Software System Design and Implementation

Case Study: The Embedded Language *Accelerate*

Trevor L. McDonell

The University of New South Wales
School of Computer Science & Engineering
Sydney, Australia

Produce better software with less effort

- Better software
  - Fewer defects (e.g. security defects)
  - Software that is more usable
- Less effort
  - Shorter development time
  - Fewer programmers
  - Less-specialised programmers

Produce better software with less effort

- Types help in design & implementation
  - Program properties in types
  - Guide the design & imply programs
  - Prevent defects in the implementation

Parallel programming

- Perform many computations simultaneously in order to reduce overall processing time
  - Break large problems into smaller problems, solve each concurrently
  - Now the dominant paradigm for increasing processor performance (i.e. multicore CPUs)

Today's hardware is too hard!

- If it costs X (time, money, pain) to develop an efficient single-threaded algorithm, then…
  - Multithreaded version costs **2x**
  - PlayStation 3 Cell version costs **5x**
  - Current GPGPU version costs **10x** or more

Tim Sweeney (Epic Games)
High Performance Graphics, 2009

Can we have

parallel programming

with

less effort?

Haskell

Composite data structures

Immutable structures

Expressive type system & inference

Strong static typing

Higher-order functions & closures

Principled, pure, functional programming

Boxed values

Polymorphism & generics

Strictly isolating side-effects

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • CANBERRA • AUSTRALIA

Function pointers

Control flow

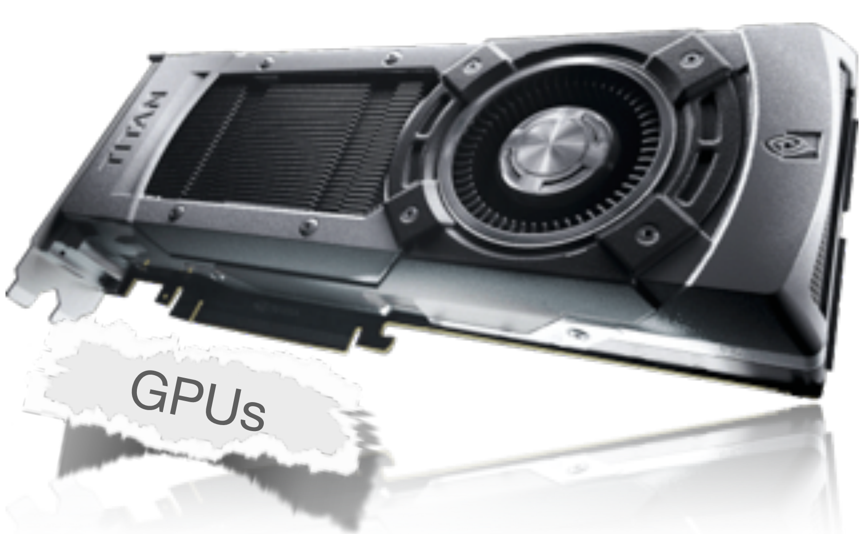
Memory access patterns

Decomposition

Haskell

Data distribution

Efficient code?



UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • CANBERRA • AUSTRALIA

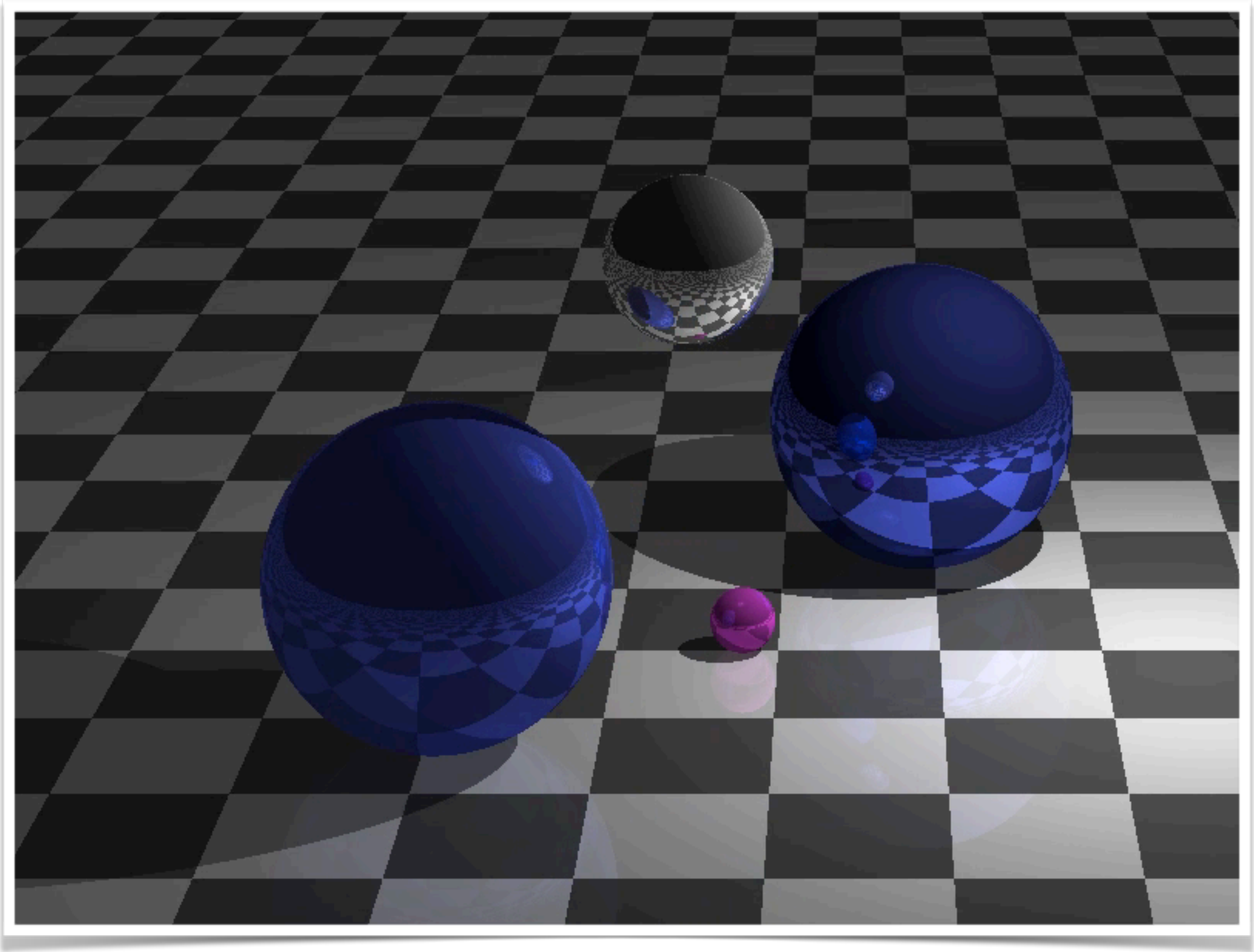
How about

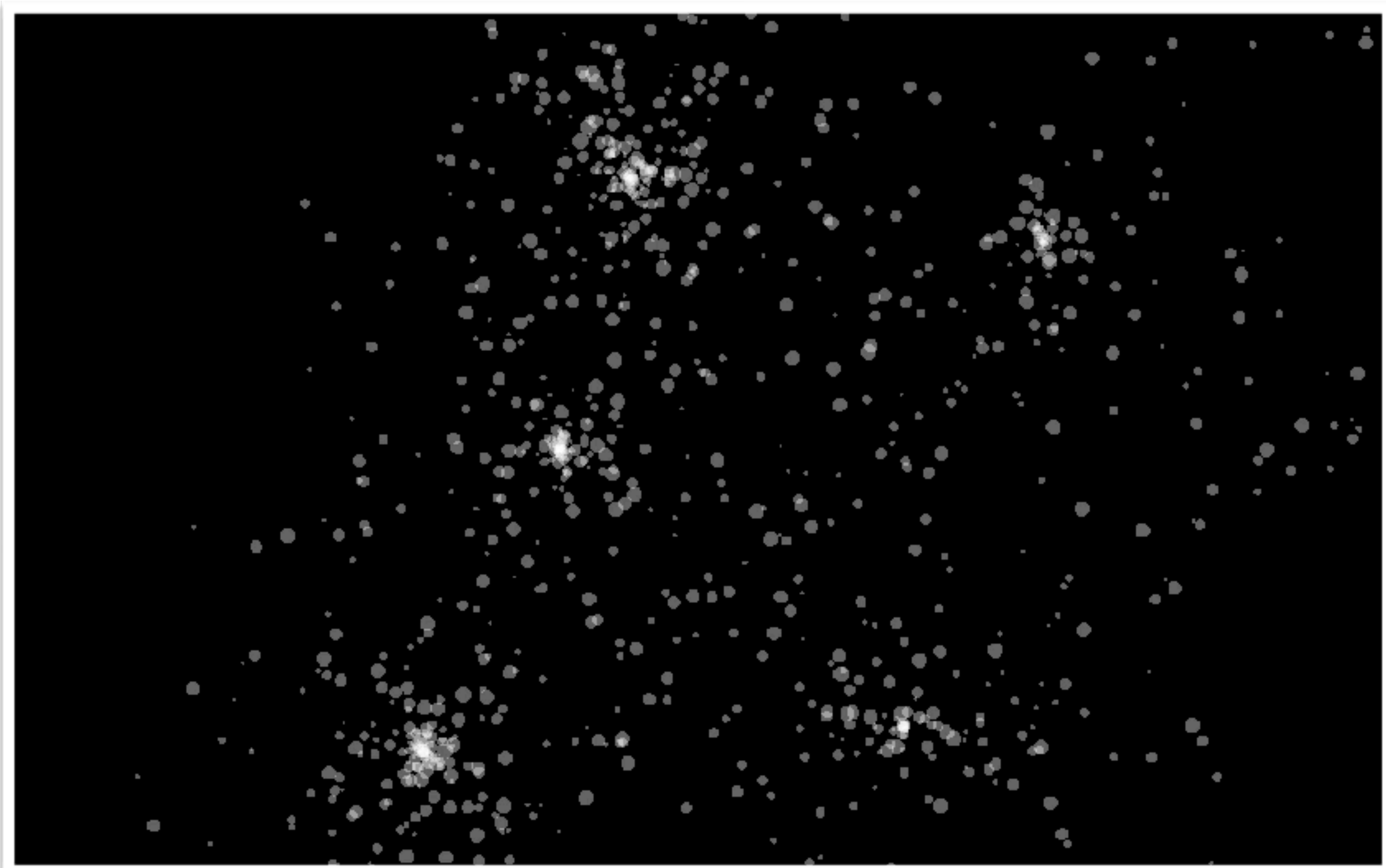
domain specific languages

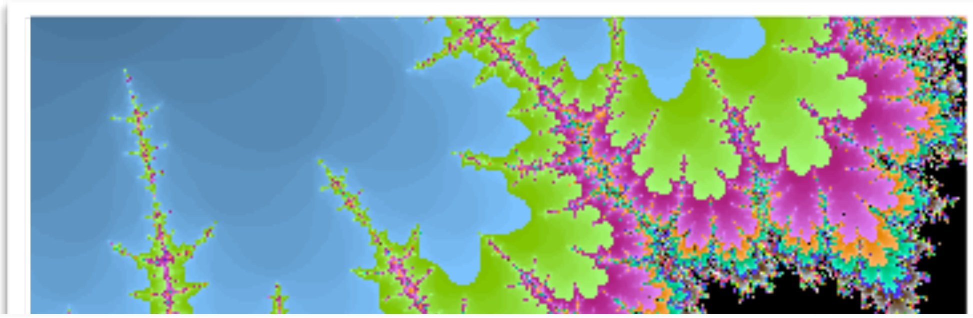
with

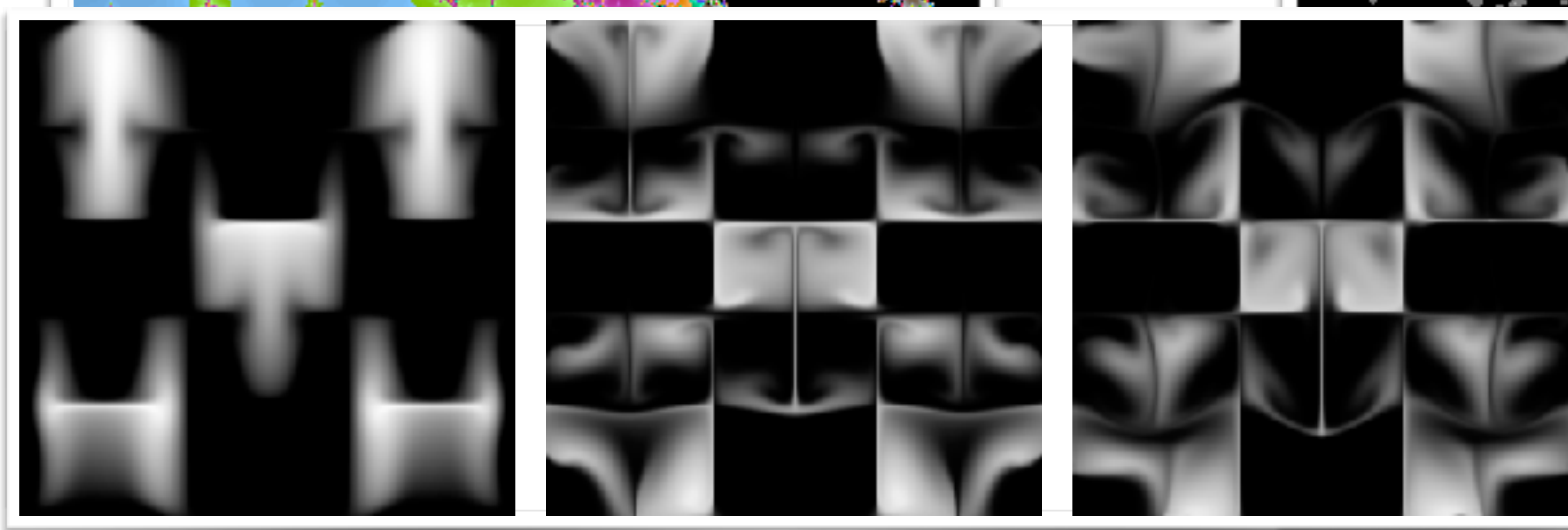
specialised code generation?

[demo]

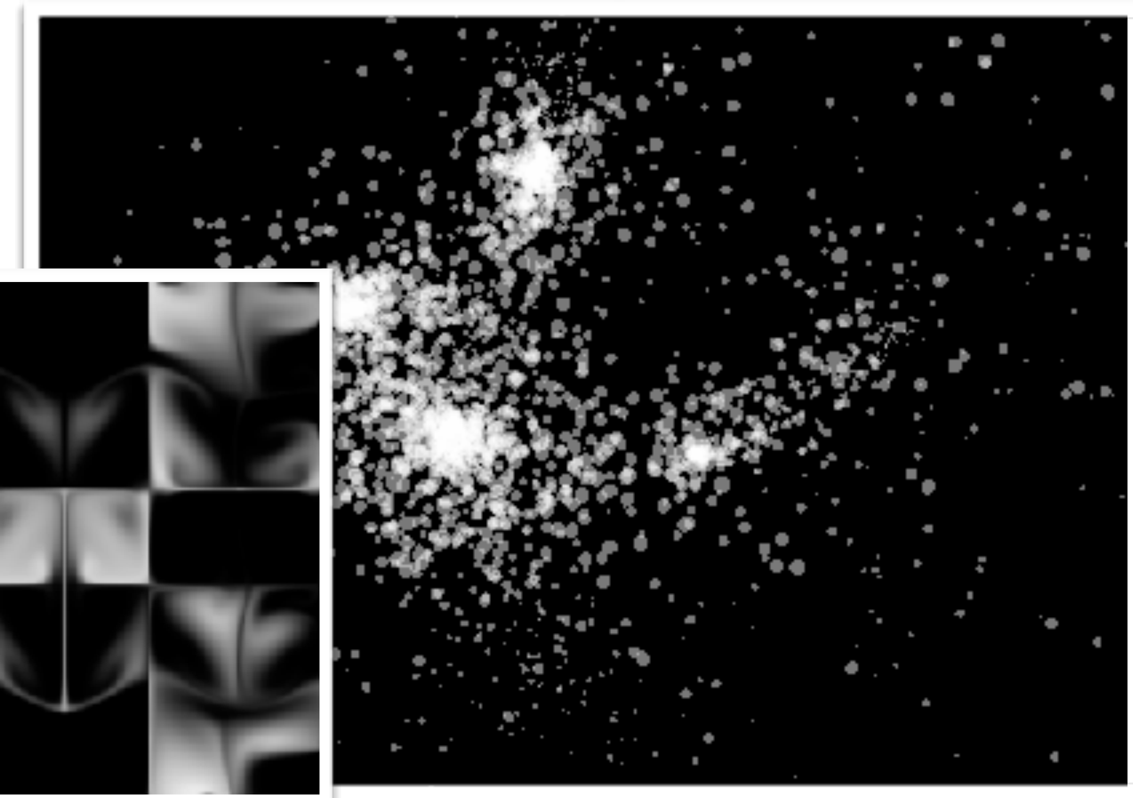




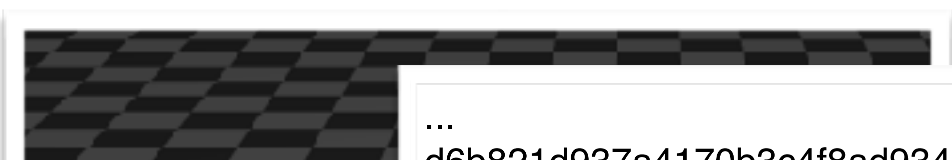




stable fluid flow



n-body gravitational simulation



SmoothLife cellular automata

```
...
d6b821d937a4170b3c4f8ad93495575d: saitek1
d0e52829bf7962ee0aa90550ffdcccaa: laura1230
494a8204b800c41b2da763f9bbbcc462: lina03
d8ff07c52a95b30800809758f84ce28c: Jenny10
e81bed02faa9892f8360c705241191ae: carmen89
46f7d75718029de99dd81fd907034bc9: mellon22
0dd3c176cf34486ec00b526b6920b782: helena04
9351c4bc8c8ba17b58d5a6a1f839f356: 85548554
9c36c5599f40d08f874559ac824d091a: 585123456
4b4dce6c91b429e8360aa65f97342e90: 5678go
3aa561d4c17d9d58443fc15d10cc86ae: momo55

Recovered 150/1000 (15.00 %) digests in 59.45 s, 185.03 MHash/sec
```

Password “recovery” (MD5 dictionary attack)

Canny edge detection

Embedded domain-specific languages

How to write specialised code with less effort

Domain specific languages

- Are restricted languages
  - Generally have specialised features to a particular application domain
  - HTML, Matlab, SQL, postscript …
- 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

Shallow vs. deep embeddings

- A shallow embedding directly executes functions in the host language
  - We don't get access to the program AST, we can only evaluate it
  - Easier to write — uses the binding constructs of the host language
- A deeply embedded reifies the program as a data structure
  - Can manipulate the entire program AST
  - But requires explicit handling of variables

Recall: the type-safe evaluator

Type-indexed expressions

```
data Expr t where
  Const ::        Int                         -> Expr Int
  Add   ::        Expr Int  -> Expr Int        -> Expr Int
  Equal :: Eq s => Expr s    -> Expr s           -> Expr Bool
  If    ::        Expr Bool -> Expr e   -> Expr e -> Expr e
```

Type-safe evaluation

```
eval :: Expr t -> t
eval (Const c)    = c
eval (Add e1 e2) = eval e1 + eval e2
  «and so on»
```

A very simple DSL!

Recall: the type-safe evaluator

- An embedded domain specific language for (very simple) arithmetic!
  - The language specifies a limited set of operations
  - Evaluator runs programs written in that language
- An example of a deeply embedded domain specific language
  - Operations in the language do not directly issue computations
  - Instead we reify the computation as a data structure — an abstract syntax tree

Extending the type-safe evaluator

- Support for more types?
  - Type safe operations, polymorphism

```
foo :: Num a
  => Exp a -> Exp a -> Exp a
```

- Writing programs in the language?
  - Don't want to write with explicit constructors

```
foo x y = 2 * (x + y)
```

```
let x =
  let y = foo x y
  in  ...
```

- Bindings and scope?

- Evaluating expressions on the CPU/GPU
  - What operations are allowable?

```
float foo(float x, float y)
{
    ...
```

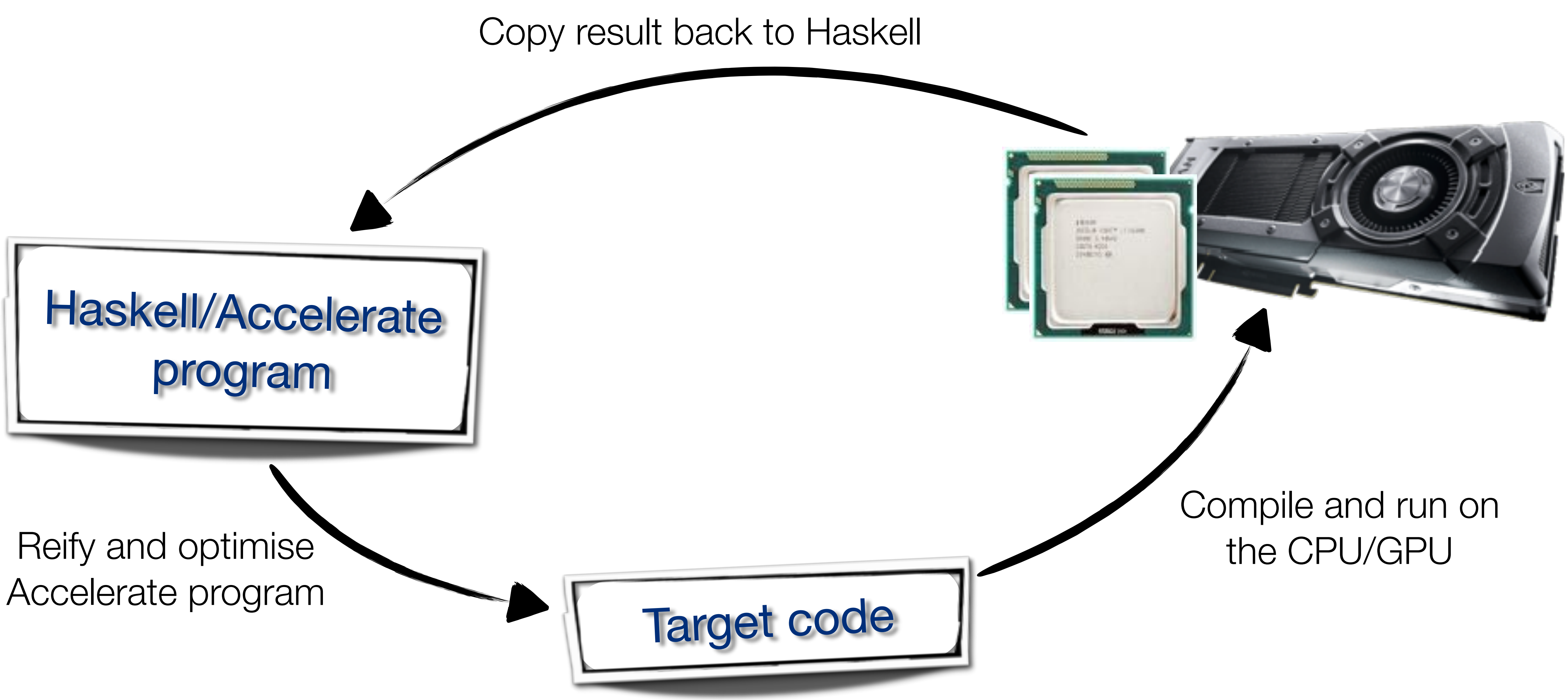
UNSW THE UNIVERSITY OF NEW SOUTH WALES SYDNEY • CANBERRA • AUSTRALIA

The Accelerate language

Design of an embedded language

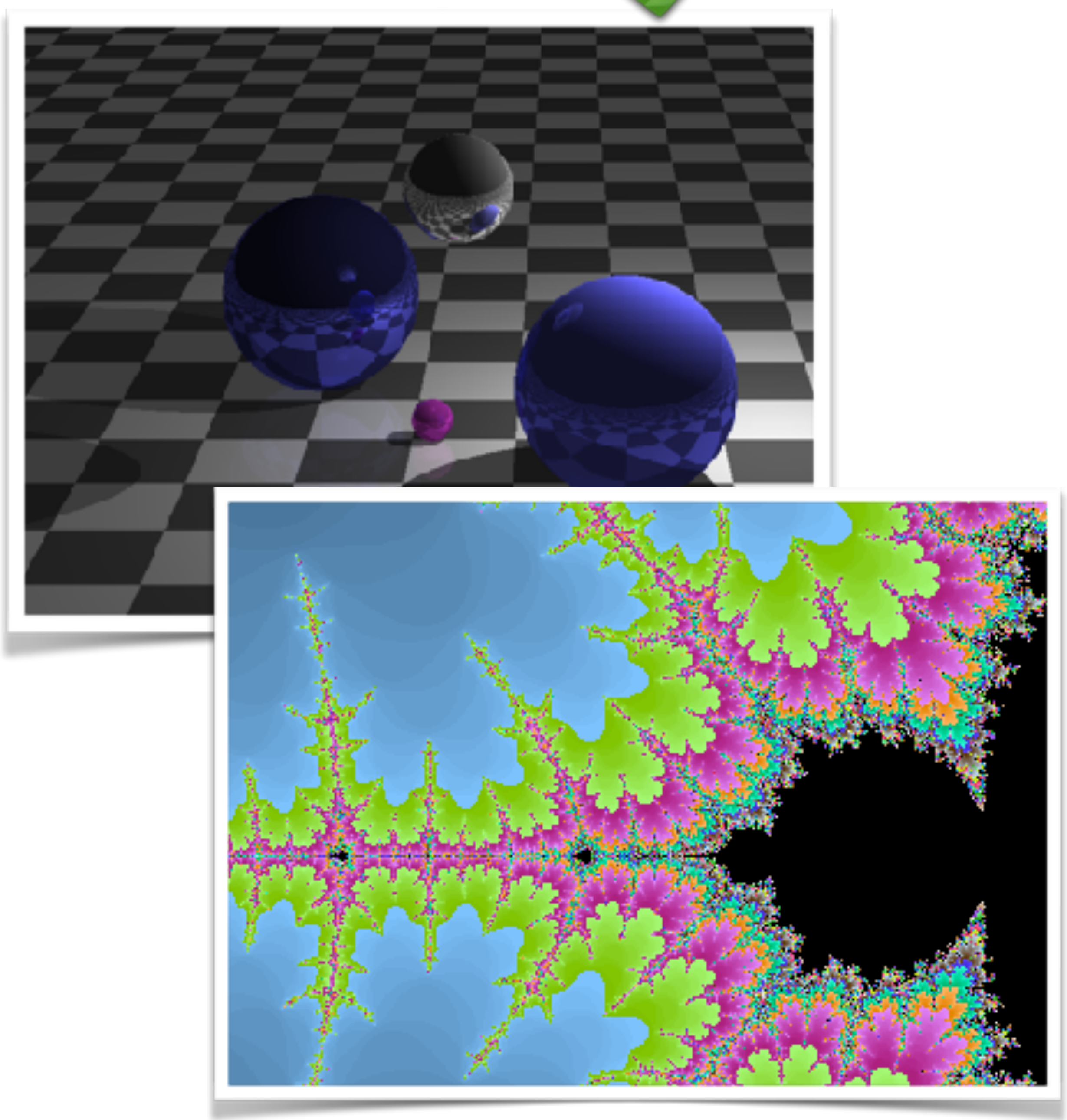
Accelerate

- An embedded domain-specific language for high-performance computing in Haskell



Accelerate is a domain specific language

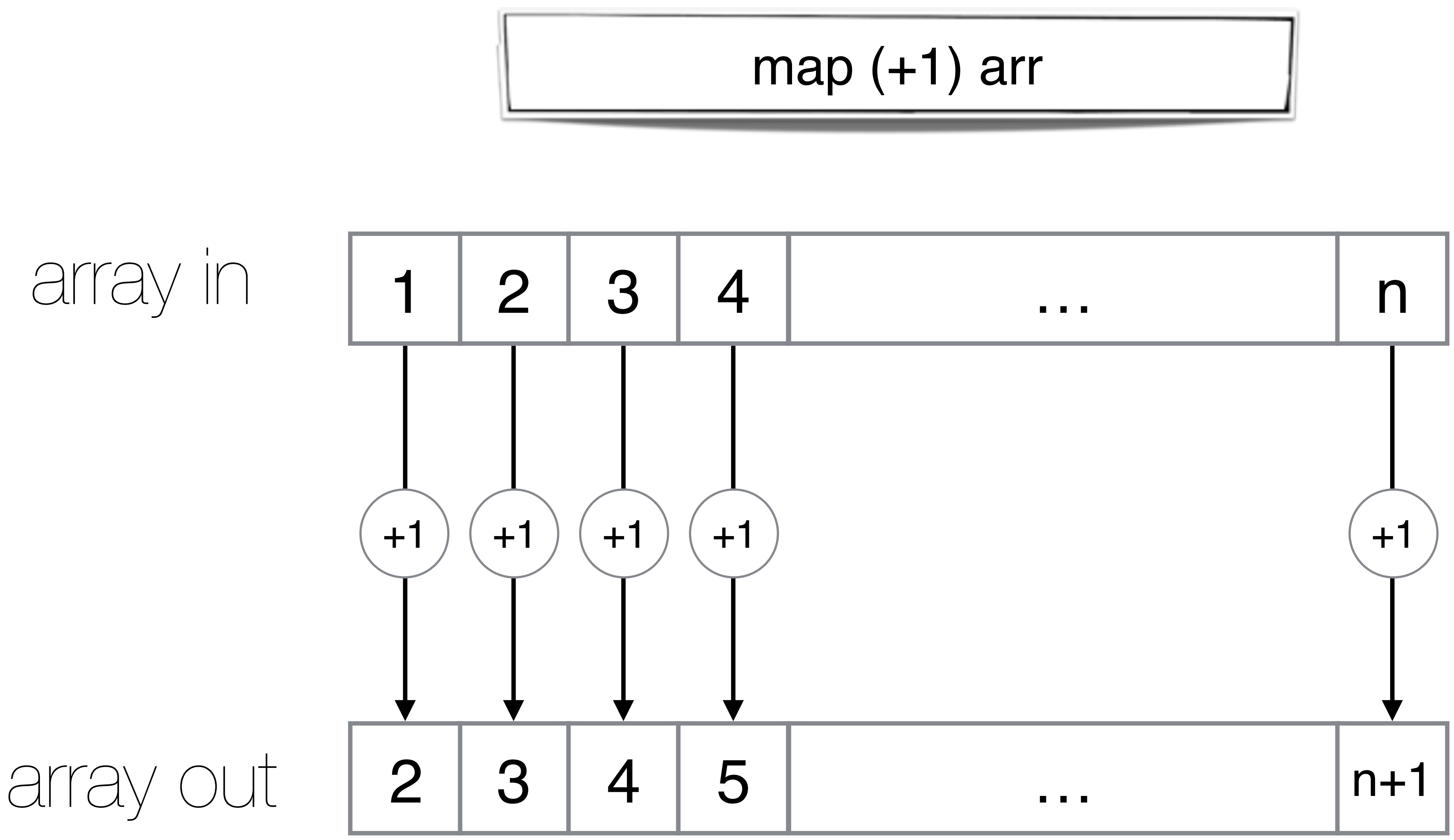
- Array computations
- Everything else



Mandelbrot fractal

Data parallelism

- Processors compute the same operation on many different data elements



Accelerate

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

Accelerate arrays

- Arrays have two type parameters
  - The dimensionally (aka shape) of the array
  - The element type of the array

Array sh e

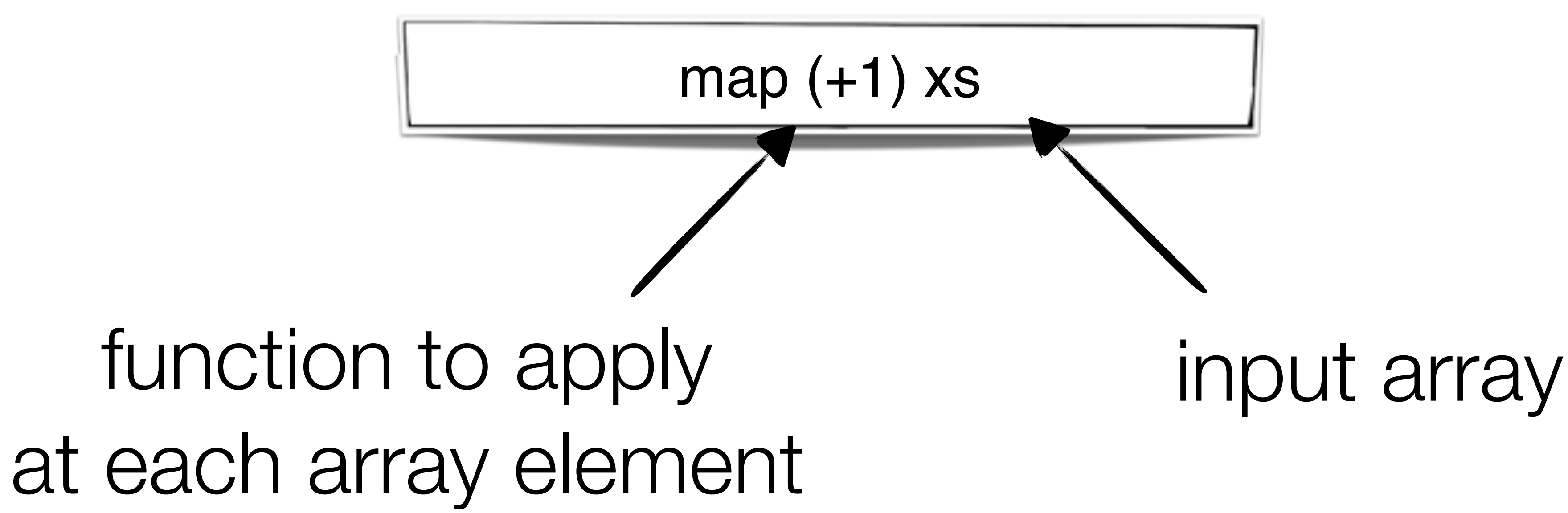
- But, specialised hardware such as GPUs often have restrictions
  - Parallel operations (kernels) can not launch more parallel operations*
  - Can we encode these restrictions into the language?

Accelerate arrays

- Allowable element types are members of the Elt class
  - ()
  - 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

Accelerate computations

- The types of array operations also statically excludes nested computations
  - A stratified language of scalar (Exp) and array (Acc) operations
  - Array computations consist of many scalar operations executed in parallel
  - Scalar operations can not contain further parallel operations



Accelerate computations

- What is the type of map?
  - map is an instance of the collective operations Acc, applying the scalar function in Exp to each element (in parallel)
  - Shape and Elt encapsulate allowable array index and element types

```
map :: (Shape sh, Elt a, Elt b)
    => (Exp a -> Exp b)
    -> Acc (Array sh a)
    -> Acc (Array sh b)
```

Embedding

```
map (+1) xs
```

- Acc is a GADT whose constructors represent collective operations
  - Writing a program with the Accelerate library amounts to constructing an AST representing that program
  - The AST can later be evaluated, or transformed into C code, etc…

```
map :: … -> Acc (Array sh b)
map = Map
```

```
data Acc a where
  Map :: (Shape sh, Elt a, Elt b)
      => (Exp a -> Exp b)
      -> Acc (Array sh a)
      -> Acc (Array sh b)
  «and many more»
```

Embedding

map (+1) xs

- 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

  «and many more»
```

Apply primitive scalar function: (+), (*) …

Embedding

map (+1) xs

- Overloaded the standard typeclasses 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)
  ...
```

Embedding

- Not all operations are valid for all types

```
(+) :: Num a      => a -> a -> a
div :: Integral a => a -> a -> a
sin :: Floating a => a      -> a
```

- How do we evaluate this?

```
eval :: (Num a, Integral a, Floating a) => Exp a -> a
```

Embedding

- 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

```
data IntegralDict a where
  IntegralDict :: ( Integral a, Num a, Eq a ... )
               => IntegralDict a

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

instance IsNum Int where
  numType = ...
```

GADTs

- 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
```

So far…

- Using types to guide the design
  - Only supports operations we know how to execute on restricted hardware
  - Stratification encodes the concept of data parallelism
- Type-safe, polymorphic operations
  - GADTs for a “type safe evaluator” style representation
  - Explicit dictionary passing to support ad-hoc polymorphism
- [Deeply] embedded languages reuse the host language syntax
  - Smart constructors that build AST terms
  - Overload standard typeclasses to reflect arithmetic operations

Properties in types

Encoding the type and scope of free variables

Surface language

- Our Acc and Exp terms are defined in Higher Order Abstract Syntax (HOAS)
  - Use the binding constructs of the host language

```
foo :: Exp a -> Exp b
foo x = ...
```

- But…
  - Does not explicitly represent variables
  - Can not peek into function bodies: can only apply functions

Internal language

- Need an explicit representation of bound and free variable names
  - Implies an explicit environment of bound terms
  - Allows us to inspect function bodies (intensional analysis)

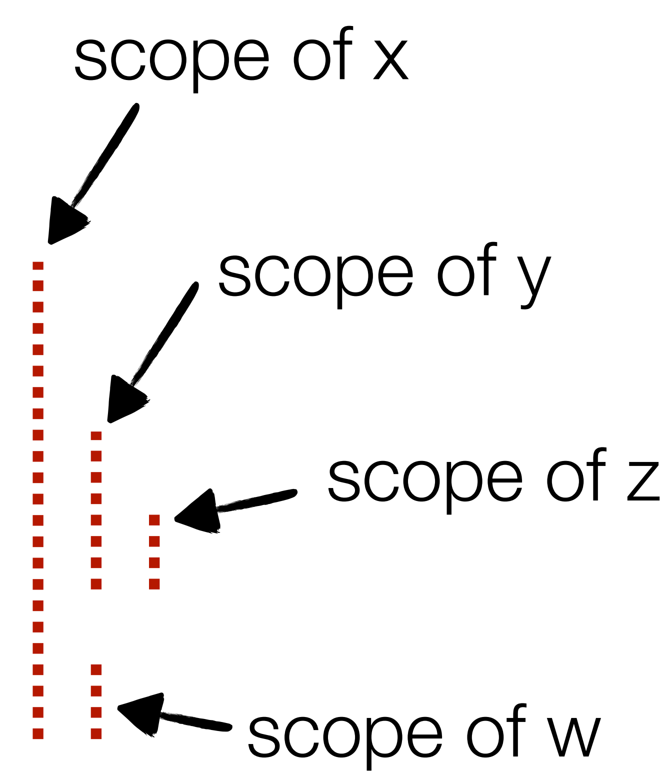
Can not depend on free scalar variables

```
data PreOpenAcc acc aenv a where
  Avar :: Arrays a => Idx aenv a -> PreOpenAcc acc     aenv a
  ...

data PreOpenExp acc env aenv t where
  Var  :: Elt t    => Idx env t  -> PreOpenExp acc env aenv t
  ...
```

Environments

- Environments keep track of what is in scope
  - To simplify code generation, define the binding as only being in scope while evaluating the body (in contrast to Haskell, let is not recursive)

```
foo x =
  let w =
    let y = 42    in
    let z = y * 2 in
      x + y + z
  in
  w * x
```



Environments

- Environments keep track of what is in scope

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

Nested datatypes & polymorphic recursion precisely enforce constraints

- A heterogenous snoc-list
  - Type: unit represents the empty environment, and the pair type for environments extended by an additional type
  - Value: snoc-list of terms that form the environment, newest on the right

De Bruijn indices

- A nameless way to represent variables
  - No variable capture: alpha-equivalence is just syntactic equivalence
  - Treat the environment as a stack of terms
  - The de Bruijn index just counts its place in the stack

Type list of terms
in the environment

```
data Idx env t where                                -- a variable is either
  ZeroIdx ::              Idx (env, top)  top -- at the top of the env; or
  SuccIdx :: Idx env t -> Idx (env, junk) t   -- under some junk
```

Can not create an index
into an empty environment

THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • CANBERRA • AUSTRALIA

De Bruijn indices

- Scalar function abstraction binds free variables
  - These are only introduced as arguments to collective operations
  - This restriction simplifies code generation: no closure conversion required

```
data PreOpenFun acc env aenv b where
  Lam  :: Elt a
       => PreOpenFun acc (env, a) aenv b
       -> PreOpenFun acc env      aenv (a -> b)

  Body :: Elt r
       => PreOpenExp acc env aenv r
       -> PreOpenFun acc env aenv r
```

De Bruijn indices

```
add :: Exp Int -> Exp Int -> Exp Int
add x y = x + y
```

```
add = \x -> \y -> PrimAdd numType `PrimApp` tup2 (x,y)
```

Introduce a new nameless variable

:: PreOpenExp acc (((), Int), Int) aenv Int

```
add = Lam (Lam (Body (
  PrimAdd (IntegralType ...)
  `PrimApp`
  Tuple (NilTup `SnocTup` (Var (SuccIdx ZeroIdx))
                `SnocTup` (Var ZeroIdx)))))
```

Wraps a de Bruijn index

De Bruijn indices

- Introduce a new nameless variable into the environment
  - Let-nodes represent sharing of sub terms
  - The type requires the binding is only in scope when evaluating the body

```
data PreOpenExp acc env aenv t where
  Var  :: Elt t => Idx env t -> PreOpenExp acc env aenv t

  Let  :: (Elt bnd, Elt body)
       => PreOpenExp acc env        aenv bnd
       -> PreOpenExp acc (env, bnd) aenv body
       -> PreOpenExp acc env        aenv body
  ...
```

Only in scope when evaluating the body

Environment projection

- How do we get a value out of the environment?
  - Recall that the environment is a heterogenous list
  - The index needs to recover both the position and type of the element

Under some junk

```
prj :: Idx env t -> Val env -> t
prj (SuccIdx idx) (Push env _) = prj idx env
prj ZeroIdx       (Push _   v) = v
prj _             Empty        = error "impossible"
```

why?

At the top

because **Empty :: Val ()**

Exercise: count the uses of each variable

- Traverse an expression searching for Var nodes
  - Generate a fresh name for each new binding
  - Use an environment to map names to counts

```
let x = 7        in
let x = x+1      in
let y = x*3 + x  in
  x + y + 2
```

→ de Bruijn notation

```
let v2 = 7            in
let v1 = v2+1         in
let v0 = v1*3 + v1    in
  v1 + v0 + 2
```

Exercise: count the uses of each variable

```haskell
type Name  = ...
data Count = Count { unique :: Int, counts :: Map Name Int }

data Ref env where
  Top  :: Ref ()
  Pop  :: Ref env -> Name -> Ref (env, s)

fresh :: State Count Name
touch :: Name -> State Count ()

lookupName :: Ref env -> Idx env t -> Name
lookupName (Pop _ n) ZeroIdx       = n
lookupName (Pop s _) (SuccIdx ix) = lookupName s ix
```

Similar to Val

encapsulate local mutable state

similar to prj

Exercise: count the uses of each variable

- Traverse the expression looking for Let and Var nodes
  - Must begin with a closed expression

```
usesOf :: OpenExp env aenv t -> Ref env -> State Count ()
usesOf exp env = case exp of
  Let bnd body -> do
    var <- fresh
    usesOf bnd  env
    usesOf body (Pop env var)

  Var idx       -> do
    touch (lookupName env idx)


  ...
```

Summary

- We use GADTs to very precisely specify types

```
data Val env where
  Empty ::                    Val ()
  Push  :: Val env' -> t -> Val (env', t)
```

```
data Idx env t where                          -- a variable is either
  ZeroIdx ::            Idx (env', top)  top -- at the top of the env; or
  SuccIdx :: Idx env' s -> Idx (env', junk) s   -- under some junk
```

```
prj :: Idx env t -> Val env -> t
prj (SuccIdx idx) (Push env _) = prj idx env
prj ZeroIdx       (Push _   v) = v
prj _             Empty        = error "impossible"
```

Executing embedded programs

Beyond the interpreter

Last time…

- Embedded languages
  - Restricted languages
  - Can reuse host language syntax (typeclass overloading)
  - Host language can compensate for restrictions in the embedded language
- Encoding properties in types
  - Use types to help guide a user in designing [data-parallel] programs
  - Hardware restrictions require no nested arrays: use a separate language for scalar (Exp) vs. collective array (Acc) operations

Executing programs

- The type-safe evaluator interprets programs step-by-step
  - Walk the AST recursively evaluating sub terms

```
eval :: Expr t -> t
eval (Const c)     = c
eval (Add e1 e2)  = eval e1 + eval e2
eval (Eq e1 e2)   = eval e1 == eval e2
eval (If p e1 e2) = if eval p then eval e1
                                else eval e2
```

Executing programs

- Instead of interpreting the expression
  - Convert the program into a form suitable for, say, GPU execution
  - Walk the AST generating C code or similar, then execute that code

```
run :: ExecOpenAcc aenv a -> Val aenv -> a
run (Map  objectcode gamma) aenv = ...
run (Fold objectcode gamma) aenv = ...
  ...
```

Executing programs

- Now we have a runtime compiler!
  - Since compilation happens at program runtime, having strong types in the embedded language means there are fewer possible runtime errors
  - But, must deal with code generation, caching, linking, calling the compiled code …

Algorithmic skeletons

- Collective operations in Acc are templates encapsulating specific behaviour
  - Parameterised by the scalar function they apply
  - Instantiate the operation by providing types and scalar expressions at predefined points

e.g. free variables

template holes

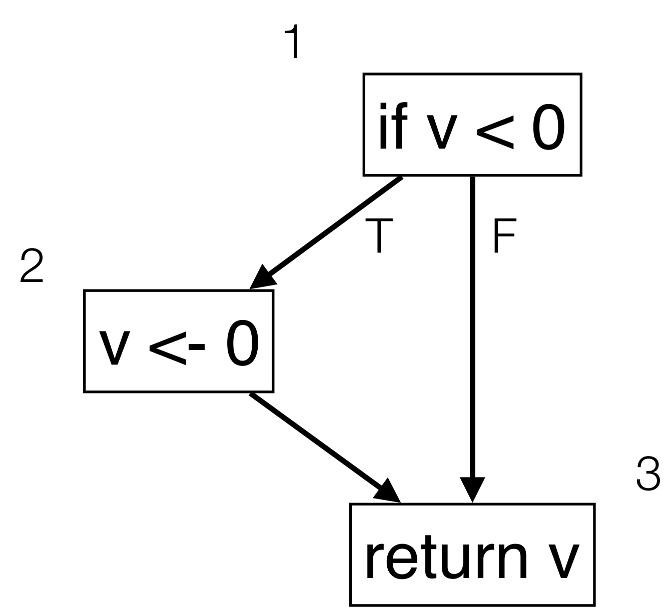
```
void map
(



){
        for ( int i = 0; i < end; ++i ) {
                x = arrIn[i];
        arrOut[i] = $function(
        }
}
```

apply embedded scalar function

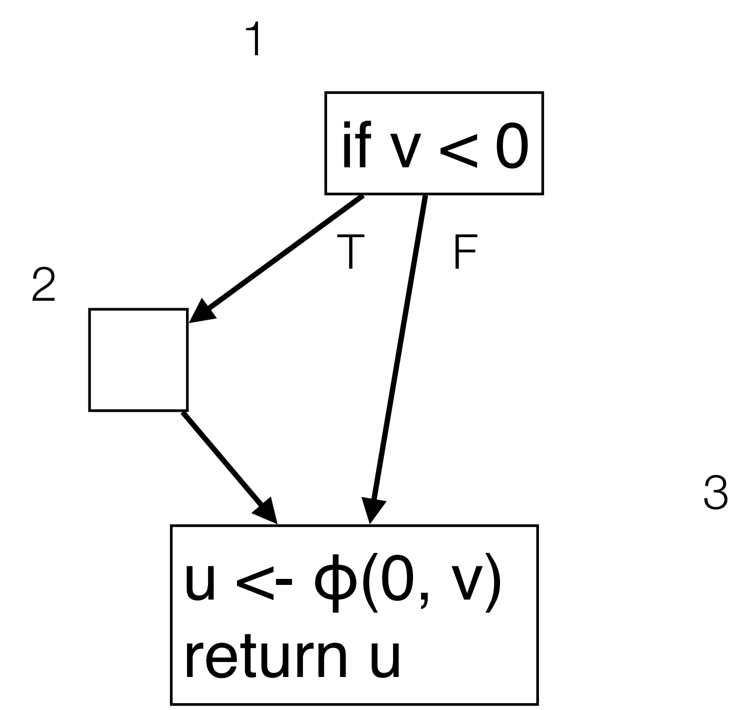
Static Single Assignment (SSA) form

- An intermediate representation where each variable is assigned exactly once, and every variable is defined before it is used
  - Designed to make optimisations efficient for imperative languages
  - A static property of program text, not a dynamic execution property

```
int relu( int v ) {
    if (v < 0) {
        v = 0
    }
    return v
}
```



CFG



SSA

Static Single Assignment (SSA) form

- Closely related to the lambda terms used by functional programs
  - *SSA is Functional Programming*
    Andrew Appel
  - *A Functional Perspective on SSA Optimisation Algorithms*
    Manuel M. T. Chakravarty, Gabriele Keller, Patryk Zadarnowski
- We can translate our first-order scalar language directly into SSA form
  - LLVM uses a statically typed intermediate representation in SSA form

Code generation

- Scalar code generation becomes a source-to-source translation
  - Translation preserves type information
  - Well typed source programs always generate well-typed target code
  - The llvm-hs library contains the necessary C++ bindings to LLVM

Code generation

(+1)

- Scalar code generation is a source-to-source translation
  - Convert accelerate expressions into form closer to LLVM instruction set
  - Lower type-level types into value-level types

```
plus1 = Lam (Body (
  PrimAdd (IntegralNumType (…))
  `PrimApp`
  Tuple (NilTup `SnocTup` (Var ZeroIdx)
              `SnocTup` (Const 1))))
```

accelerate

Code generation

if-then-else

- Branches and loops require insertion of ϕ-nodes
  - Need to create, keep track of basic block labels to use as branch targets

monad for fresh names, etc.

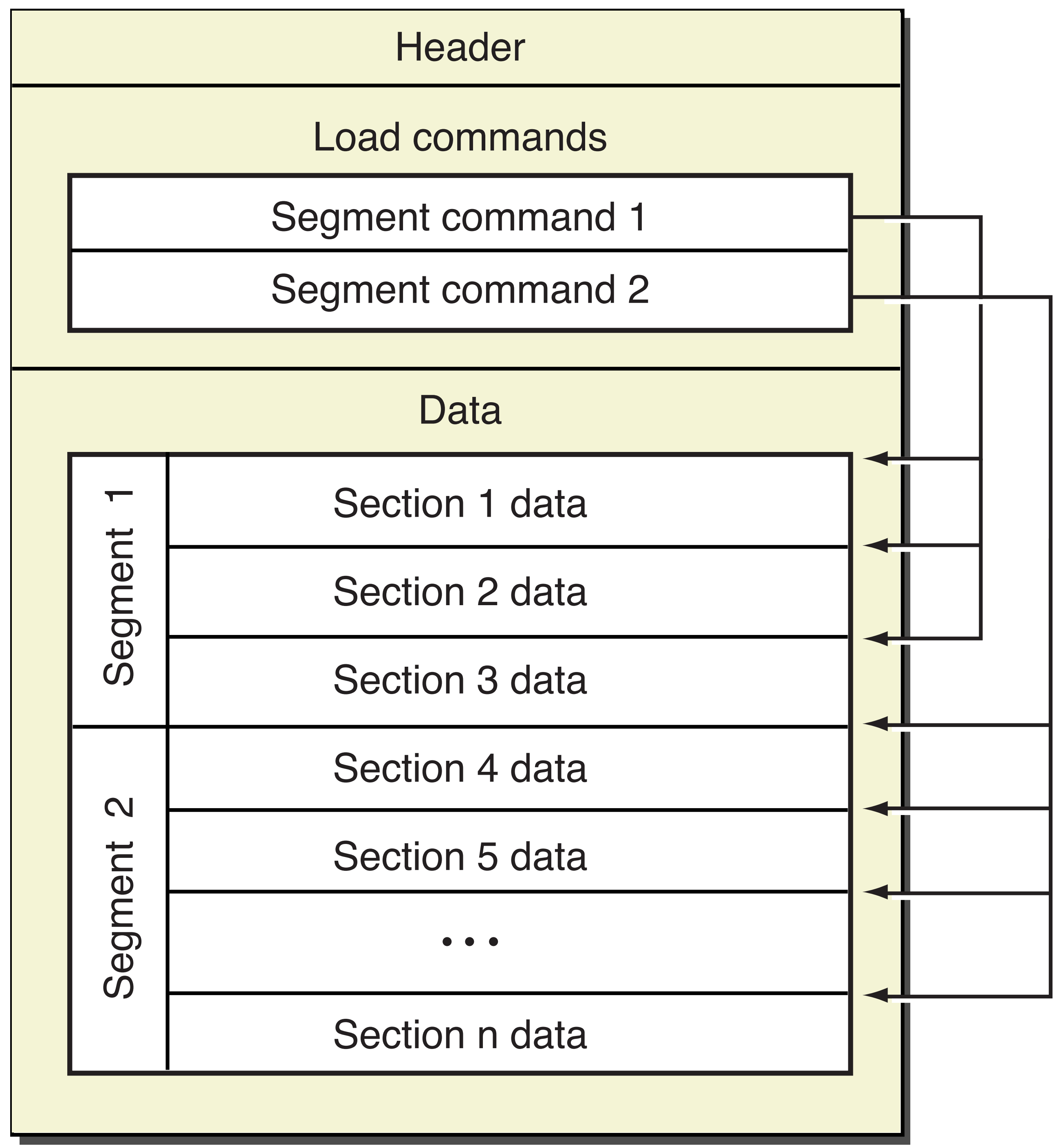
```
-- create a new basic block
newBlock :: String -> CodeGen Block

-- branch instructions return the block they came from
br  :: Block -> CodeGen Block
cbr :: IR Bool -> Block -> Block -> CodeGen Block

-- pick value depending on incoming edge
phi :: Elt a => [(IR a, Block)] -> CodeGen (IR a)
```

Runtime linking

- Finally, link the JIT compiled code into the running application
- We compile into a standard object file, rather than as a shared library
  - ELF (*nix): /usr/include/elf.h
  - MachO (MacOS): /usr/include/mach-o/loader.h
  - COFF (Windows): ¯_(ツ)_/¯

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • CANBERRA • AUSTRALIA

Mach-O file format



Relocations

- The process of assigning load addresses to position independent code
  - updates addresses/offsets from relocating the object code
  - resolving symbols to system library functions such as sin()

[demo]

Relocations

- The process of assigning load addresses to position independent code
  - updates addresses/offsets from relocating the object code
  - resolving symbols to system library functions such as sin()

- Intermediate jump islands can be used for > 32-bit displacement
  - initial 32-bit displacement to the jmp island, followed by long jump to actual target address

```
  0x0000000000000000    # target address
→ jmp *-14(%rip)        # relative instruction pointer
```

Summary

- Embedded domain specific languages are restricted languages
  - Reduce effort by generating code that embodies specialised knowledge
  - The embedding partly compensates for this restriction be seamlessly integrating with the host language
  - The host language can generate embedded code
- Types can be used to…
  - Encode properties and restrictions into the language
  - This can statically prevent writing programs which can not be compiled
  - Improve safety by eliminating sources of runtime failure

Accelerate

- Available on Hackage (hackage.haskell.org):
  - Core language: accelerate
  - CPU backend: accelerate-llvm-native
  - NVIDIA GPU backend: accelerate-llvm-ptx
  - Examples: accelerate-examples
- More information & short tutorial:
  - http://www.acceleratehs.org
- Contributions welcome! ^_^


fin.