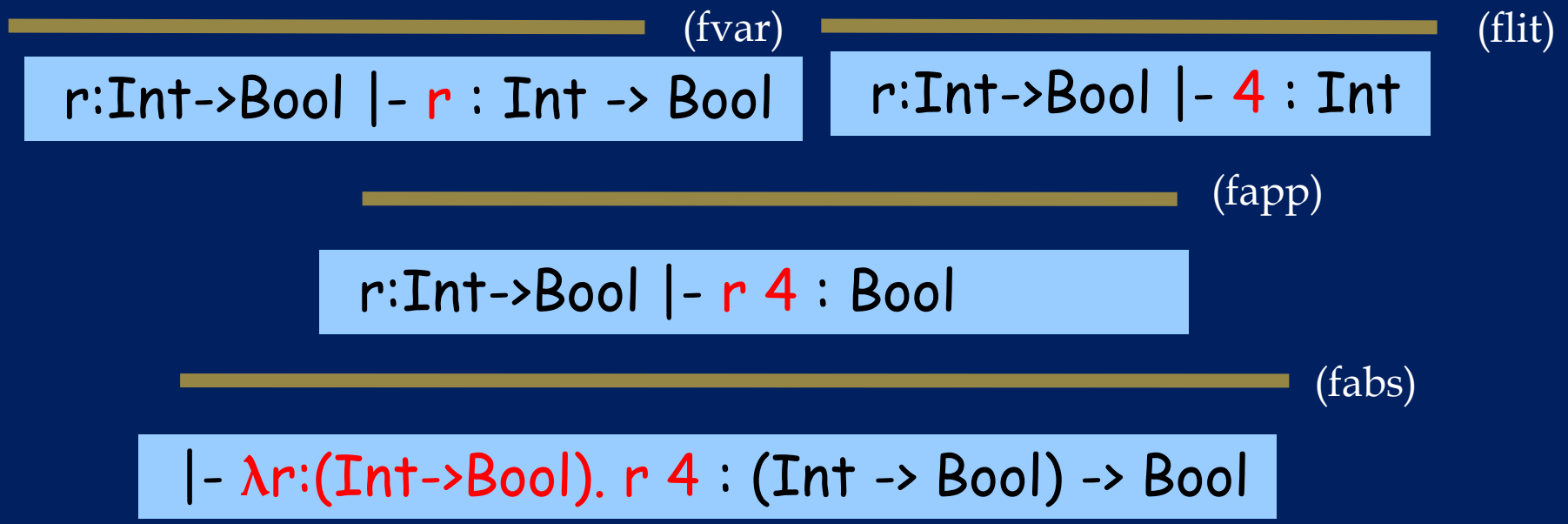
- In our presentation of System F, each variable occurrence is annotated with its type.
- Hence every term has a unique type

```
exprType :: Expr -> Type
exprType (Var v)   = varType v
exprType (Lam v a) = Arrow (varType v) (exprType a)
...more equations...
```

- `exprType` is pure; needs no "Gamma" argument
- Sharing of the `Var` means that the apparent duplication is not real

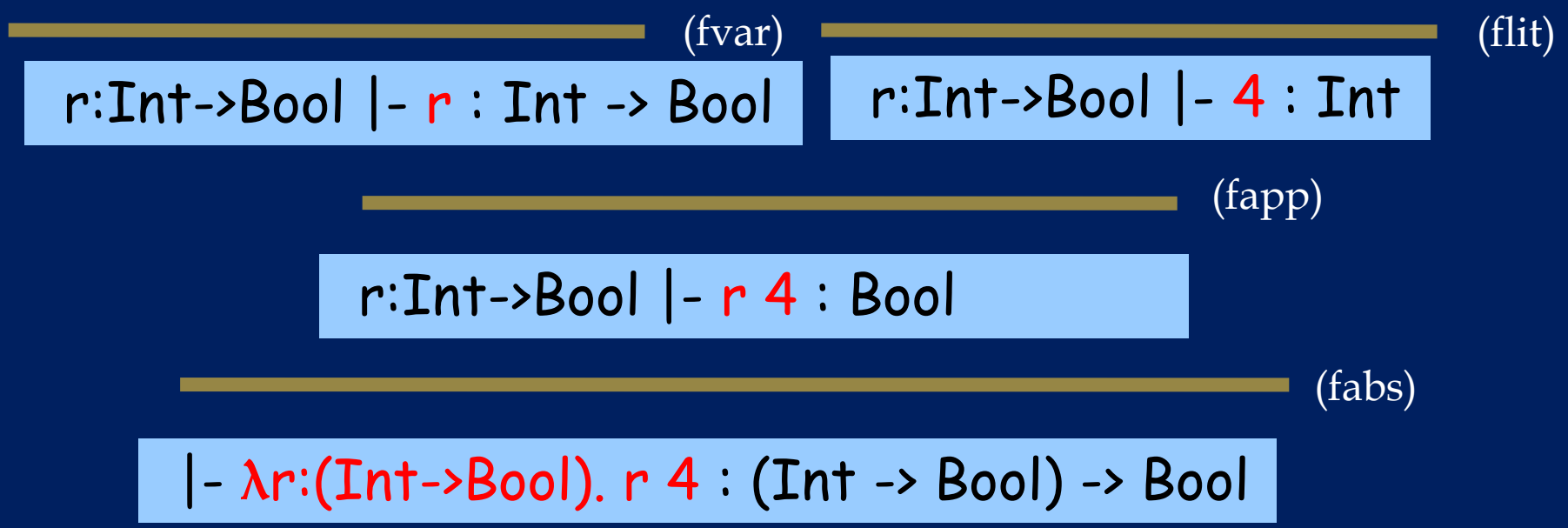
What's good about System F?

Type checking (Lint) is fast and easy, because the rules are syntax-directed



What's good about System F?

Type checking (Lint) is fast and easy, because the rules are syntax-directed

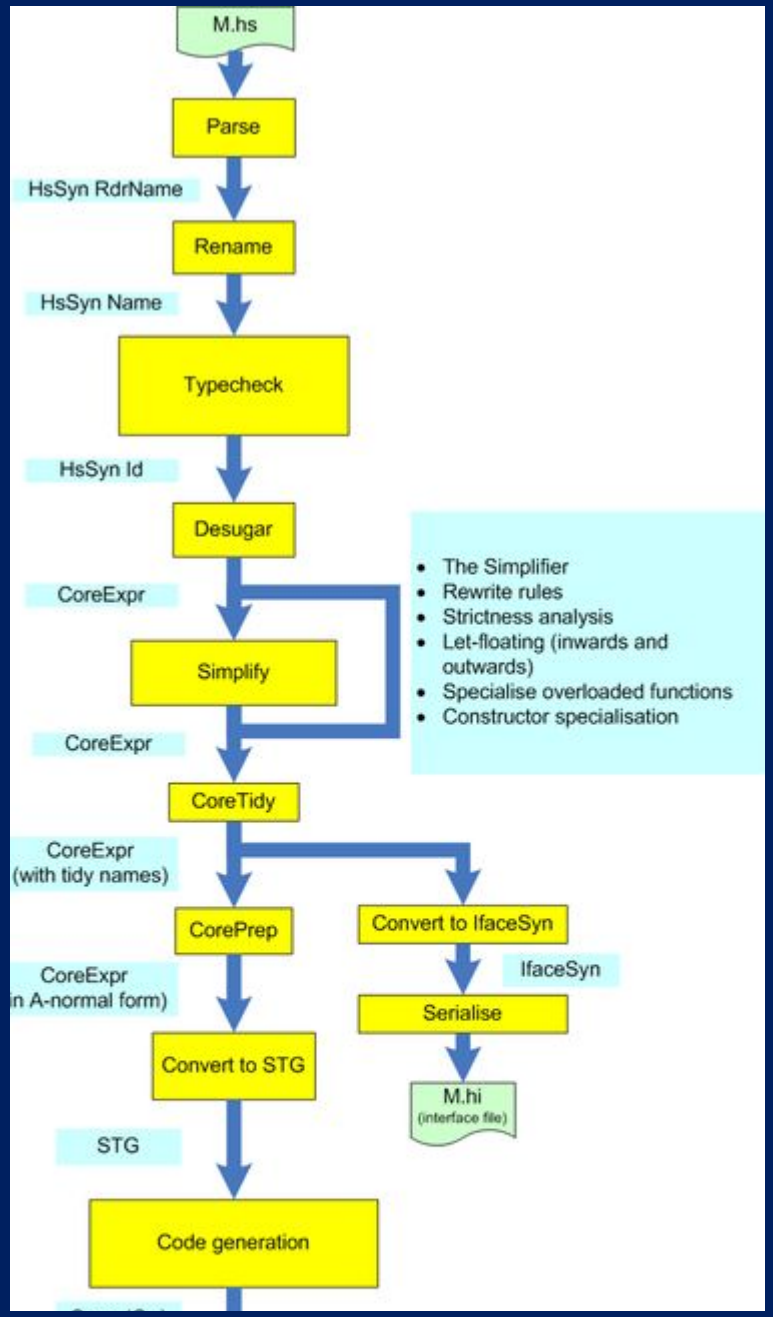


The syntax of a term encodes its typing derivation

Story so far

- Very small, statically typed language
- Robust to transformations (ie if the term is well typed, then the transformed term is well typed)
- Simple, pure **`exprType`**
- Type checking (Lint) is easy and fast

THE CORE PIPELINE



- The Simplifier
  - Inlining
  - Rewrite rules
  - Beta reduction
  - Case of case
  - Case of known constructor
  - etc etc etc

- Specialise overloading
- Float out
- Float in
- Demand, cardinality, and CPR analysis
- Arity analysis
- Call-pattern specialisation (SpecConstr)

```
module Foo where
   f :: Int -> Int
   f x = x+1
```

```
bash$ ghc -c Foo.hs -ddump-simpl
==================== Tidy Core ====================
Result size of Tidy Core
   = {terms: 7, types: 4, coercions: 0}

Foo.f :: GHC.Types.Int -> GHC.Types.Int
[GblId, Arity=1, Str=DmdType]
Foo.f =
  \ (x_apE :: GHC.Types.Int) ->
    GHC.Num.+ @ GHC.Types.Int
              GHC.Num.$fNumInt
              x_apE (GHC.Types.I# 1)
```

```
bash$ ghc -c Foo.hs -dshow-passes
*** Checking old interface for main:Foo:
*** Parser:
*** Renamer/typechecker:
*** Desugar:
Result size of Desugar (after optimization)
  = {terms: 7, types: 4, coercions: 0}
*** Simplifier:
Result size of Simplifier = {terms: 7, types: 4, coercions: 0}
*** Tidy Core:
Result size of Tidy Core = {terms: 7, types: 4, coercions: 0}
*** CorePrep:
Result size of CorePrep = {terms: 9, types: 5, coercions: 0}
*** Stg2Stg:
*** CodeOutput:
*** New CodeGen:
```

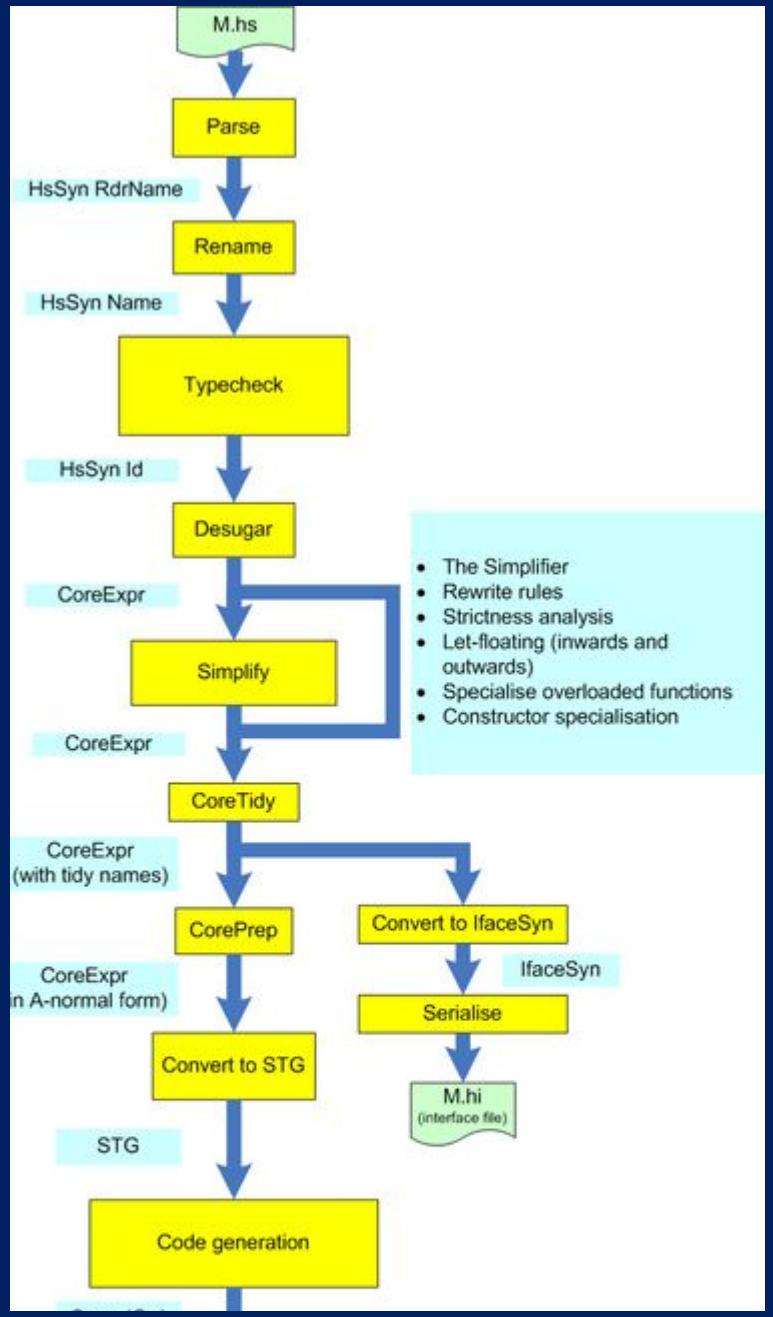
```
bash$ ghc -c Foo.hs -ddump-simpl -O
==================== Tidy Core ====================
Result size of Tidy Core
    = {terms: 9, types: 5, coercions: 0}

Foo.f :: GHC.Types.Int -> GHC.Types.Int
[GblId, Arity=1, Caf=NoCafRefs,
 Str=DmdType <S,1*U(U)>m,
 Unf=Unf{Src=InlineStable, TopLvl=True,
         Arity=1, Value=True,
         ConLike=True, WorkFree=True, Expandable=True,
         Guidance=ALWAYS_IF(unsat_ok=True,
                            boring_ok=False)
         Tmpl= ...]
Foo.f =
  \ (x_arV :: GHC.Types.Int) ->
    case x_arV of _ [Occ=Dead] { GHC.Types.I# x1_aNv ->
    GHC.Types.I# (GHC.Prim.+# x1_aNv 1)
    }
```

```
bash$ ghc -c Foo.hs -O -dshow-passes
*** Parser:
*** Renamer/typechecker:
*** Desugar:
Result size of Desugar (after optimization)
  = {terms: 7, types: 4, coercions: 0}
*** Simplifier:
Result size of Simplifier iteration=1
  = {terms: 6, types: 3, coercions: 0}
Result size of Simplifier = {terms: 6, types: 3, coercions: 0}
*** Specialise:
Result size of Specialise = {terms: 6, types: 3, coercions: 0}
*** Float out(FOS {Lam = Just 0, Consts = True, PAPs = False}):
Result size of Float out(FOS {Lam = Just 0,Consts = True,PAPs = False})
  = {terms: 8, types: 4, coercions: 0}
*** Float inwards:
Result size of Float inwards = {terms: 8, types: 4, coercions: 0}
*** Simplifier:
Result size of Simplifier iteration=1
  = {terms: 12, types: 6, coercions: 0}
Result size of Simplifier = {terms: 9, types: 5, coercions: 0}
*** Simplifier:
Result size of Simplifier = {terms: 9, types: 5, coercions: 0}
*** Simplifier:
Result size of Simplifier = {terms: 9, types: 5, coercions: 0}
*** Demand analysis:
Result size of Demand analysis = {terms: 9, types: 5, coercions: 0}
*** Worker Wrapper binds:
Result size of Worker Wrapper binds
  = {terms: 9, types: 5, coercions: 0}
*** Simplifier:
Result size of Simplifier = {terms: 9, types: 5, coercions: 0}
*** Float out(FOS {Lam = Just 0, Consts = True, PAPs = True}):
Result size of Float out(FOS {Lam = Just 0,Consts = True, PAPs = True})
  = {terms: 9, types: 5, coercions: 0}
*** Common sub-expression:
Result size of Common sub-expression
  = {terms: 9, types: 5, coercions: 0}
*** Float inwards:
Result size of Float inwards = {terms: 9, types: 5, coercions: 0}
*** Simplifier:
Result size of Simplifier = {terms: 9, types: 5, coercions: 0}
*** Tidy Core:
Result size of Tidy Core = {terms: 9, types: 5, coercions: 0}
*** CorePrep:
Result size of CorePrep = {terms: 12, types: 6, coercions: 0}
*** Stg2Stg:
*** CodeOutput:
*** New CodeGen:
```

CORE OPTIMISATIONS

A hail of bullets



- The Simplifier
  - Inlining
  - Rewrite rules
  - Beta reduction
  - Case of case
  - Case of known constructor
  - etc etc etc

- Specialise overloading
- Float out
- Float in
- Demand, cardinality, and CPR analysis
- Arity analysis
- Call-pattern specialisation (SpecConstr)

The mighty simplifier

- Simplifier performs a raft of local optimisations
  - Inlining
  - Beta reduction
  - Rewrite rules
  - Case of known constructor
  - Case of case
  - etc etc etc
- These transformations cascade

LET: Inlining

- Inlining is **the** key transformation
- Replace x by x's definition

```
let x = <rhs> in …x…x…
    ===>
let x = <rhs> in …x…<rhs>…
```

- Issues:
  - May duplicate code <rhs>
    How often does 'x' occur? How big is <rhs>?
  - May duplicate work let x = factorial y in …
    Is <rhs> a value or not?
  - Which occurrences to substitute for.
    All? Some? None?

Lots of heuristics here

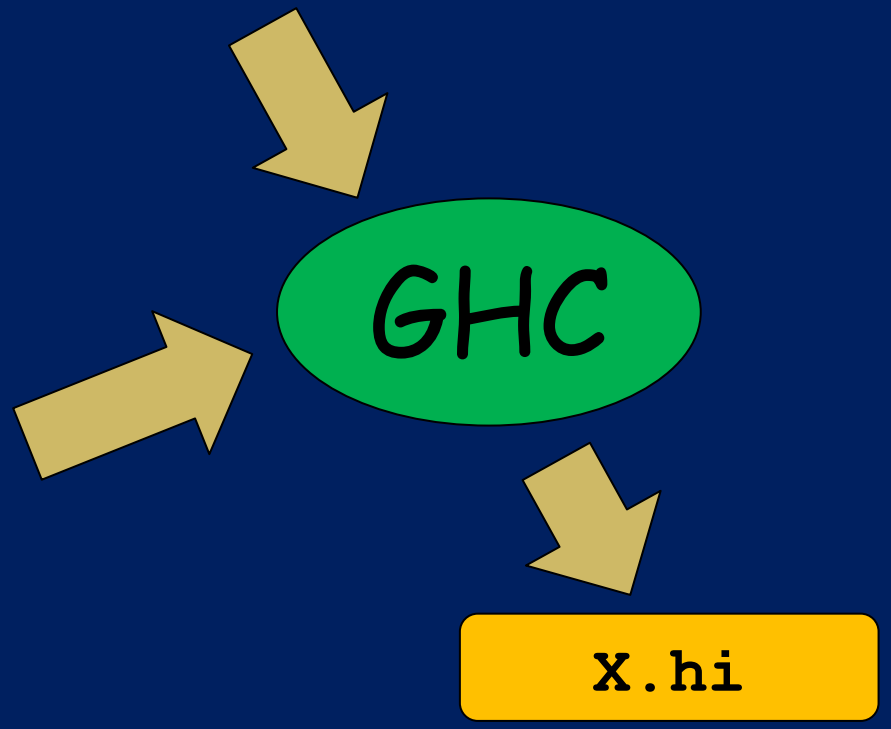
LET: Inlining

- Works even if x's definition is in another module, or another package

```
module Prelude where
  null :: [a] -> Bool
  null [] = True
  null (x:xs) = False

  not :: Bool -> Bool
  not True  = False
  not False = True
```

```
module X where
  import Prelude
  notNull xs = not (null xs)
```



Prelude.hi
Gives type signature and unfolding for null, not

LAMBDA: Beta reduction

- Beta reduction is easy: make a 'let'

```
(\x. e1) e2
    ===>
let x = e2 in e1
```

- Note that: beta reduction produces a 'let'; now inlining decisions take over again.
- Other compilers: inline function call
  GHC: inline occurrence of function; then beta reduce

APPLY: Rewrite rules

- Lots of optimisations of the form
  fun arg_1 … arg_n ==> result
- A large number are built-in to GHC; very easy to extend this list

```
+# x 0    ==>    x
```

- Simplifier: applies the rules exhaustively
- Simple "phase system" to control order of application. Apply phase-2 rules; then phase-1 rules; then phase-0 rules.

Rewrite rules

- Users can write more RULES

```
{-# RULES
  "foldr/build" forall k z g.
      foldr k z (build g) = g k z
 #-}
```

- Works across modules
- Effectively: extends the compiler with library-specific optimisation rules!
- Compiler does not (cannot) check correctness; that's up to the programmer.

Compiler-generated rewrite rules

- Compiler writes some new rewrite rules itself

```
f :: Num a => a -> a
{-# SPECIALISE f :: Int -> Int #-}
f x = x * x
```

- To specialise 'f' at type Int:

```
$sf1 :: Int -> Int
$sf1 x = mulInt x x

{-# RULES
  "SPEC f" forall (d:Num Int).
      f @Int d = $sf1     #-}
```

Replace f at type Int by $sf1

Compiler-generated rewrite rules

- Again, two stages
  - **Definition site**: generate the specialised function, and the rewrite rule
  - **Call sites**: use the rewrite rule, call site by call site, to optimise the program
  - Works beautifully across modules

CASE: case rules

- Case of known constructor

```
case True of { False -> e1; True -> e2 }
  ===>
e2
```

- Savvy to enclosing let-binding

```
let x = a:as in …(case x of
                    []     -> e1
                    (p:ps) -> e2)…
===>
let x = a:xs in …(let p=a; ps=as in e2)…
```

CASE: case rules

- Case of known constructor

```
case True of { False -> e1; True -> e2 }
  ===>
e2
```

- …and to enclosing case

```
case x of
  True  -> …(case x of {True -> e1; False -> e2 })…
  False -> e3
===>
case x of
  True  -> …e1…
  False -> e3
```

CASE: case of case

```
notNull xs = not (null xs)
```

```
 = case (null xs) of
     True -> False
     False -> True
```

```
 = case (case xs of
           []     -> True
           (p:ps) -> False) of
     True -> False
     False -> True
```

```
module Prelude where
  null :: [a] -> Bool
  null [] = True
  null (x:xs) = False

  not :: Bool -> Bool
  not True  = False
  not False = True
```

CASE: case of case

```
notNull xs = not (null xs)
```

```
= case (case xs of
            []     -> True
            (p:ps) -> False) of
     True  -> False
     False -> True
```

```
module Prelude where
  null :: [a] -> Bool
  null [] = True
  null (x:xs) = False

  not :: Bool -> Bool
  not True  = False
  not False = True
```

```
= case xs of
    []     -> case True of
                True -> False
                False -> True

    (p:ps) -> case False of
                True  -> False
                False -> True
```

```
= case xs of
    []     -> False
    (p:ps) -> True
```

A worry

```
= case (case xs of
          []     -> True
          (p:ps) -> False) of
    True  -> BIG1
    False -> BIG2
```

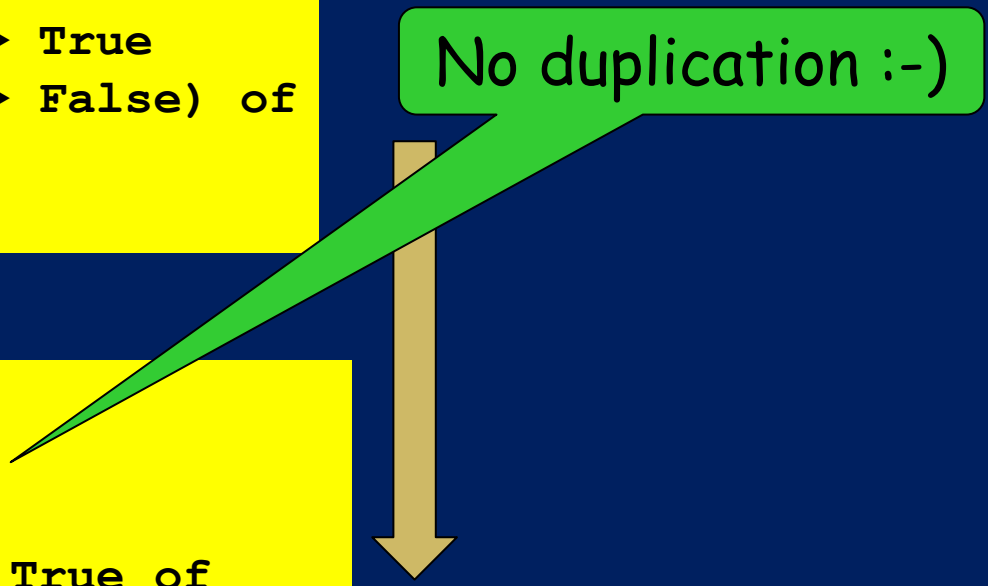
```
= case xs of
    []     -> case True of
                True -> BIG1
                False -> BIG2

    (p:ps) -> case False of
                True  -> BIG1
                False -> BIG2
```

Duplicates arbitrary amounts of code :-(

Join points (continuations)

```
= case (case xs of
          []     -> True
          (p:ps) -> False) of
     True  -> BIG1
     False -> BIG2
```




```
= let j1 () = BIG1
      j2 () = BIG2
  in case xs of
       []     -> case True of
                   True  -> j1 ()
                   False -> j2 ()

       (p:ps) -> case False of
                   True  -> j1 ()
                   False -> j2 ()
```

Join points

```
= case (case xs of
          []     -> True
          (p:ps) -> False) of
    True  -> BIG1
    False -> BIG2
```

```
= let j1 () = BIG1
      j2 () = BIG2
  in case xs of
    []     -> case True of
                True  -> j1 ()
                False -> j2 ()

    (p:ps) -> case False of
                True  -> j1 ()
                False -> j2 ()
```

```
= case xs of
    []     -> BIG1
    (p:ps) -> BIG2
```

Ordinary inlining applies

```
= let j1 () = BIG1
      j2 () = BIG2
  in case xs of
    []     -> j1 ()
    (p:ps) -> j2 ()
```

Arbitrary patterns

```
= case (case xs of
          []     -> True
          (p:ps) -> False) of
     Just x  -> BIG1
     Nothing -> BIG2
```

```
= let j1 x  = BIG1
      j2 () = BIG2
  in case xs of
    []     -> case True of
                Just x -> j1 x
                False  -> j2 ()

    (p:ps) -> case False of
                True  -> j1 x
                False -> j2 ()
```

Simply abstract over the pattern bound variables

- Works fine for existentials, GADTs

Join points are like control-flow labels

```
= let j1 x  = let v = x+1 in Just v
      j2 () = BIG2
  in case xs of
    []     -> case e1 of
                Just x -> j1 x
                False  -> j2 ()

    (p:ps) -> case e2 of
                True  -> j1 x
                False -> j2 ()
```

```
printf( "yes" ) ;
if blah then { stuff }
        else { other stuff } ;
printf( "yes" );
```

- Let bindings allocate a thunk or data constructor
- Join points allocate nothing!
- "Calling" a join point = adjust stack pointer and goto

Hot news: Sequent Core

```
case (let j x = E1
          in case xs of
            Just x  -> j x
            Nothing -> E2) of
        True  -> R1
        False -> R2
```

```
= let j x = E1
  in case xs of
    Just x  -> case (j x) of
                 True  -> R1
                 False -> R2
    Nothing -> case E2 of
                 True  -> R1
                 False -> R2
```

Bad bad bad!

1. 'j' is no longer a join point
2. The R1/R2 case does not scrutinise E1

Hot news: Sequent Core

```
case (let j x = E1
          in case xs of
               Just x  -> j x
               Nothing -> E2) of
        True  -> R1
        False -> R2
```

Move outer case into join point


```
= let j x = case E1 of
                 True  -> R1
                 False -> R2
  in case xs of
       Just x  -> j x
       Nothing -> case E2 of
                    True  -> R1
                    False -> R2
```

Sequent Core

- Based on Sequent Calculus
- But in "direct style"
- Turns out to be very modest variant of Core; easy to implement
- Some amazing new fusion happens

THINGS I HAVE NOT TOLD YOU

More in Core

- Module-level transformations
  - Strictness analysis
  - Specialisation
  - Float in, float out, lambda lifting
  - Static argument transformation
- All of Haskell's exotic type system is in Core
  - Existentials
  - GADTs
  - Newtypes, coercions, casts
  - Kind polymorphism
  - Type level functions
  - Roles

Still only nine constructors

```
data Expr
  = Var       Var
  | Lit       Literal
  | App       Expr Expr
  | Lam       Var Expr
  | Let       Bind Expr
  | Case      Expr Var Type [(AltCon, [Var], Expr)]
  | Type      Type
  | Coercion Coercion
  | Cast  Expr Coercion

data Var = Id    Name Type
         | TyVar Name Kind
         | CoVar Name Type Type

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

Conclusion

- Typed lambda calculus is totally amazing
- Little here is Haskell specific
- Cross-module inlining (SO IMPORTANT) might be problematic for Erlang
- 25 years on, still in active development:
  - Sequent Core
  - Strict Core

EXISTENTIALS, GADTS, AND COERCIONS

Existentials

```
data T a where
  T1 :: ∀a. ∀b. b -> (b -> a) -> T a

f :: T a -> a
f = Λa. \(x:T a).
    case x of
      T1 (b:*) (y:b) (g:b->a) -> g y
```

'b' is not mentioned in T1's result type

Pattern-matching on T1 binds the type variable 'b' as well as the term variables 'y' and 'g'

- We say that 'b' is an existential variable of T1

```
T1 :: ∀ab. b -> (b -> a) -> T a
    ≡ ∀a. (∃b.(b, b->a)) -> T a
```

Reminder
data types and pattern matching

- OrOr

```
data Maybe a    = Nothing | Just a
```

Or

```
data Maybe a where
    Just :: a -> Maybe a
    Nothing :: Maybe a
```

These two declarations mean the same thing

```
Just 4     :: Maybe Int
Just True     :: Maybe Bool
Nothing   :: Maybe Int
Nothing   :: Maybe Bool
Just (Just True) :: Maybe (Maybe Bool)
```

A typical evaluator

```
data Term    = Lit Int
     | Succ Term
     | IsZero Term
     | If Term Term Term

data Value = VInt Int | VBool Bool

eval :: Term -> Value
eval (Lit i)    = VInt i
eval (Succ t  = case eval t of { VInt i -> VInt (i+1) }
eval (IsZero t)  = case eval t of { VInt i -> VBool (i==0) }
eval (If b t1 t2)= case eval b of
          VBool True -> eval t1
          VBool False -> eval t2
```

Richer data types

What if you could define data types with richer return types?
Instead of this:

```
data Term where
    Lit    :: Int -> Term
    Succ   :: Term -> Term
    IsZero    :: Term -> Term
    If :: Term -> Term -> Term -> Term
```

we want this:

```
data Term a where
    Lit    :: Int -> Term Int
    Succ   :: Term Int -> Term Int
    IsZero :: Term Int -> Term Bool
    If :: Term Bool -> Term a -> Term a -> Term a
```

Richer data types

```
data Term a where
    Lit    :: Int -> Term Int
    Succ   :: Term Int -> Term Int
    IsZero :: Term Int -> Term Bool
    If     :: Term Bool -> Term a -> Term a -> Term a
```

Major payoff: rules out ill-typed Terms

```
Lit 4   :: Term Int
IsZero (Lit 4)   :: Term Bool
If (Lit 4)     ::   *** TYPE ERROR ***
IF (IsZero (Lit 4)) :: Term a -> Term a -> Term a
```

Type evaluation

Now you can write a cool typed evaluator

```
eval :: Term a -> a
eval (Lit i) = i
eval (Succ t)  = 1 + eval t
eval (IsZero i)    = eval i == 0
eval (If b e1 e2) = if eval b then eval e1
                                else eval e2
```

In here a~Int

In here a~Bool

- Evaluator is easier to read and write
- Evaluator is more efficient too

GADTs

Generalised Algebraic Data Types (GADTs):

Single idea: allow **arbitrary return type** for constructors, provided outermost type constructor is still the type being defined

```
data Term a where
    Lit    :: Int -> Term Int
    Succ   :: Term Int -> Term Int
    IsZero :: Term Int -> Term Bool
    If     :: Term Bool -> Term a -> Term a -> Term a
```

GADTs in Core?

```
data T a where
  T1 :: Bool -> T Bool
  T2 :: T a

f :: T a -> a -> Bool
f = Λa. λ(x:T a) (y:a).
    case x of
      T1 (z:Bool) -> let (v:Bool) = not y
                     in v && z
      T2 -> False
```

Problem 1
not :: Bool -> Bool
but
y::a

```
f :: T a -> a -> Bool
f = Λa. λ(x:T a) (y:a).
    let (v:Bool) = not y
    in case x of
      T1 (z:Bool) -> v && z
      T2 -> False
```

Problem 2
Floating the let seems well-scoped, but gives a bogus program

Solution to both problems: EVIDENCE

```
data T a where
  T1 :: Bool -> T Bool
  T2 :: T a

f :: T a -> a -> Bool
f = Λa. λ(x:T a) (y:a).
    case x of
      T1 (c:a~Bool) (z:Bool)
           -> let (v:Bool) = not (y ▷ c)
              in v && z
      -> False
```

Pattern matching on T1 brings into scope some EVIDENCE that (a=Bool)

We can USE the evidence to convert (y::a) to type Bool

c is an EVIDENCE VARIABLE

If e:T and c: T~σ, then (e ▷ c) : σ

```
T1 :: ∀a. (a~Bool) -> Bool -> T a
```

Evidence

```
T1 :: ∀a. (a~Bool) -> Bool -> T a
```

- Any application of T1 must supply evidence
  T1 σ e1 e2
  where e1 : (σ~Bool), e2 : Bool
- Here e1 is a value that denotes evidence that **σ=Bool**
- And any pattern match on T1 gives access to evidence
  case s of { T1 (c:σ~Bool) (y:Bool) -> ... }
  where s : T σ

System FC

$$
\begin{aligned}
e ::= {} & x \mid k \\
& \mid e_1\ e_2 \mid \lambda(x{:}\tau).e \\
& \mid e\ \tau \mid \Lambda(a{:}\kappa).e \\
& \mid \text{let bind in } e \\
& \mid \text{case } e \text{ of } \{ alt_1 \ldots alt_n \} \\
& \mid e\ \gamma \mid \lambda(c{:}\tau_1 \sim \tau_2).e \\
& \mid e \triangleright \gamma
\end{aligned}
$$

A coercion $\gamma{:}\tau_1 \sim \tau_2$ is evidence that t1 and t2 are equivalent

Coercion abstraction and application

Type-safe cast
If $e{:}\tau$ and $\gamma{:}\ \tau \sim \sigma$, then $(e \triangleright \gamma) : \sigma$

The syntax of a term (again) encodes its typing derivation

Modifications to Core

```
data Expr
  = Var       Var
  | Lit       Literal
  | App       Expr Expr
  | Lam       Var Expr
  | Let       Bind Expr
  | Case      Expr Var Type [(AltCon, [Var], Expr)]
  | Type      Type
  | Coercion Coercion  -- Used for coercion apps
  | Cast  Expr Coercion -- Type-safe cast

data Var = Id    Name Type  -- Term variable
         | TyVar Name Kind  -- Type variable
         | CoVar Name Type Type  -- Coercion var
```

Evidence terms

```
T1 :: ∀a. (a~Bool) -> Bool -> T a
```

- Consider the call:
  T1 Bool <Bool> True : T Bool
- Here <Bool> : Bool ~ Bool
  γ ::= <T> | ...
- Can I call T1 Char γ True : T Char?
- No: that would need (γ : Char ~ Bool) and there are no such terms γ

Composing evidence terms

```
data T a where
  T1 :: Bool -> T Bool
  T2 :: T a

g :: T a -> Maybe a
g = Λa. λ(x:T a).
    case x of
      T1 (c:a~Bool) (z:Bool)
           -> Just a (z ▷ sym c)
      T2 -> Nothing
```

Have evidence c:a~Bool
Need evidence
sym c : Bool~a

γ ::= <τ> | sym γ | ...

- If γ : τ ~ σ then sym γ : σ ~ τ

Composing evidence terms

```
data T a where
  T1 :: Bool -> T Bool
  T2 :: T a

g :: T a -> Maybe a
g = Λa. λ(x:T a).
    case x of
      T1 (c:a~Bool) (z:Bool)
           -> (Just Bool z) ▷ Maybe (sym c)
      T2 -> Nothing
```

Have evidence c:a~Bool
Need evidence
Maybe (sym c) : Maybe Bool ~ Maybe a

$$\gamma ::= \langle \tau \rangle \mid \mathrm{sym}\ \gamma \mid T\ \gamma_1 \ldots \gamma_n \mid \ldots$$

- If $\gamma_i : \tau_i \sim \sigma_i$
  then $T\ \gamma_1 \ldots \gamma_n : T\ \tau_1 \ldots \tau_n \sim T\ \sigma_1 \ldots \sigma_n$

Evidence terms

Coercion values

γ, δ	$::=$	x	Variables
	$\mid$	$C\ \overline{\gamma}$	Axiom application
	$\mid$	$\gamma_1\ \gamma_2$	Application
	$\mid$	$\langle \varphi \rangle$	Reflexivity
	$\mid$	$\gamma_1\ ;\ \gamma_2$	Transitivity
	$\mid$	$sym\ \gamma$	Symmetry
	$\mid$	$nth\ k\ \gamma$	Injectivity
	$\mid$	$\forall a{:}\eta\,.\,\gamma$	Polymorphic coercion
	$\mid$	$\gamma @ \varphi$	Instantiation

Cost model

- Coercions are **computationally irrelevant**
- Coercion abstractions, applications, and casts are erased at runtime

Bottom line

- Just like type abstraction/application, **evidence abstraction/application** provides a simple, elegant, consistent way to
  - express programs that use local type equalities
  - in a way that is fully robust to program transformation
  - and can be typechecked in an absolutely straightforward way
- Cost model: coercion abstractions, applications, and casts are erased at runtime

AXIOMS

newtypes

- Haskell

```
newtype Age = MkAge Int

bumpAge :: Age -> Int -> Age
bumpAge (MkAge a) n = MkAge (a+n)
```

- No danger of confusing Age with Int
- Type abstraction by limiting visibility of MkAge
- Cost model: Age and Int are represented the same way

In Core?

```
newtype Age = MkAge Int

bumpAge :: Age -> Int -> Age
bumpAge (MkAge a) n = MkAge (a+n)
```

```
axiom ageInt :: Age ~ Int

bumpAge :: Age -> Int -> Age
bumpAge = \(a:Age) (n:Int).
            (a ▷ ageInt + n) ▷ sym ageInt
```

- Newtype constructor/pattern matching turn into casts
- (New) Top-level axiom for equivalence between Age and Int
- Everything else as before

Parameterised newtypes

```
type GenericQ r = GQ (forall a. Data a => a -> r)
```

```
axiom axGQ r :: GenericQ r ~ ∀a. Data a => a -> r
```

- Axioms can be parameterised, of course
- No problem with having a polytype in s~t

Type functions

```
type family Add (a::Nat) (b::Nat) :: Nat

type instance Add Z     b = b
type instance Add (S a) b = S (Add a b)
```

```
axiom axAdd1 b   :: Add Z b ~ b
axiom axAdd2 a b :: Add (S a) b ~ S (Add a b)
```

WRAP UP

Wrap up

- Many more aspects not covered in this talk
  - Roles: nominal and representational equality
  - Optimising coercions
  - "Closed" type families with non-linear patterns, and proving consistency thereof

```
type family Eq a b where
  Eq a a = True
  Eq a b = False
```

  - Heterogeneous equalities; coercions at the type level
  - Collapsing types and kinds: * :: *

Wrap up

- Small, statically typed intermediate language
  - Sanity
  - Powerful, general optimisations
  - Lint
- Main "new" idea: programs manipulate **evidence** along with types and values. This single idea in Core explains multiple source-language concepts:
  - GADTs
  - Newtypes
  - Type and data families (both open and closed)
- Great example of theory into practice