

the implementation of

# ACCELERATE:

# PARALLEL

# FUNCTIONAL

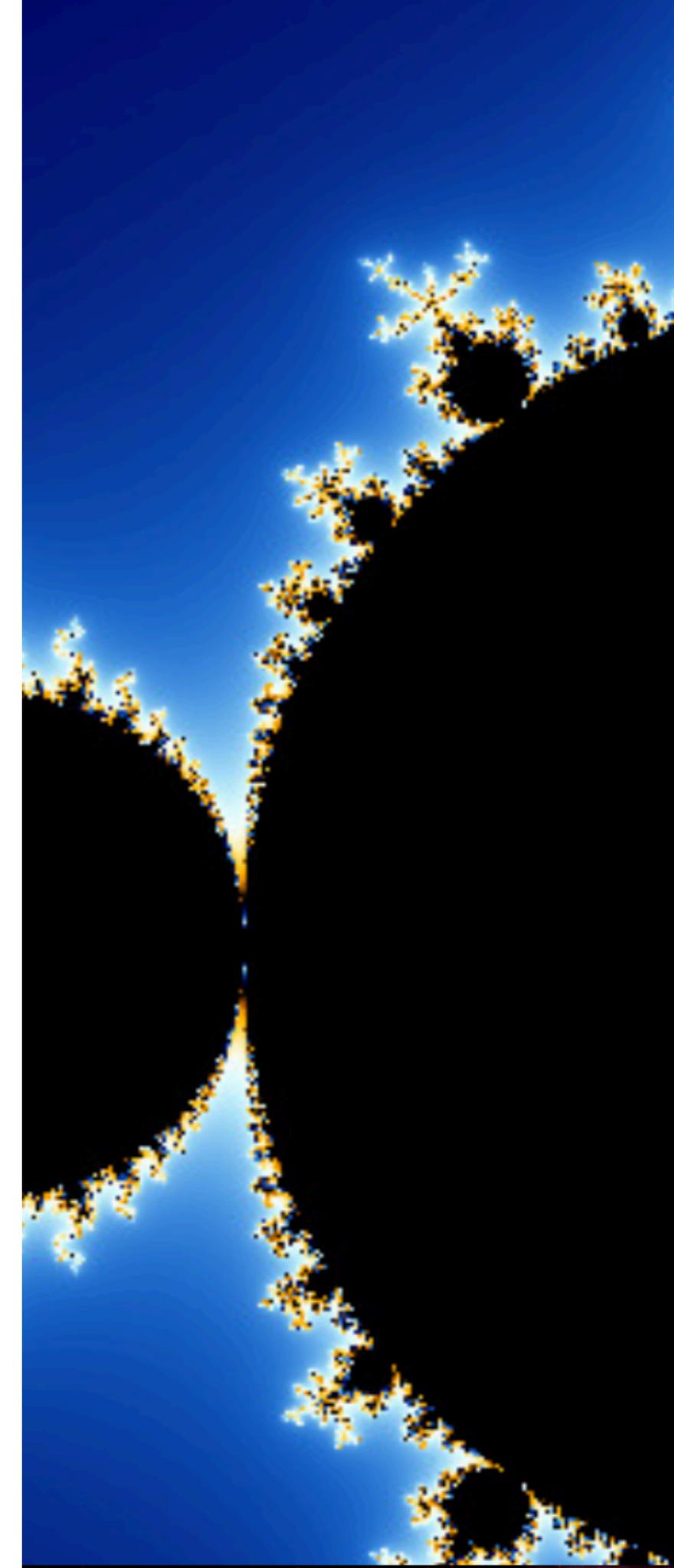
# ARRAY

# PROGRAMMING

---

*Trevor L. McDonell  
Gabriele Keller  
Manuel M T Chakravarty*

*Robert Clifton-Everest  
Josh Meredith  
Ben Lippmeier*



# PARALLEL FUNCTIONAL PROGRAMMING

Parallel Programming  
performance!



Functional Languages  
abstraction  
higher order functions  
controlled side effects  
...

# MAKING FP (AND TYPES) WORK FOR US

- Abstraction also means the compiler has more information
  - controlled side effects, no user-level pointers,...
- Collection oriented versus explicit loops/recursion
- Expressive type systems help to
  - guide the user
  - guide the compiler writer

but still, parallel versions of general purpose  
FP language are very hard to implement  
efficiently!

How about  
domain specific languages  
with  
specialised code generation?

# DOMAIN SPECIFIC LANGUAGES

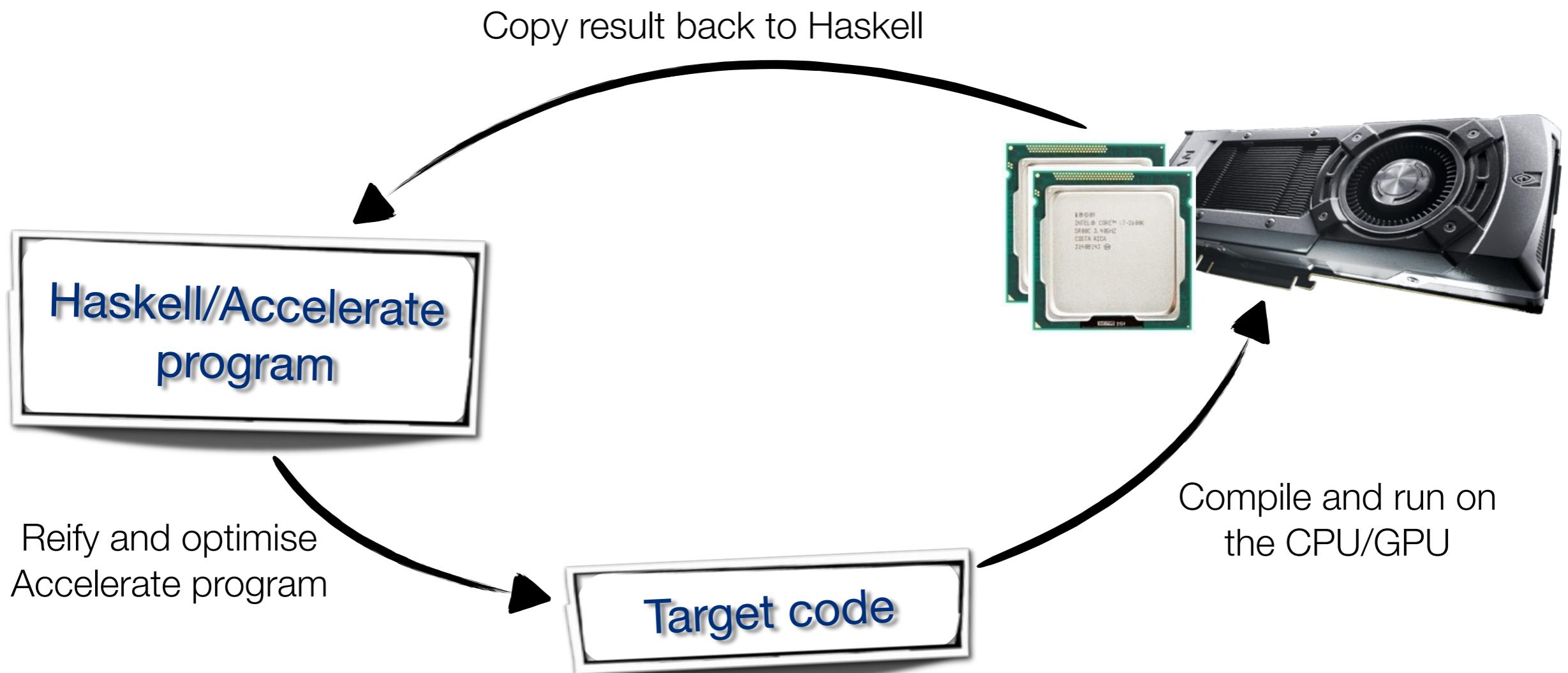
---

- Are **restricted** languages
  - Generally have specialised features to a particular application domain
  - HTML, Matlab, SQL, postscript, LaTeX ...
- **Embedded** domain specific languages
  - Implemented as libraries in the host language, so can integrate with the host language
  - Reuse the syntax of the host language (as well as parser, type checker...)
  - The host language can generate embedded code
  - Functional languages are great as host languages

# ACCELERATE

---

- An embedded domain-specific language for (high-performance) data-parallel arrays in Haskell



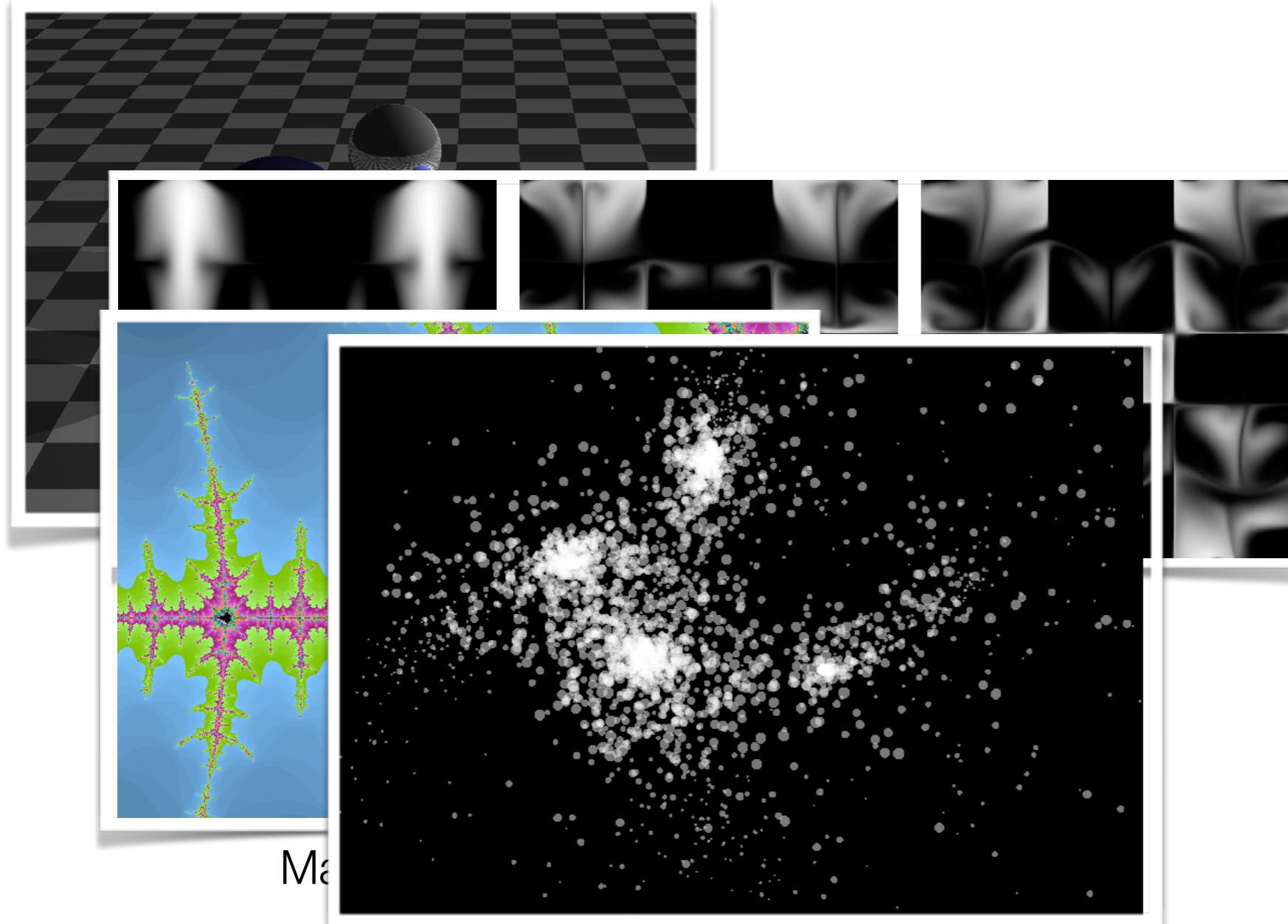
# ACCELERATE

---

- Array computations



- Everything else



n-body gravitational simulation

# DOMAIN SPECIFIC EMBEDDED LANGUAGES

---

- There are two ways to embed a language
  - shallow embedding
  - deep embedding

# SHALLOW EMBEDDING

---

- Shallow embedding provides fixed interpretation
- Semantics captured in the type
- **Example:** arithmetic expression language

```
type Expr = Float
```

# DEEP EMBEDDING

---

- Captures DSL expression as abstract syntax tree (AST), allowing multiple interpretations

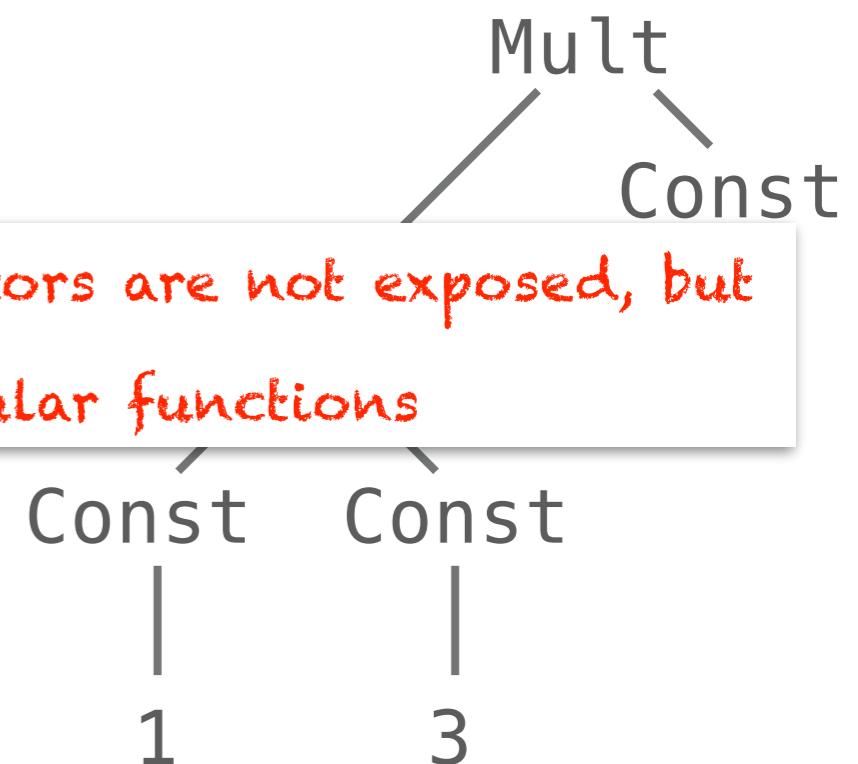
```
data Expr
= Add Expr Expr
| Mult Expr Expr
| Neg Expr
| Const Float
```

```
data Expr where
Add :: Expr -> Expr -> Expr
Mult :: Expr -> Expr -> Expr
Neg :: Expr -> Expr -> Expr
Const :: Float -> Expr
```

```
add :: Expr -> Expr -> Expr
add = Add
```

```
sampleExpr
= Mult (Add (Const 1) (Const 3)) (Const 5)
```

often, constructors are not exposed, but wrapped in regular functions



# DEEP EMBEDDING

---

- The expression representation is untyped:

```
data Expr where
  Add   :: Expr -> Expr -> Expr
  Mult  :: Expr -> Expr -> Expr
  Neg   :: Expr -> Expr
  If    :: Expr -> Expr -> Expr -> Expr
  Less  :: Expr -> Expr -> Expr
  NConst :: Float -> Expr
  BConst :: Bool -> Expr
```

```
data Result where
  FRes :: Float -> Result
  BRes :: Bool -> Result
```

```
eval :: Expr -> Result
eval (BConst b) = BRes b
eval (FConst n) = FRes n
eval (If cond e1 e2) =
  case (eval cond) of
    BConst b -> if b then eval e1 else eval e2
    FConst n -> error "..."
```

# DEEP EMBEDDING

---

## ► Generalised Algebraic Data Types (GADTs)

```
data Expr a where
  Add    :: Expr Float -> Expr Float -> Expr Float
  Mult   :: Expr Float -> Expr Float -> Expr Float
  Neg    :: Expr Float -> Expr Float -> Expr Float
  Less   :: Expr Float -> Expr Float -> Expr Bool
  Const  :: a -> Expr a
  If     :: Expr Bool -> Expr a -> Expr a -> Expr a
```

```
eval :: Expr a -> a
eval (Const c) = c
eval (If cond e1 e2) =
  if (eval cond)
    then eval e1
    else eval e2
```

# DEEP EMBEDDING

---

- Generalised Algebraic Data Types (GADTs)

```
data Expr a where
  Add    :: Expr Float -> Expr Float -> Expr Float
  Mult   :: Expr Float -> Expr Float -> Expr Float
  Neg    :: Expr Float -> Expr Float
  Less   :: Expr Float -> Expr Float -> Expr Bool
  Const  :: a -> Expr a
  If     :: Expr Bool -> Expr a -> Expr a -> Expr a
```

```
simplify :: Expr a -> Expr a
simplify (Const n) = Const n
simplify (Neg (Neg e))
  = simplify e
simplify (Add e1 e2)
  = Add (simplify e1) (simplify e2)
simplify (Mult e1 e2)
  = Mult (simplify e1) (simplify e2)
```

**LET'S LOOK AT ACCELERATE NOW!**

# ACCELERATE

---

- Computations take place on **dense, multidimensional arrays**
- Parallelism is introduced in the form of **collective operations** on arrays



- The usual suspects: maps, zipWiths, folds, generators, permutes and backpermutes, stencil operations

# FIRST EXAMPLE

---

- dot-product in Haskell (on lists):

```
import Prelude

dotp :: Num a => [a] -> [a] -> a
dotp xs ys = foldl (+) 0 (zipWith (*) xs ys)
```

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

foldl :: (b -> a -> b) -> b -> [a] -> b

# FIRST EXAMPLE

---

- dot-product in Haskell (on vectors):

```
import Data.Vector.Unboxed

dotp :: (Num a, Unbox a)
      => Vector a
      -> Vector a
      -> a
dotp xs ys = foldl (+) 0 (zipWith (*) xs ys)
```

`zipWith :: (a -> b -> c) -> Vector a -> Vector b -> Vector c`

`foldl :: (b -> a -> b) -> b -> Vector a -> b`

# FIRST EXAMPLE

---

- dot-product in Haskell (using Accelerate):

```
import Data.Array.Accelerate

dotp :: (Num a, Elt a)
      => Acc (Vector a)
      -> Acc (Vector a)
      -> Acc (Scalar a)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

`zipWith :: (Elt a, Elt b, Elt c)  
=> (Exp a -> Exp b -> Exp c)  
-> Acc (Vector a)  
-> Acc (Vector b)  
-> Acc (Vector c)`

`fold :: Elt a  
=> (Exp a -> Exp a -> Exp a)  
-> Exp a  
-> Acc (Vector a)  
-> Acc (Scalar a)`

# DIFFERENT RUN FUNCTIONS

---

- Running an accelerate program:

```
dotp :: (Num a, Elt a) =>
         Acc (Vector a) -> Acc (Vector a) -> Acc (Scalar a)

vec1, vec2 :: Acc (Vector Float)
vec1 = ...
vec2 = ...

accPrg:: Acc (Scalar Float)
accPrg = dotp vec1 vec2
```

run :: Arrays a ⇒ Acc a → a

```
putStrLn $ show $ run (dotp vec1 vec2)
```

# RUNNING AN ACCELERATE PROGRAM

---

- Plugging it all together:

```
dotp :: (Num a, Elt a) =>
    Acc (Vector a) -> Acc (Vector a) -> Acc (Scalar a)

dotp vec1 vec2 = ...

vec1, vec2 :: Vector Float
vec1 = ...
vec2 = ...

main = P.putStrLn $ P.show $
        run1 (uncurry dotp) (vec1, vec2)
```

run1 :: Arrays a ⇒ (Acc a → Acc b) → a → b

# DIFFERENT RUN FUNCTIONS

---

- Compile a function on arrays:

```
runN :: Afunction f => f → AfunctionR f
```

- Compile an array function at *Haskell* compile time:

```
runQ :: Afunction f => f → TH.ExpQ
```

# ACCELERATE EXPRESSIONS

---

- Accelerate expressions can be of two distinct types:

- Embedded sequential, **scalar** expression:

Exp a

- Embedded **array** computations:

Acc a

- What is the difference between these two?

Exp Int

Acc (Scalar Int)

# ACCELERATE EXPRESSIONS

---

- Nested parallel computations can't be expressed:

```
map :: (Elt a, Elt b)
      ⇒ (Exp a → Exp b)
      → Acc (Vector a)
      → Acc (Vector b)
```

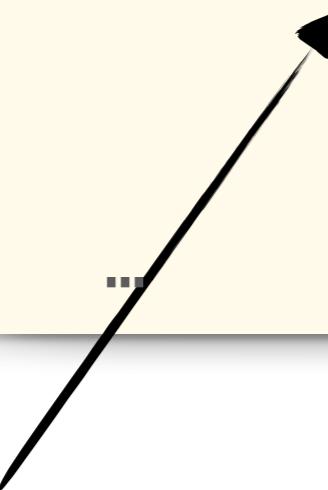
# DEFINITION OF EXP

---

- Exp is a GADT whose constructors represent scalar operations

```
data Exp a where
  Const   :: Elt c
            => c
            -> Exp c

  PrimApp :: (Elt a, Elt r)
            => PrimFun (a -> r)
            -> Exp a
            -> Exp r
```



Apply primitive scalar function: (+), (\*) ...

# AD-HOC POLYMORPHISM FOR EXP

---

- Overloaded the standard type classes to reflect arithmetic expressions
- The Num instance for Exp terms allows us to reuse standard operators like (+) and (\*)

```
instance Num (Exp Int) where
    x + y = PrimAdd numType `PrimApp` tup2 (x, y)
    ...

```

# AD-HOC POLYMORPHISM FOR EXP

---

- Use explicit dictionary passing to support ad-hoc polymorphism
  - Type checker chooses the correct instance when creating the dictionary
  - Pattern matching on the dictionary constructor makes the class constraints available

```
class (Num a, IsSingle a) => IsNum a where  
    numType :: NumType a
```

# AD-HOC POLYMORPHISM FOR EXP

---

- How does the dictionary trick work?
- With a standard algebraic data type the following are equivalent:

```
foo :: Foo a → a → a  
foo _           x = x+1
```

```
bar :: Foo a → a → a  
bar (Foo _) x = x+1
```

- But, with GADTs this is not the case

```
data Foo a where  
  Foo :: Num a ⇒ a → Foo a
```

# ACCELERATE TYPES

---

- We encountered two different Accelerate array types:

```
dotp :: (Num a, Elt a)
      => Acc (Vector a)
      -> Acc (Vector a)
      -> Acc (Scalar a)
```

- These are just two special instances of Accelerate's Array types
  - parametrised with the shape type sh
  - element type a

Array sh a

# ARRAY SHAPES

---

- The shape of an array determines its dimensionality and extent

```
data Z          = Z
data head :. tail = head :. tail

type DIM0 = Z
type DIM1 = DIM0 :. Int
type DIM2 = DIM1 :. Int

type Scalar a = Array DIM0 a
type Vector a = Array DIM1 a
```

# ARRAY SHAPES

---

- Operations are shape polymorphic:

```
map :: (Shape sh, Elt a, Elt b)
      ⇒ (Exp a → Exp b)
      → Acc (Array sh a) → Acc (Array sh b)
```

```
zipWith :: (Shape sh, Elt a, Elt b, Elt c)
          ⇒ (Exp a → Exp b → Exp c)
          → Acc (Array sh a) → Acc (Array sh b) → Acc (Array sh c)
```

```
fold :: (Shape sh, Elt a)
       ⇒ (Exp a → Exp a → Exp a) → Exp a
       → Acc (Array (sh :: Int) a)
       → Acc (Array sh a)
```

```
generate :: (Shape sh, Elt a)
           ⇒ Exp sh → (Exp sh → Exp a) → Acc (Array sh a)
```

# ARRAY SHAPES

---

- This means that our dot-product has actually a more general type:

```
dotp :: (Num a, Elt a)
      => Acc (Vector a)
      -> Acc (Vector a)
      -> Acc (Scalar a)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

```
dotp :: (Num a, Elt a, Shape sh) =>
        Acc (Array (sh :. Int) a) ->
        Acc (Array (sh :. Int) a) ->
        Acc (Array           sh       a)
```

# THE ELT CLASS

```
dotp :: (Num a, Elt a, Shape sh) =>
        Acc (Array (sh :. Int) a) ->
        Acc (Array (sh :. Int) a) ->
        Acc (Array      sh      a)
```

- Members of the Elt class contain admissible **surface types** for **array elements**:
  - ()
  - Int, Int32, Int64, Word, Word32, Word64 ...
  - Float, Double
  - Char
  - Bool
  - Array indices formed from Z and ( :.)
  - Tuples of all of these, e.g. (Bool, Int, (Float, Float))
  - To meet hardware restrictions, there are **no nested arrays** in Accelerate

# ELT CLASS

---

- GPUs are efficient processing arrays of elementary type
- not so much for aggregate types, pointers
- similarly CPU when using SIMD vector instructions
- set of types LLVM supports is fixed
- We map the user-friendly surface types to efficient representations

# ELT CLASS IS USER EXTENSIBLE

---

- Using type families (i.e., functions from type to type)

```
type family EltRepr t
type instance EltRepr Int    = Int
type instance EltRepr Float  = Float
type instance EltRepr (a,b)  =
    ProdRepr ( EltRepr a, EltRepr b )

type family ProdRepr t
type instance ProdRepr (a,b) = ((((), a), b)
type instance ProdRepr (a,b,c) = ((((), a), b), c)
```

- To extend the class, define

```
fromElt  :: a -> EltRepr a
toElt    :: EltRepr a -> a
```

# LIFTING

---

- How can we construct values of Exp type?
  - Accelerate supplies Exp versions of Haskell Prelude operations, some constant values via overloading
- lift/unlift to switch to and (sometimes) back

```
| trueExp :: Exp Bool
| trueExp = lift True

true :: Bool
true = unlift trueExp

swap :: Exp (Int, Int) -> Exp (Int, Int)
swap pairExp =
  let (x, y) = unlift pairExp :: (Exp Int, Exp Int)
  in lift (y, x)
```

# LIFTING

c is a type constructor  
(e.g., Exp, Acc)

Plain is an associated type (function  
from type e to some other type)

Strips away the surface type constructors

```
class Lift c e where
  type Plain e
  lift :: e -> c (Plain e)
```

```
class Lift c e => Unlift c e where
  unlift :: c (Plain e) -> e
```

```
instance Lift Exp Int where
  type Plain Int = Int
  lift = Exp . Const
```

```
instance Lift Exp (Exp e) where
  type Plain (Exp e) = e
  lift = id
```

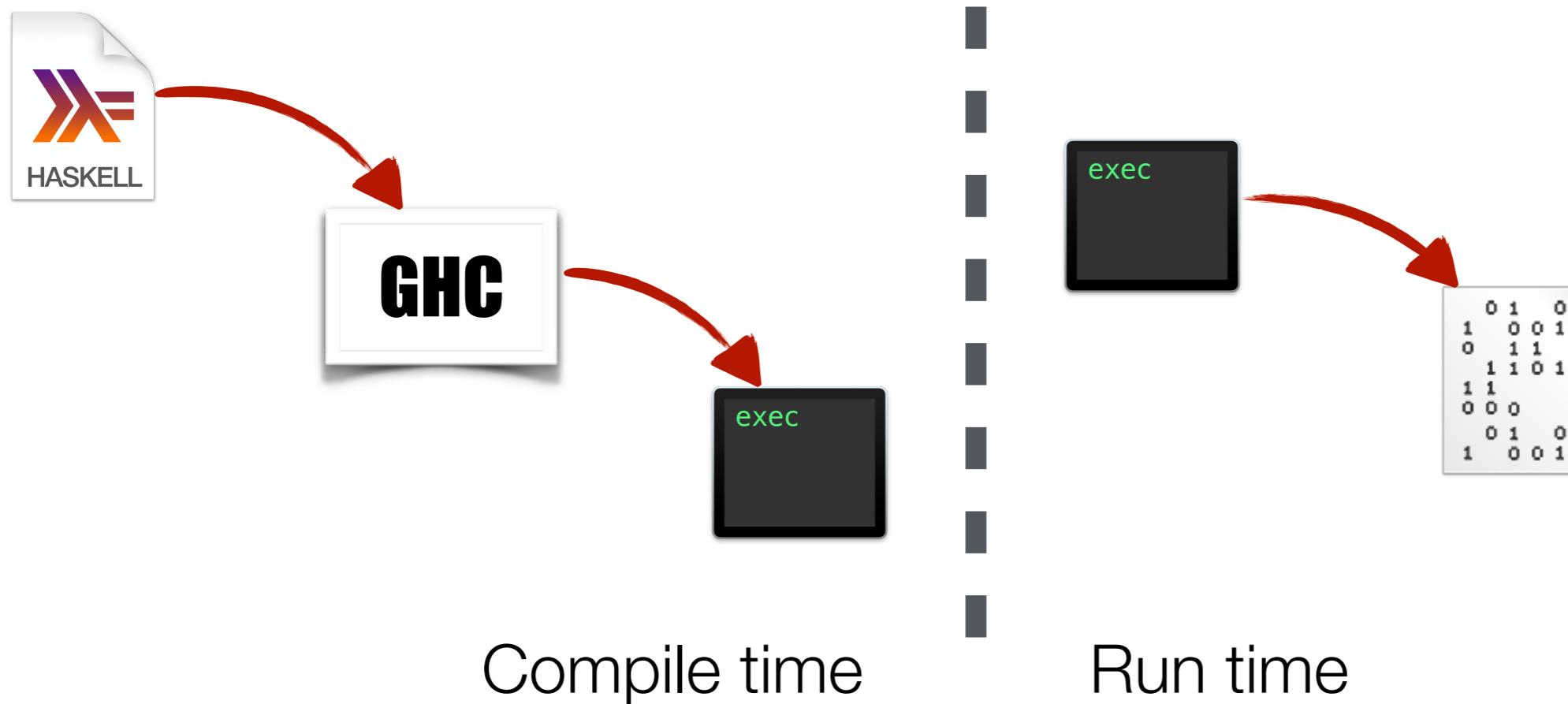
Plain (Exp Int, Int) ~ (Int, Int) ~ Plain (Int, Exp Int)

# **BACK TO THE IMPLEMENTATION OF EDSLs**

# EXECUTION OF AN ACCELERATE PROGRAM

---

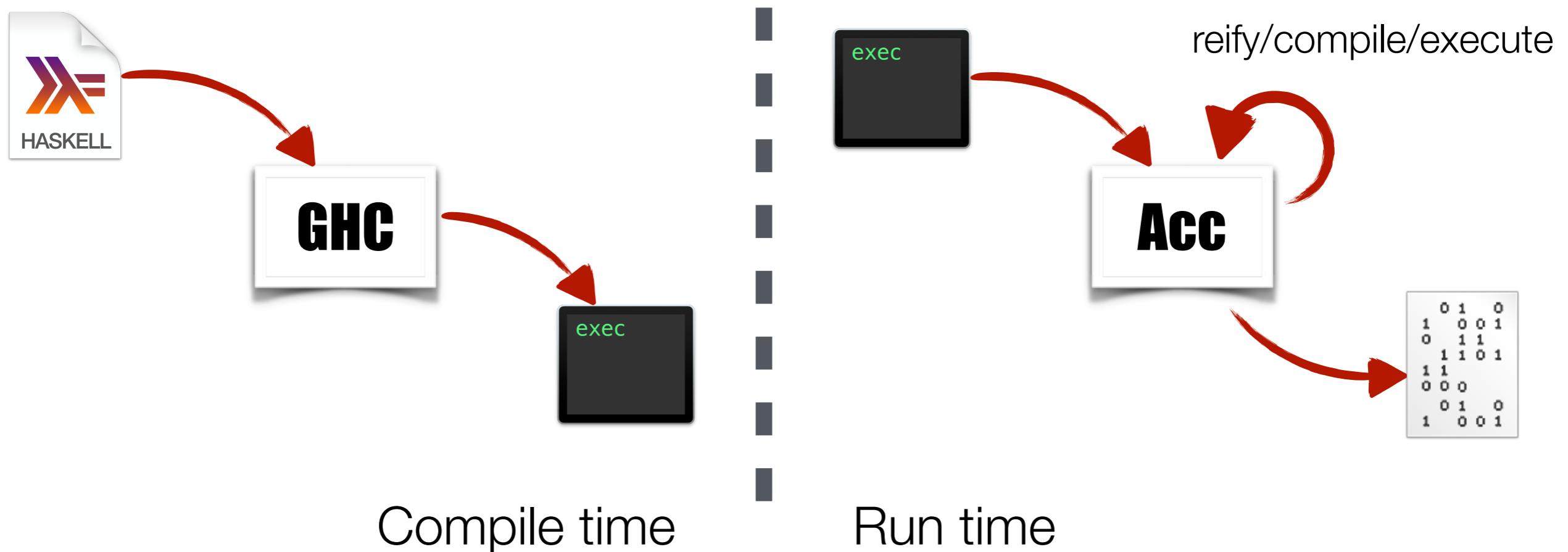
- What happens when we compile & run a regular Haskell program?



# EXECUTION OF AN ACCELERATE PROGRAM

---

- What happens when we compile & run an Accelerate program?



# WHY A TYPED AST?

---

- We compile the AST during application runtime
  - embedded **compile time** errors become application **runtime** errors
- Source of the type error can be
  - Accelerate user error
  - bugs in the Accelerate compiler
- Type checking the intermediate representation during Accelerate compilation
  - only shows this particular program is correct
  - program transformations may destroy type correctness

# WHY A TYPED AST?

---

- Applies to all runtime compiled EDSLs
- but particularly important for high-performance DSL
  - compilation is more challenging
- We learned the hard way
  - original CUDA backend was untyped
  - many bugs we found could have been avoided with a typed backend

# FIRST ORDER AND HIGHER ORDER AST

---

- Let us look at a slightly more interesting EDSL:
  - values of the source language can be lifted into the DSL like the Const constructor in the arithmetic DSL
  - function application
  - lambda-abstraction

*How do we model variables?*

# HIGHER ORDER AST

---

- We can use the variables and abstraction mechanism of the host language:

```
data HOExpr a where
  HConst :: a          -> HOExpr a
  HApp   :: HOExpr (a -> b) -> HOExpr a -> HOExpr b
  HLambda :: (HOExpr a -> HOExpr b)      -> HOExpr (a -> b)
```

```
f :: Num a => HOExp a -> HOExp a
let f = \ea -> (HApp (HApp (HConst (+)) ea) ea)
eval (HApp (HLambda f) (HConst 5))
```

```
evalHO :: HOExpr a -> a
evalHO (HConst c)  = c
evalHO (HApp e1 e2) = (evalHO e1) (evalHO e2)
evalHO (HLambda f) = \a -> (evalHO (f (HConst a)))
```

# HIGHER-ORDER AST

---

- Convenient to write and evaluate:
  - abstraction
  - application of the host language
- Not suitable for transformation & analysis of the AST
  - can't see inside functions
- **Summary:** good for surface syntax, not great for internal representation

# FIRST ORDER AST

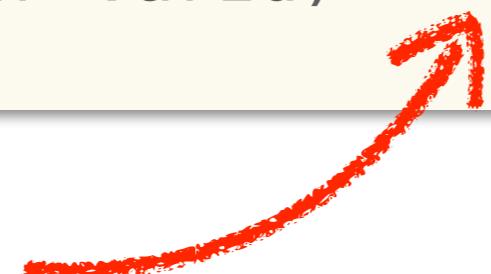
---

- Variables as regular terms of the language

```
type VarId = String

data F0Expr a where
    FConst   :: a          -> F0Expr a
    FApp     :: F0Expr (a -> b) -> F0Expr a -> F0Expr b
    FVar     :: VarId
    FLambda  :: (a -> b)
```

```
evalF0 :: F0Expr a -> a
evalF0 (FVar varId) = ???
```



we need an environment of some sort  
but what is its type??

# DE BRUIJN INDEX

---

- Alternative representation of lambda-terms eliminating names:

$$\lambda x. \lambda y. (x + y)$$

$$\lambda x. (x + (\lambda f. fx)(+1))$$

- Names are replaced with indices encoding the nesting depth of the binder

$$\begin{aligned} & \lambda \lambda (i_1 + i_0) \\ & \lambda (i_0 + (\lambda i_0 i_1)(+1)) \end{aligned}$$

The diagram illustrates the assignment of De Bruijn indices to variables in the terms. In the first term,  $i_1$  is the index for the inner  $\lambda y.$  and  $i_0$  is the index for the outer  $\lambda x.$  In the second term,  $i_0$  is the index for the inner  $\lambda f.$  and  $i_1$  is the index for the outer  $\lambda x.$  Red arrows indicate the binding of  $i_0$  and  $i_1$  to their respective  $\lambda$  abstractions, while green arrows indicate the binding of the  $\lambda$  abstractions to the  $(+1)$  constant.

# FIRST ORDER SYNTAX WITH DE BRUIJN

---

## ➤ Idea:

- a environment is either empty, or a tuple of value and rest environment
- the type of the environment describes the type of all the values it contains
- $i$ th entry is the value of variable bound at nesting level  $n$

```
data Val env where
  Empty :: Val ()
  Push  :: Val env -> t -> Val (env, t)
```

```
Push (Push Empty 5) True :: Val ((((), Int), Bool))
```

# FIRST ORDER SYNTAX WITH DE BRUIJN

---

- Idea:

- a variable is a typed index
- the type encodes the type of the value the variable it represents, as well as the type of the environment is needs

```
data Idx env t where
  ZeroIdx :: Idx (env, t) t
  SuccIdx :: Idx env t -> Idx (env, s) t
```

```
prj :: Idx env t -> Val env -> t
prj ZeroIdx      (Push val v) = v
prj (SuccIdx idx) (Push val _) = prj idx val
```

# FIRST ORDER SYNTAX WITH DE BRUIJN

---

- A term type in our language is parametrised with two types:
  - the result type  $t$
  - the environment type  $\text{env}$

```
data Term env t where
  Var :: Idx env t
  Con :: t
  Lam :: Term (env, s) t
  App :: Term env (s -> t) -> Term env s -> Term env t
```

# FIRST ORDER SYNTAX WITH DE BRUIJN

---

- The type-safe evaluator is now written as:

```
eval :: Term env t → Val env → t
eval (Var ix)      val = prj ix val
eval (Con v)        val = v
eval (Lam body)    val = (eval body) . (val `Push`)
eval (App fun arg) val = (eval fun val) (eval arg val)
```

# FIRST ORDER SYNTAX WITH DE BRUIJN

---

- Typed higher-order abstract syntax is
  - convenient as surface syntax
  - not suitable for program analysis, program transformations
- Typed de Bruijn first order abstract syntax
  - impractical to use as surface syntax
  - well suited as internal representation
- Solution:
  - user writes program in HO-syntax
  - we convert it to de Bruijn representation



**PROBLEM SOLVED NOW, RIGHT?**

**UHM, NO....**

# SHARING

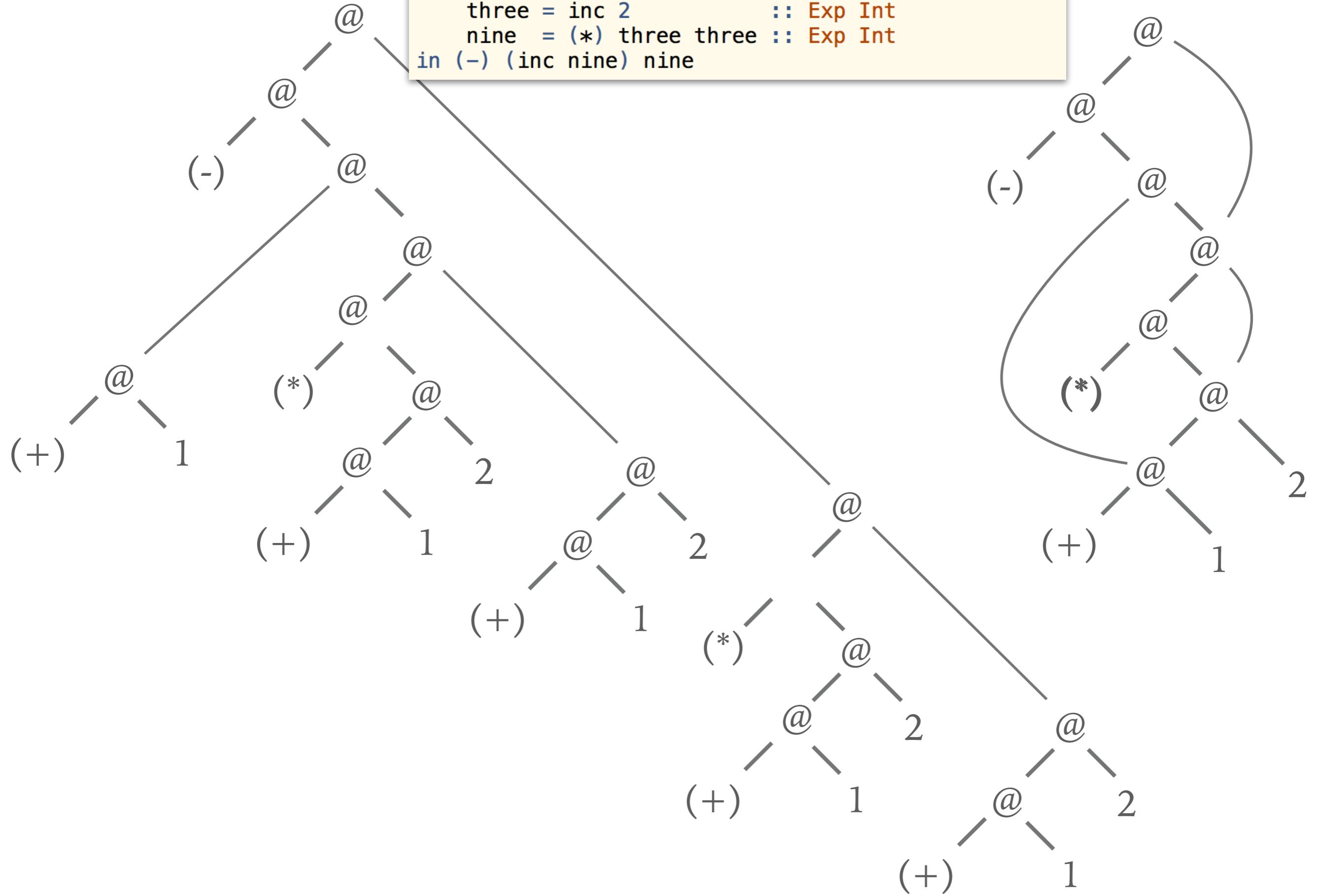
---

- What does the AST look like for this expression?

```
let
    inc   = (+) 1           :: Exp Int -> Exp Int
    three = inc 2          :: Exp Int
    nine  = (*) three three :: Exp Int
in (-) (inc nine) nine
```

let

```
inc  = (+) 1           :: Exp Int -> Exp Int
three = inc 2          :: Exp Int
nine  = (*) three three :: Exp Int
in (-) (inc nine) nine
```



# PROBLEM

---

- Sharing in the host language internal representation
- Not readily observable
- Processing the tree means we are loosing the sharing information
  - often results in large ASTs
  - expressions are evaluated multiple times
  - really, really inefficient

# PROBLEM

---

- Black-Scholes option pricing
  - Without sub-expression sharing, 20 times slower

```
blackscholes :: Vector (Float, Float, Float)
              -> Acc (Vector (Float, Float))
blackscholes = map callput . use
  where
callput x =
  let (price, strike, years) = unlift x
    r      = constant riskfree
    v      = constant volatility
    v_sqrtT = v * sqrt years
    d1     = (log (price / strike) +
              (r + 0.5 * v * v) * years) / v_sqrtT
    d2     = d1 - v_sqrtT
    cnd d = let c = cnd' d in d >* 0 ? (1.0 - c, c)
    cndD1 = cnd d1
    cndD2 = cnd d2
    x_expRT = strike * exp (-r * years)
  in
lift ( price * cndD1 - x_expRT * cndD2
      , x_expRT * (1.0 - cndD2) - price * (1.0 - cndD1))
```

```
riskfree, volatility :: Float
riskfree   = 0.02
volatility = 0.30

horner :: Num a => [a] -> a -> a
horner coeff x = x * foldr1 madd coeff
  where
    madd a b = a + x*b

cnd' :: Floating a => a -> a
cnd' d =
  let poly      = horner coeff
      coeff     = [0.31938153, -0.356563782,
                  1.781477937, -1.821255978,
                  1.330274429]
      rsqrt2pi = 0.39894228040143267793994605993438
      k        = 1.0 / (1.0 + 0.2316419 * abs d)
  in
    rsqrt2pi * exp (-0.5*d*d) * poly k
```

# PROBLEM

---

- Including ‘let’ in the surface language would make it extremely awkward to use
- Can we have ‘let’ in the internal representation, and convert without loosing sharing?

# PROBLEM

---

- We need to be able to observe an implementation detail pure functional languages abstract over
  - referential equality:
    - not enough to know two values are the same, we need to check if they share a location
    - language level reference equality clashes with referential transparency, garbage collection, compiler optimisations
- Luckily, the need for referential equality pops up in other contexts as well
  - memoization, O(1) comparison of large objects,...

# STABLE NAMES

---

➤ Idea:

- associate values with an address-like **stable** name

```
data StableName a  
  
mkStableName :: a -> IO (StableName a)  
hashStableName :: StableName a -> Int  
  
instance Eq (StableName a)  
instance Ord (StableName a)
```

# STABLE NAMES

---

$\text{mkStableName } x = \text{mkStableName } y$

$\Rightarrow$

$x = y$

$x \neq y$

$\Rightarrow$

$\text{mkStableName } x \neq \text{mkStableName } y$

$x = y$

$\not\Rightarrow$

$\text{mkStableName } x = \text{mkStableName } y$

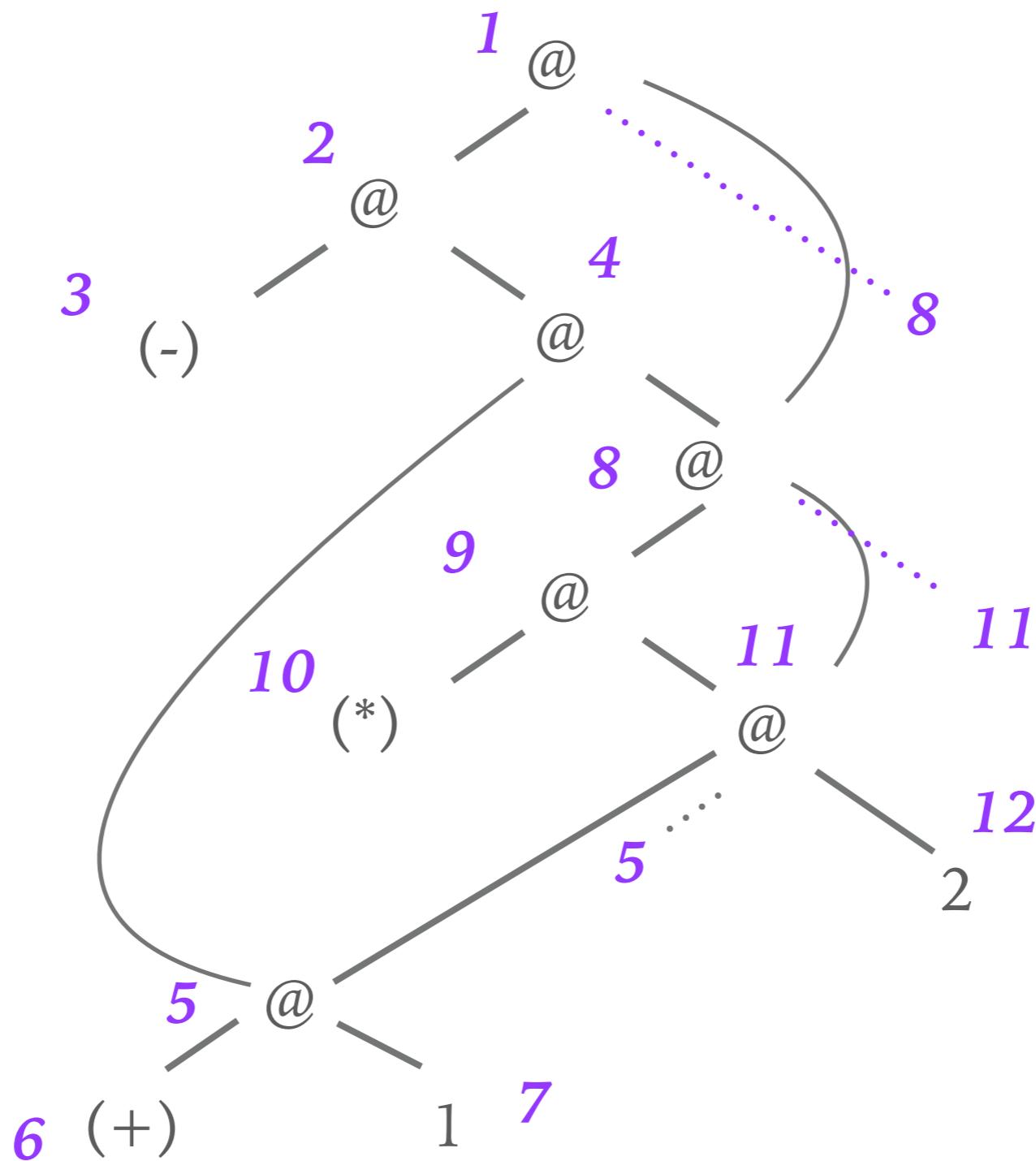
# SHARING RECOVERY

---

- Traverse the ADT structure and identify the shared nodes with the help of stable names
- insert let-bindings in the de Bruijn internal representation at the right positions

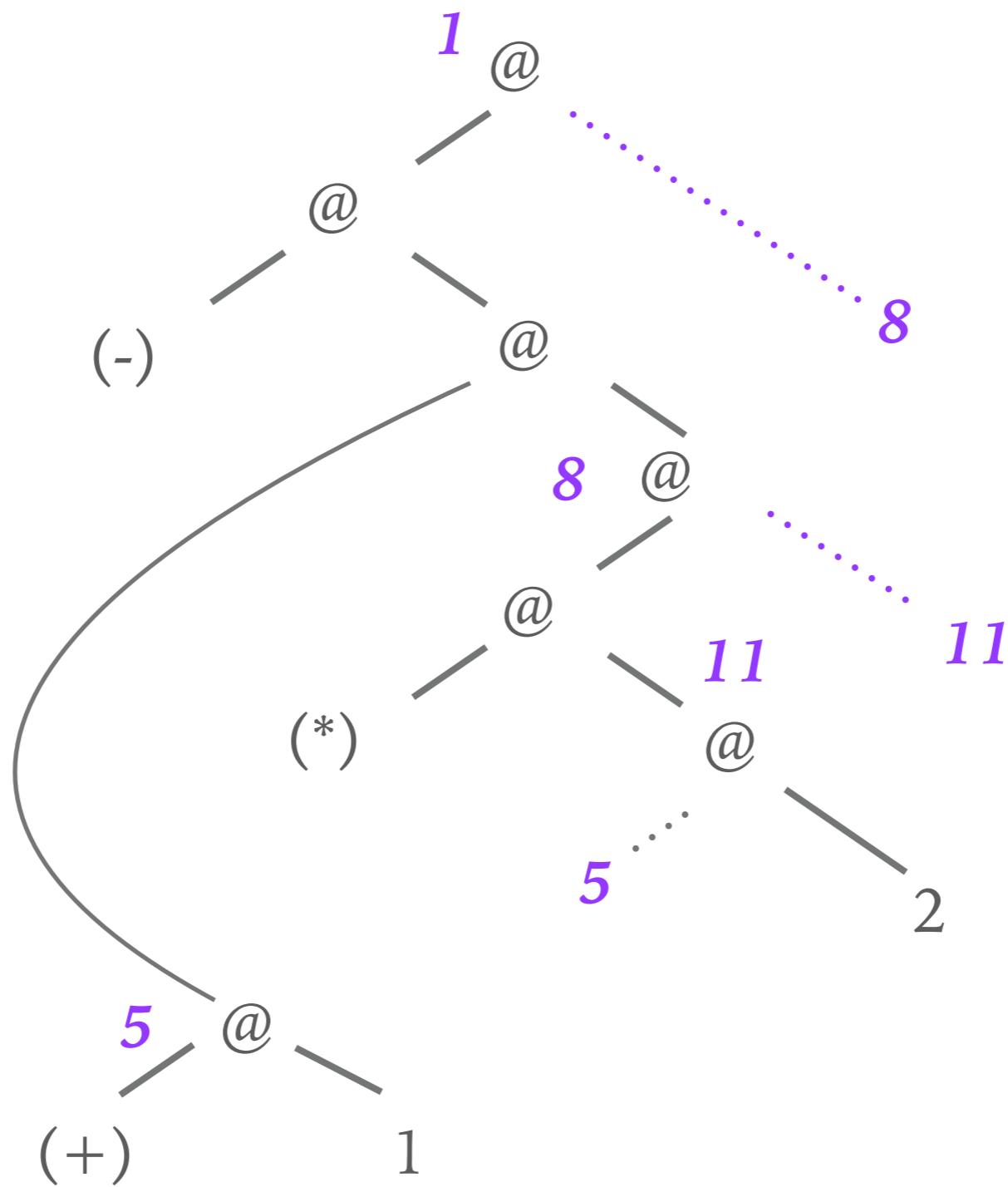
# SHARING RECOVERY

---



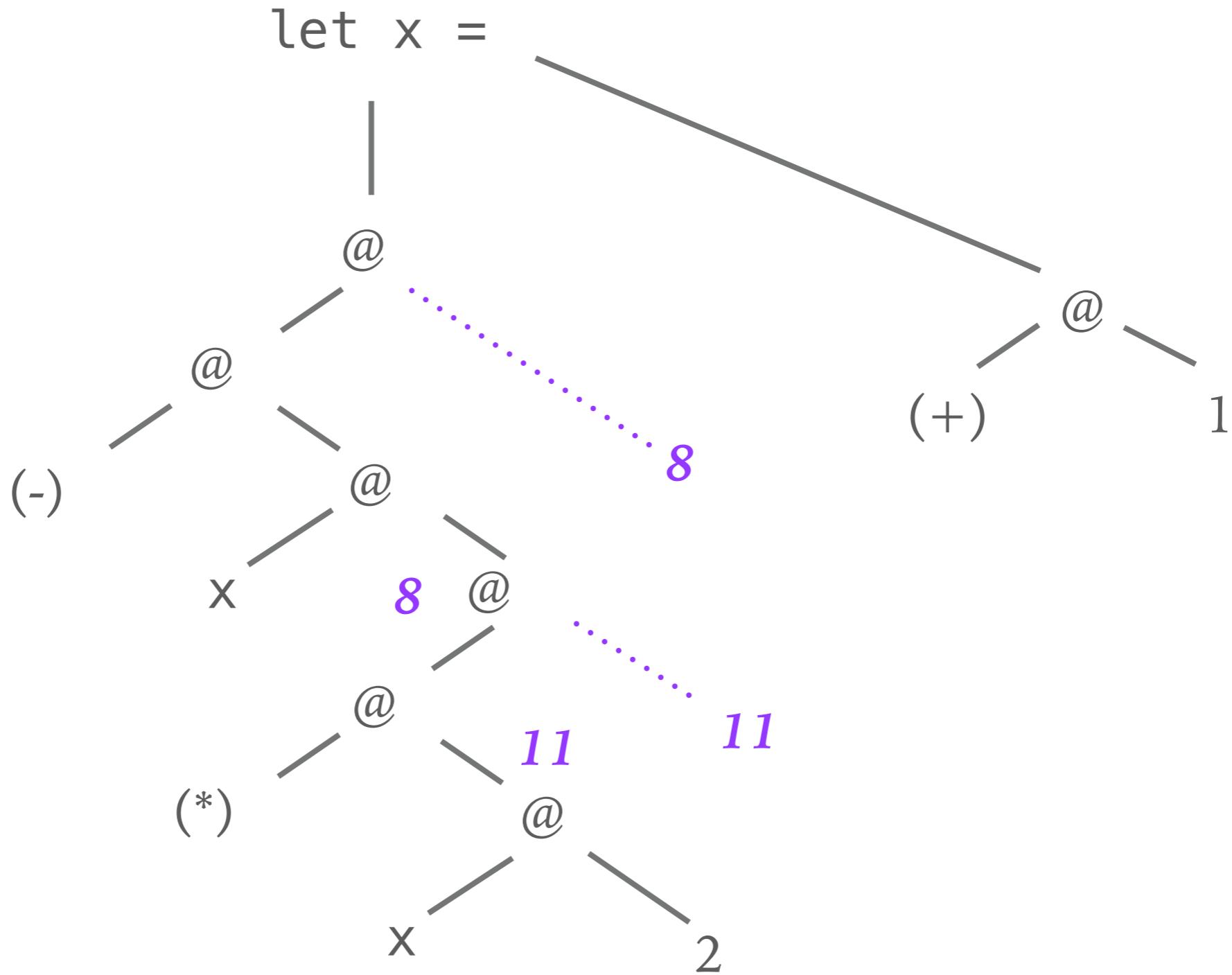
# SHARING RECOVERY

---

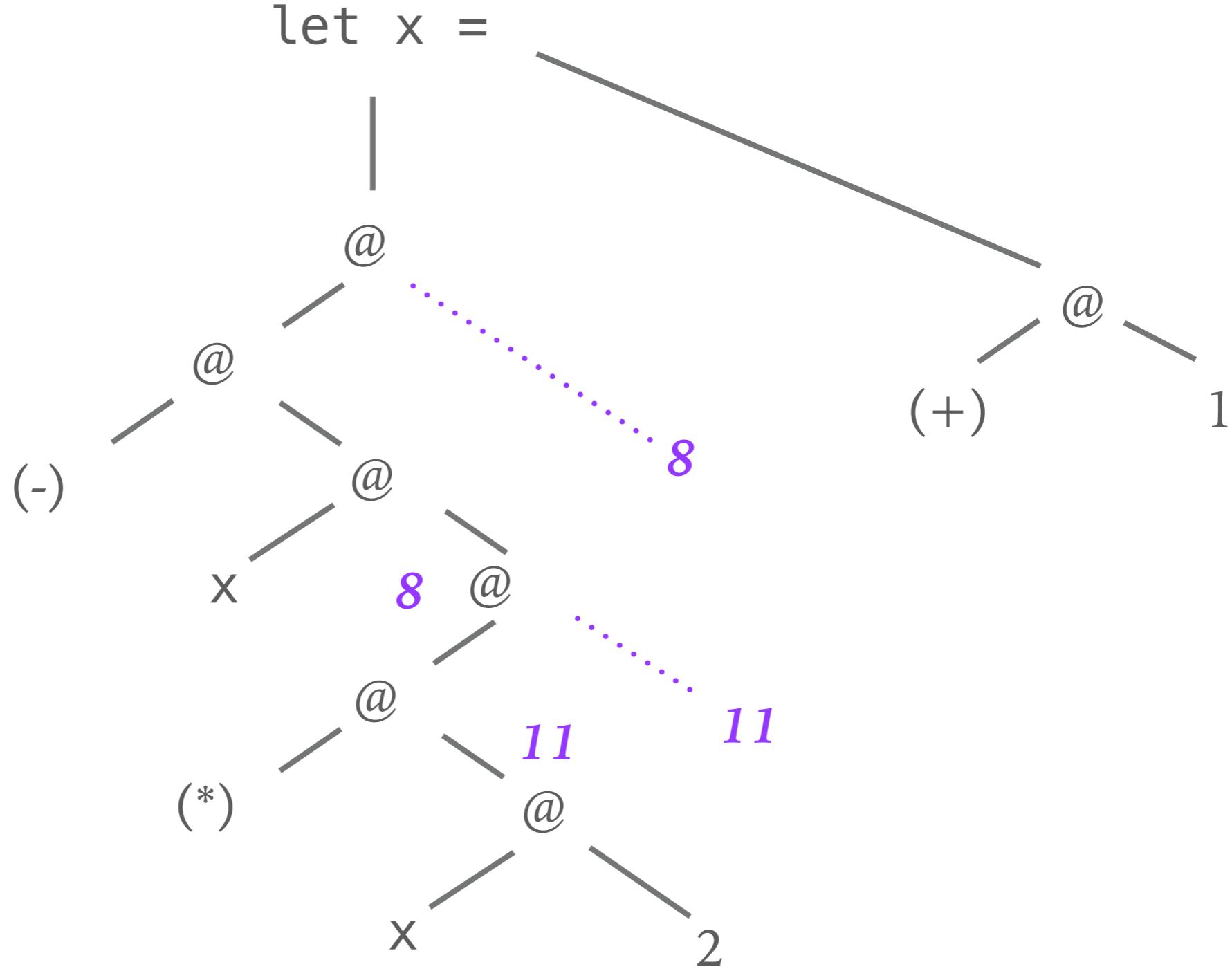


# SHARING RECOVERY

---

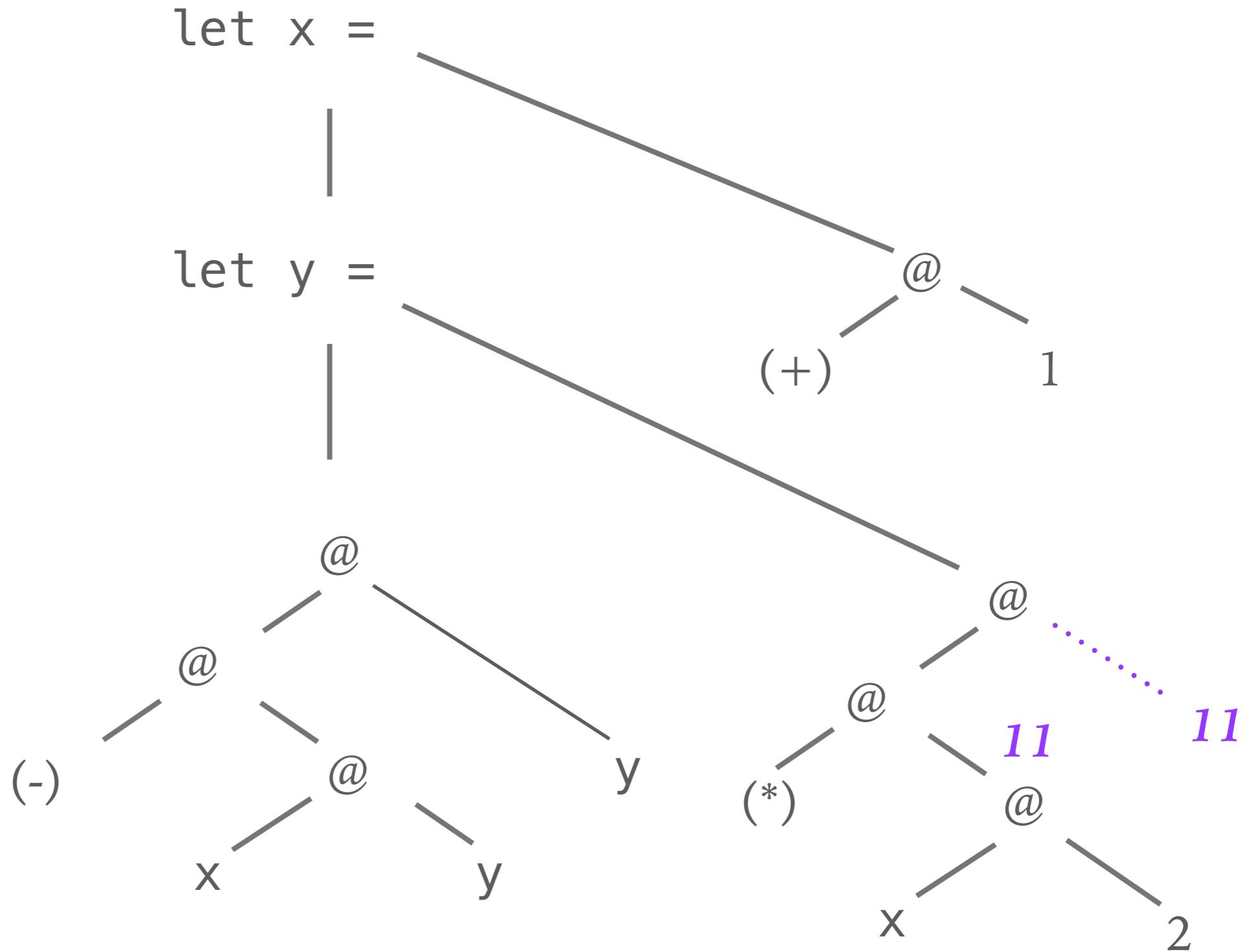


# SHARING RECOVERY



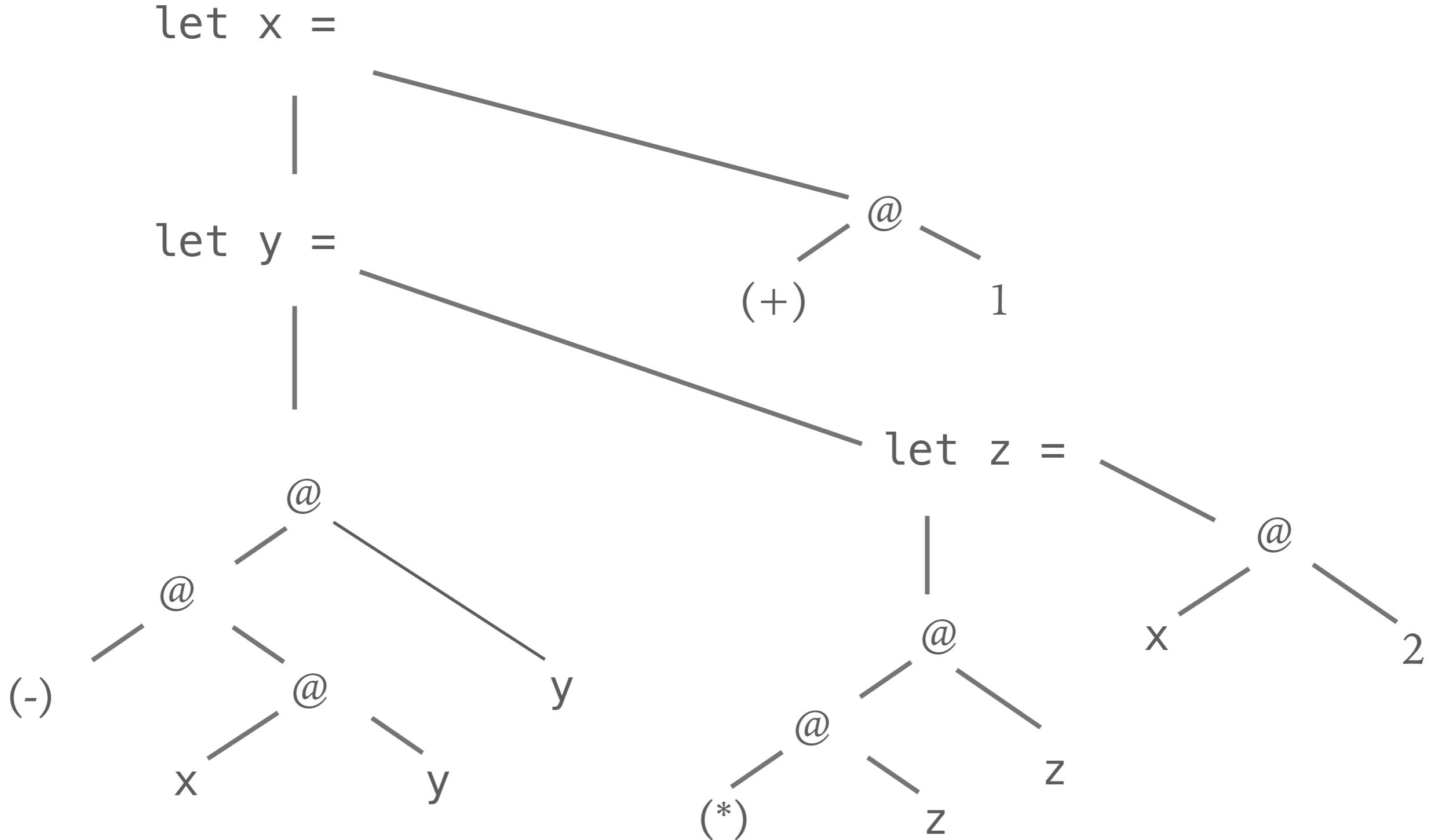
# SHARING RECOVERY

---



# SHARING RECOVERY

---



# COMBINING SHARING RECOVERY AND DE BRUIJN CONVERSION

---

- The two transformation are fused into a single one
  - applying de Bruijn conversion first would destroy sharing
- We loose type information during conversion from HOAS to de Bruijn
  - type checked dynamically

# PERFORMANCE

---

## ► Impact of sharing recovery\*:

- Black Scholes, 20M elements:
  - CUDA, handwritten: 6.7ms
  - Accelerate, w/o sharing: 116ms
  - Accelerate w. sharing: 6.12ms
- Canny edge detection, 16M pixel:
  - OpenCV: 50.6ms
  - Accelerate, w/o sharing: 82.7ms
  - Accelerate w. sharing: 78.4ms
- Fluid-flow simulation, 2M particles
  - Accelerate, w/o sharing: 107ms
  - Accelerate w. sharing: 119ms

\*Tesla T10 processor (compute capability 1.3, 30 multiprocessors = 240 cores at 1.3GHz, 4GB RAM) backed by two quad-core Xenon E5405 CPUs (64-bit, 2GHz, 8GB RAM), running GNU/Linux (Ubuntu 12.04 LTS). The reported GPU runtimes are averages of 100 runs.

# FUSION

---

- Well-known problem of collection-oriented programming:

```
map f $ map g xs  
fold (+) $ enumFromThenTo 0 1 1000000000  
  
let  
  xs = map sqrt $ enumFromThenTo 0 1 1000000000  
in fold (+) $ zipWith (*) $ map (+ (fold (+) xs)) xs
```

- Unnecessary intermediate structures, traversals

# FUSION

---

- Like sharing recovery, **fusion is essential** if we care about performance
  - Reduce the number of intermediate data structures
  - Improve memory locality

# FUSION

---

- Many of the classical techniques don't work in this context:
  - e.g., build/fold like fusion approaches destroy the parallel pattern

```
build ::  
  (forall b. (a -> b -> b) -> b -> b) -> [a]  
build g = g (:) []
```

```
foldr :: (a -> b -> b) -> b -> [a] -> b  
foldr f z []      =  z  
foldr f z (x:xs) =  f x (foldr f z xs)
```

```
foldr k z (build g) = g k z
```

```
map f xs = build (\c n -> foldr  
                  (\a b -> c (f a) b) n xs)
```

...

# FUSION

---

- Fusion often relies on inlining for producer/consumer pairs to be detected, only done conservatively
- Accelerate fusion happens at run time, need to be aware of the costs (but also: more information available)
- Result of fusion transformation needs to fit in to our code generation templates

# FUSION

---

➤ Producers:

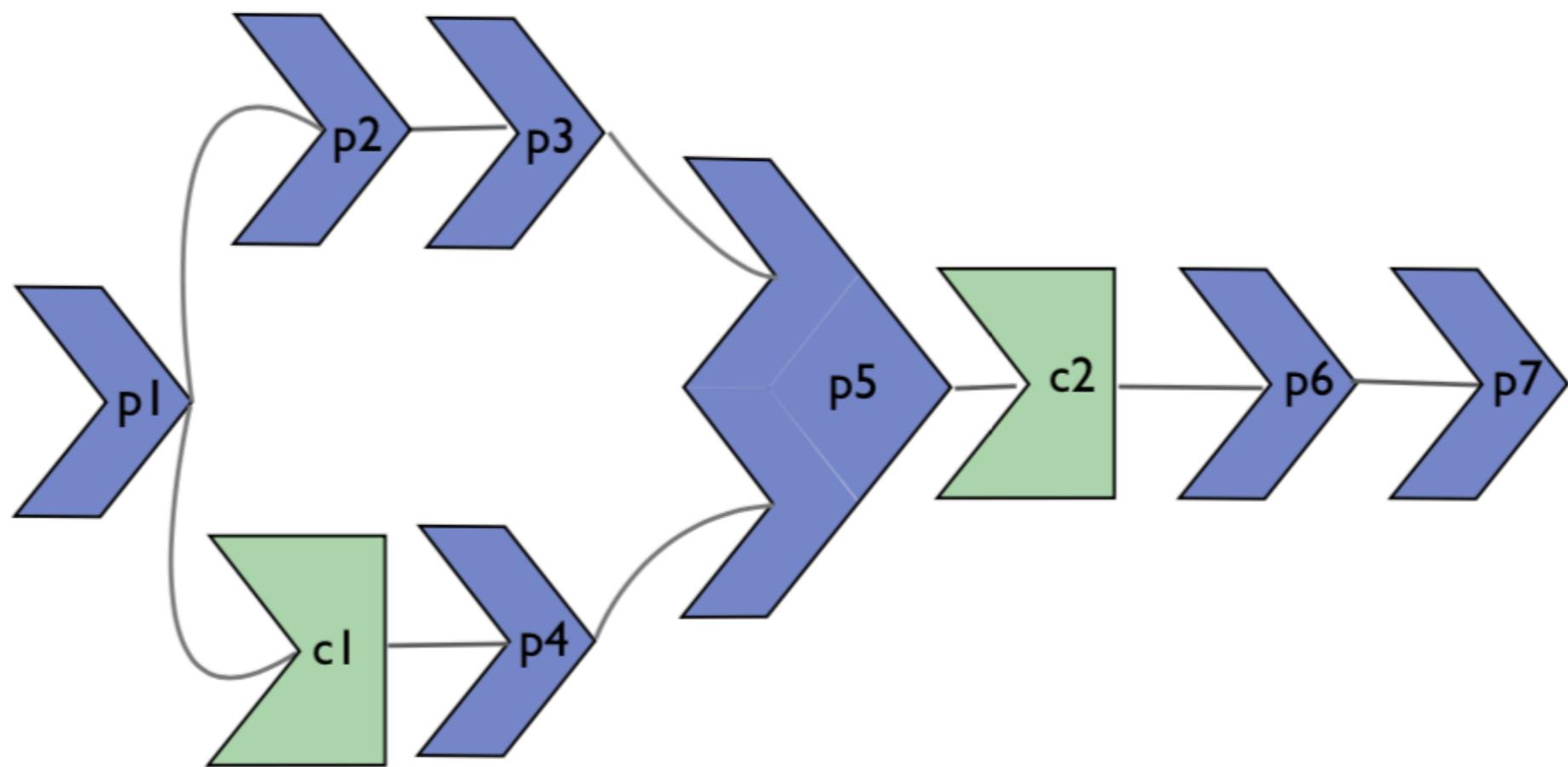
- each element of the result depends on at most one element of input array (e.g, map, backpermute, generate)

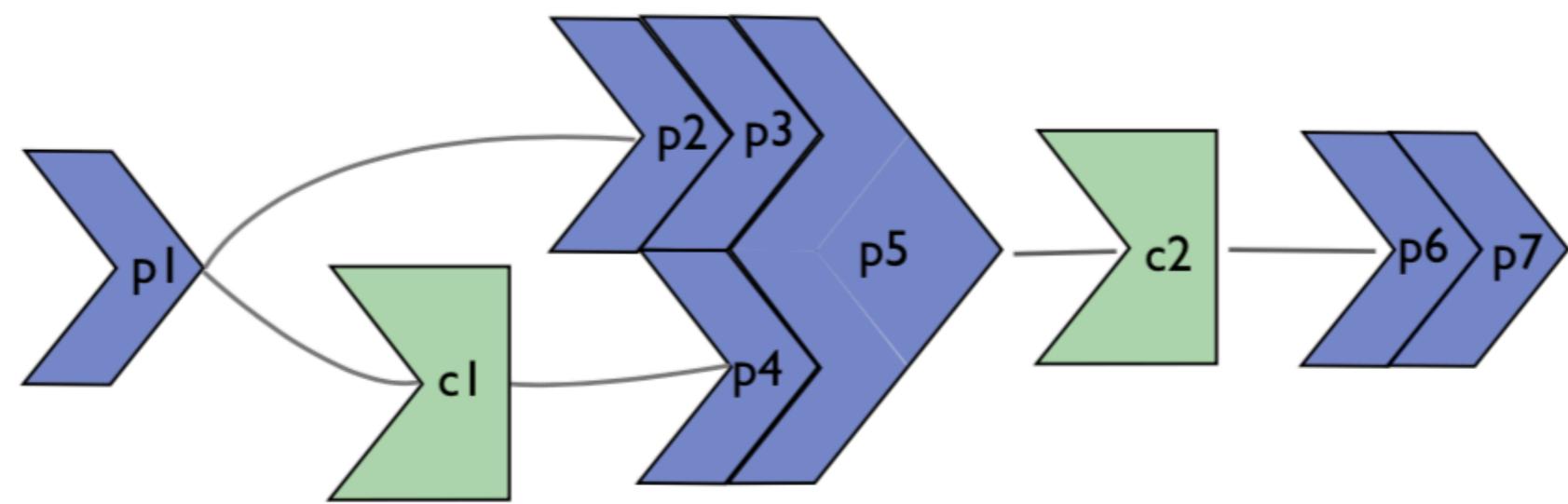
➤ Consumers:

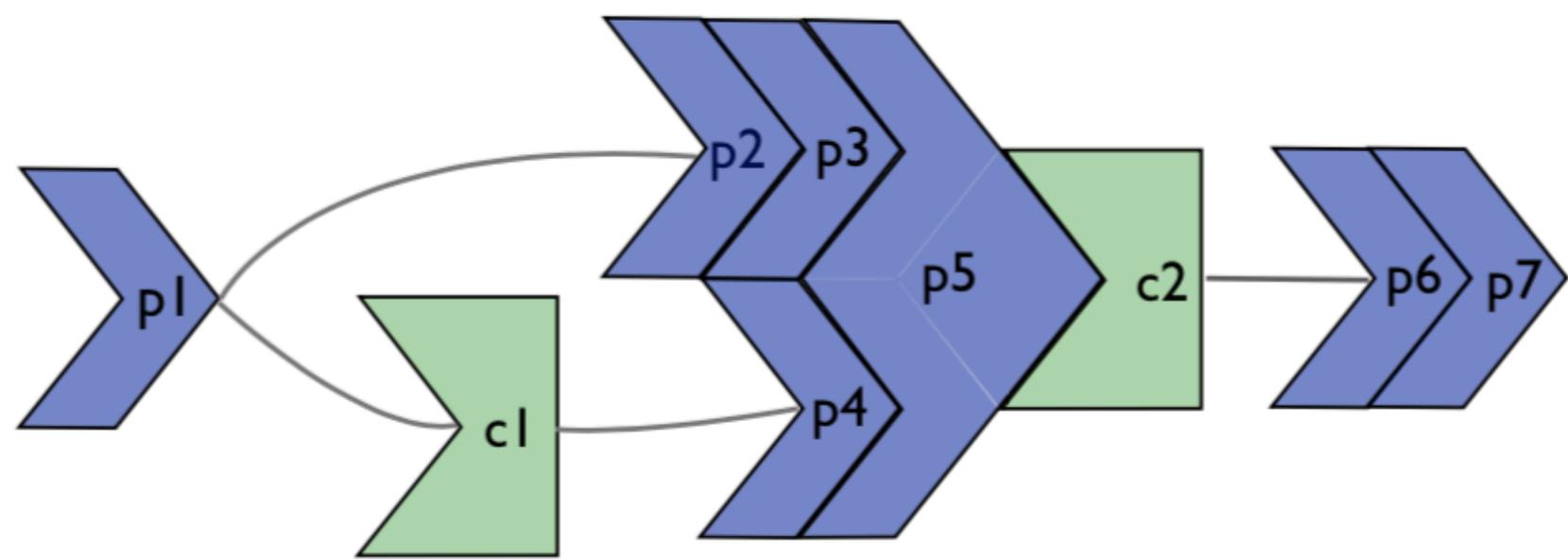
- each element of result depends on multiple elements of input array (e.g., folds, scans, stencil operations)

➤ We treat them separately

- Producer/Producer fused via program transformation
- Producer/Consumer during code generation







# PRODUCER/PRODUCER FUSION

---

- Arrays represented as delayed computations:

```
data DelayedAcc a where
  Done   :: Acc a
    -> DelayedAcc a
  Yield :: (Shape sh, Elt e)
    => Exp sh
    -> Fun (sh -> e)
    -> DelayedAcc (Array sh e)
  Step  :: (Shape sh, Shape sh', Elt e, Elt e')
    => Exp sh'
    -> Fun (sh' -> sh)
    -> Fun (e -> e')
    -> Idx (Array sh e)
    -> DelayedAcc (Array sh' e')
```

# FUSION

---

- Example: map

```
| mapD :: (Shape sh, Elt a, Elt b)
|   => Fun (a -> b)
|   -> DelayedAcc (Array sh a)
|   -> DelayedAcc (Array sh b)
mapDf(Step shpgv)
= Step shp (f.g) v
mapD f (Yield sh g)
= Yield sh (f . g)
```

- To prevent fusion, arrays can be made manifest

```
compute :: Arrays a => Acc a -> Acc a
```

# PRODUCER/CONSUMER FUSION

---

- Producer/consumer fusion is done during code generation
- Producer operations are inserted in the consumer code templates
- No support for consumer/producer fusion yet

# TYPE SAFE CODE GENERATION

---

- LLVM IR represents types as value-level data structure
- We track them as Haskell types in the LLVM binding
- Guarantees we only generate type correct LLVM programs
- GADT to define LLVM instruction set:

```
data Instruction a where
    Add :: NumType a
        -> Operand a
        -> Operand a
        -> Instruction a
    ...|
```

- We translate the well-typed Accelerate AST into a well-typed LLVM AST

# LLVM BACKEND FRAMEWORK

---

- LLVM is a reusable framework, portable across diverse architectures
- Accelerate LLVM backend framework
  - a set of re-usable components
  - reduces the cost of implementing future backends
- Existing backends:
  - vectorising multicore CPU
  - (NVIDIA) GPU backend

Benchmark	Input Size	Contender (ms)	Accelerate full (ms)	Accelerate no fusion (ms)	Accelerate no sharing (ms)
Black Scholes	20M	6.70 (CUDA)	6.19 (92%)	(not needed)	116 (1731%)
Canny	16M	50.6 (OpenCV)	78.4 (155%)	(not needed)	82.7 (164%)
Dot Product	20M	1.88 (CUBLAS)	2.35 (125%)	3.90 (207%)	(not needed)
Fluid Flow	2M	5461 (Repa -N7)	107 (1.96%)	(not needed)	119 (2.18%)
Mandelbrot (limit)	2M	14.0 (CUDA)	24.0 (171%)	245 (1750%)	245 (1750%)
N-Body	32k	54.4 (CUDA)	607 (1116%)	(out of memory)	(out of memory)
Radix sort	4M	780 (Nikola)	442 (56%)	657 (84%)	657 (84%)
SMVM (protein)	4M	0.641 (CUSP)	0.637 (99%)	32.8 (5115%)	(not needed)

Name	Non-zeros (nnz/row)	CUSP	Accelerate	Accelerate no fusion
Dense	4M (2K)	14.48	14.62	3.41
Protein	4.3M (119)	13.55	13.65	0.26
FEM/Spheres	6M (72)	12.63	9.03	4.70
FEM/Cantilever	4M (65)	11.98	7.96	4.41
Wind Tunnel	11.6M (53)	11.98	7.33	4.62
FEM/Harbour	2.37M (50)	9.42	6.14	0.13
QCD	1.9M (39)	7.79	4.66	0.13
FEM/Ship	3.98 (28)	12.28	6.60	4.47
Economics	1.27M (6)	4.59	0.90	1.06
Epidemiology	1.27M (4)	6.42	0.59	0.91
FEM/Accelerator	2.62M (22)	5.41	3.08	2.92
Circuit	959k (6)	3.56	0.82	1.08
Webbase	3.1M (3)	2.11	0.47	0.74
LP	11.3M (2825)	5.22	5.04	2.41

GFLOPS/s (higher is better)

# LULESH

*Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics*

DB: lulesh\_c\*.silo database

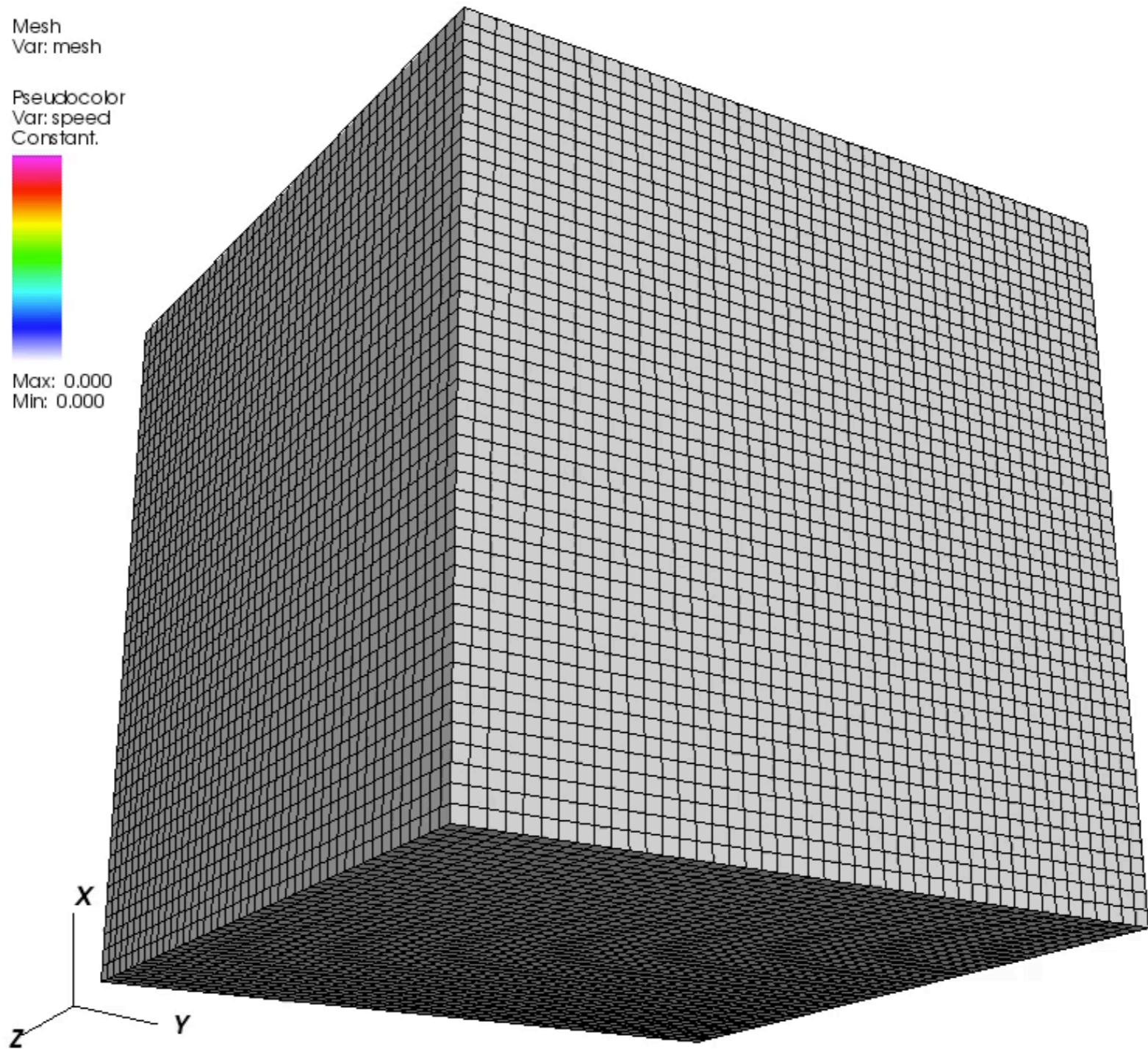
Cycle: 0 Time:0

Mesh  
Var: mesh

Pseudocolor  
Var: speed  
Constant:



Max: 0.000  
Min: 0.000



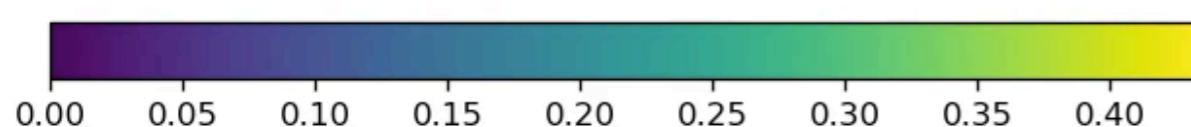
# LULESH

*Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics*

	<b>Lines of Code</b>	<b>Runtime @ <math>64^3</math> (s)</b>
<b>C (OpenMP)</b>	2400	64
<b>CUDA</b>	3000	5.2
<b>Accelerate (CPU)</b>	1200	38
<b>Accelerate (GPU)</b>	±1	4.1

# SALT MARSH CREEK FORMATION

Sediment

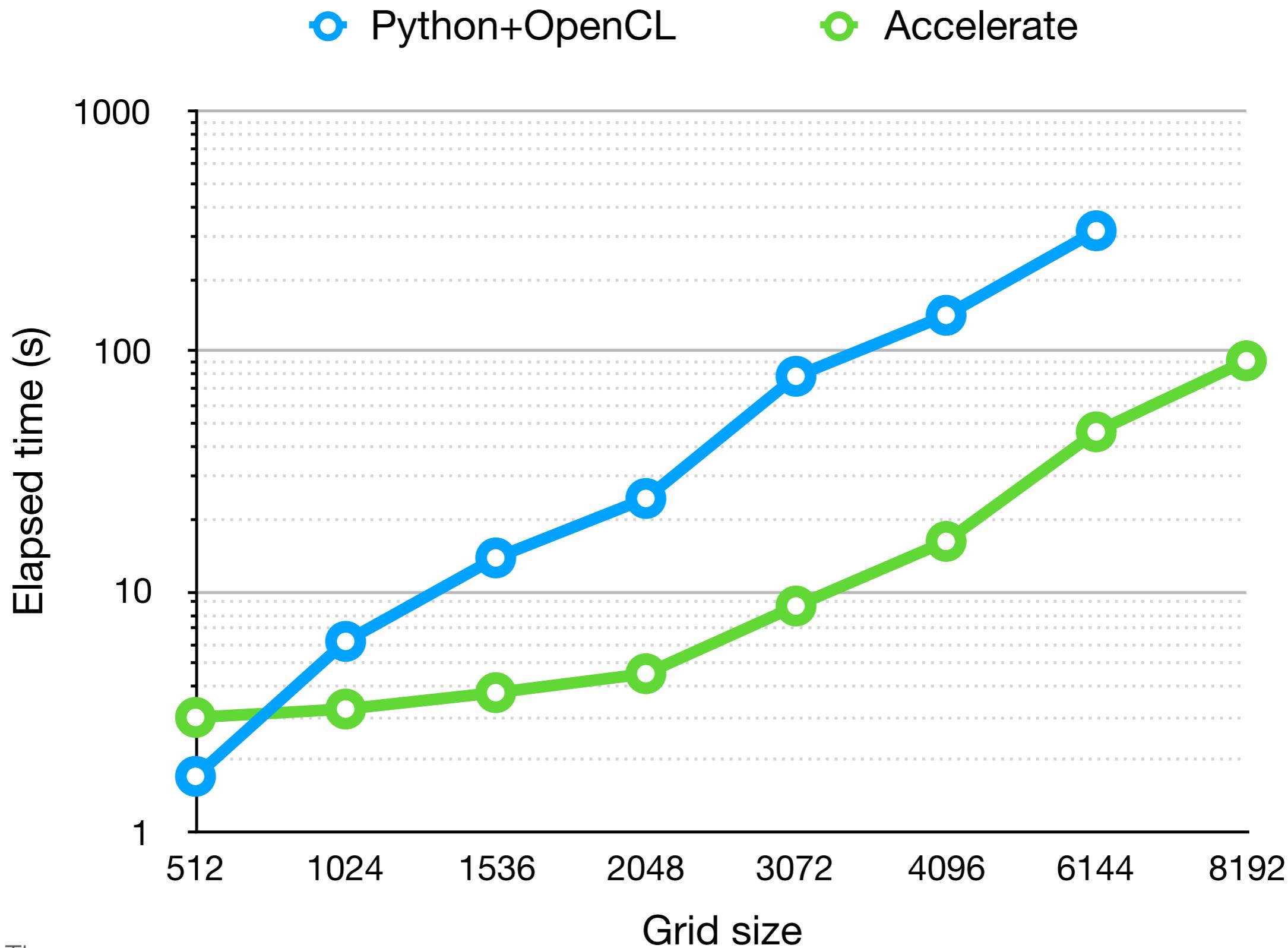


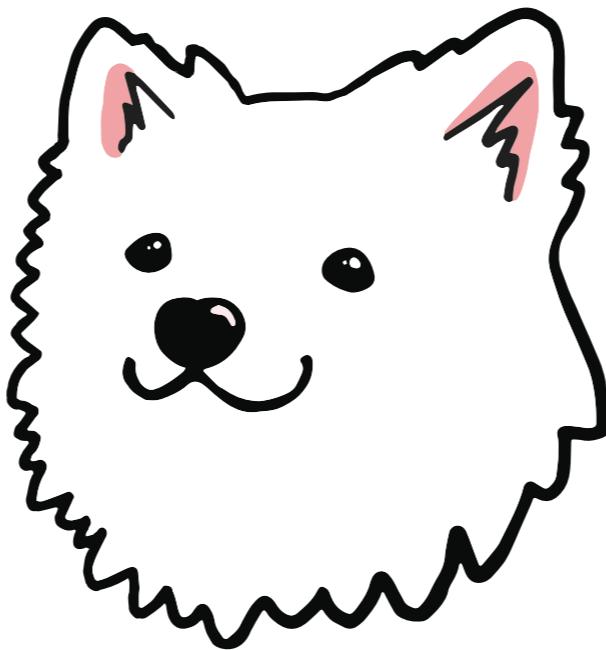
Water flow rate



Time: 40 of 2000

# SALT MARSH CREEK FORMATION





# acceleratehs.org

 <https://github.com/AccelerateHS/>

Trevor L. McDonell  
Robert Clifton-Everest

Manuel M.T. Chakravarty  
Josh Meredith

Gabriele Keller  
Ben Lippmeier