Developing Haskell Applications

The Practice of Haskell Programming

Andres Löh

Well-Typed

June 17, 2012

Goals of this course

- Learning about Haskell's toolchain and extended infrastructure.
- Writing robust and scalable code.
- Reasoning about evaluation and performance.
- A few more advanced Haskell concepts.

Goals of this lecture

- Some stylistic guidelines and conventions when writing Haskell programs:
  - layout and syntax,
  - code structure,
  - common programming pitfalls.
- Modules vs. packages.
- Introduction to helpful development tools:
  - Cabal and cabal-install,
  - HLint,
  - GHC warnings,
  - Haddock.

Never use TABs

- Haskell uses layout to delimit language constructs.
- Haskell interprets TABs to have 8 spaces.
- Editors often display them with a different width.
- TABs lead to layout-related errors that are difficult to debug.
- Even worse: mixing TABs with spaces to indent a line.

So:

- Never use TABs.
- Configure your editor to expand TABs to spaces, and/or highlight TABs in source code.

Alignment

- Use alignment to highlight structure in the code!
- Do not use long lines.
- Do not indent by more than a few spaces.

```
map :: (a → b) → [a] → [b]
map f [ ]      = [ ]
map f (x : xs) = f x : map f xs
```


Well-Typed

Identifier names

- Use informative names for functions.
- Use CamelCase for long names.
- Use short names for function arguments.
- Use similar naming schemes for arguments of similar types.

Well-Typed

Spaces and parentheses

- Generally use exactly as many parentheses as are needed.
- Use extra parentheses in selected places to highlight grouping, particularly in expressions with many less known infix operators.
- Function application should always be denoted with a space.
- In most cases, infix operators should be surrounded by spaces.

Blank lines

- Use blank lines to separate top-level functions.
- Also use blank lines for long sequences of `let`-bindings or long `do`-blocks, in order to group logical units.

Avoid large functions

- Try to keep individual functions small.
- Introduce many functions for small tasks.
- Avoid local functions if they need not be local (why?).

Type signatures

- Always give type signatures for top-level functions.
- Give type signatures for more complicated local definitions, too.
- Use type synonyms.

```
checkTime :: Int → Int → Int → Bool
```

```
checkTime :: Hours → Minutes → Seconds → Bool
type Hours   = Int
type Minutes = Int
type Seconds = Int
```

Comments

- Comment top-level functions.
- Also comment tricky code.
- Write useful comments, avoid redundant comments!
- Use Haddock.

Booleans

Keep in mind that Booleans are first-class values.

Negative examples:

```
f x | isSpace x == True = ...
if x then True else False
```

Use (data)types!

- Whenever possible, define your own datatypes.
- Use `Maybe` or user-defined types to capture failure, rather than `error` or default values.
- Use `Maybe` or user-defined types to catpure optional arguments, rather than passing `undefined` or dummy values.
- Don't use integers for enumeration types.
- By using meaningful names for constructors and types, or by defining type synonyms, you can make code more self-documenting.

Use common library functions

- Don't reinvent the wheel. If you can use a `Prelude` function or a function from one of the basic libraries, then do not define it yourself.
- If a function is a simple instance of a higher-order function such as `map` or `foldr`, then use those functions (why?).

Pattern matching

- When defining functions via pattern matching, make sure you cover all cases.
- Try to use simple cases.
- Do not include unnecessary cases.
- Do not include unreachable cases.

Avoid partial functions

- Always try to define functions that are total on their domain, otherwise try to refine the domain type.
- Avoid using functions that are partial.

Negative example

```
if isJust x then 1 + fromJust x else 0
```

Use pattern matching!

Positive example

```
case x of
  Nothing → 0
  Just n  → 1 + n
```

Avoid partial functions – contd.

Negative example

```
map :: (a → b) → [a] → [b]
map f xs = if null xs then []
                      else f (head xs) : map f (tail xs)
```

Positive example

```
map :: (a → b) → [a] → [b]
map f []       = []
map f (x : xs) = f x : map f xs
```


Well-Typed

Use let instead of repeating complicated code

Write

```
let x = foo bar baz in x + x * x
```

rather than

```
foo bar baz + foo bar baz * foo bar baz
```

Questions

- Is there a semantic difference between the two pieces of code?
- Could/should the compiler optimize from the second to the first version internally?

Well-Typed

Let the types guide your programming

- Try to make your functions as generic as possible (why?).
- If you have to write a function of type $\mathsf{Foo} \rightarrow \mathsf{Bar}$, consider how you can destruct a Foo and how you can construct a Bar.
- When you tackle an unknown problem, think about its type first.

Well–Typed

Let the types guide your programming – contd.

A function on lists:

```
length :: [a] → Int
length xs      = ...
```

Look at the **type** of the input:

- xs is a list, so we can **pattern match**.

Well–Typed

Let the types guide your programming – contd.

A function on lists:

$$\begin{aligned}&\mathrm{length} :: [a] \to \mathrm{Int}\\&\mathrm{length}\ [\,] \quad\;\; = \boxed{\ldots}\\&\mathrm{length}\ (x : xs) = \ \ldots\end{aligned}$$

Follow the structure of the types:

- one pattern per constructor,
- try to **recurse** where the datatype is recursive!

The base case is easy to solve here.

Let the types guide your programming – contd.

A function on lists:

$$\begin{aligned}&\mathrm{length} :: [a] \to \mathrm{Int}\\&\mathrm{length}\ [\,] \quad\;\; = 0\\&\mathrm{length}\ (x : xs) = \ \ldots\end{aligned}$$

Let us consider the second case:

- The `xs` are a (shorter) list.
- Let us try to recurse.

Let the types guide your programming – contd.

A function on lists:

```
length :: [a] → Int
length []       = 0
length (x : xs) = ... length xs ...
```

It is now easy to complete the definition.

Let the types guide your programming – contd.

A function on lists:

```
length :: [a] → Int
length []       = 0
length (x : xs) = 1 + length xs
```

Done.

Let the types guide your programming – contd.

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

A function on trees:

```
size :: Tree a → Int
size t        = ...
```

Look at the **type** of the input:

- t is a tree, so we can **pattern match**.

Let the types guide your programming – contd.

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

A function on trees:

```
size :: Tree a → Int
size (Leaf x)   = ...
size (Node l r) = ...
```

Again, we follow the structure of the typr

- one pattern per constructor,
- try to **recurse** where the datatype is recursive!

Again, we have an easy base case.

Let the types guide your programming – contd.

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

A function on trees:

```
size :: Tree a → Int
size (Leaf x)   = 1
size (Node l r) = ...
```

Let us consider the second case:

- Both `l` and `r` are (smaller) trees.
- Let us try to recurse (twice).

Let the types guide your programming – contd.

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

A function on trees:

```
size :: Tree a → Int
size (Leaf x)   = 1
size (Node l r) = ... size l ... size r ...
```

It is now easy to complete the definition.

Let the types guide your programming – contd.

```haskell
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

A function on trees:

```haskell
size :: Tree a → Int
size (Leaf x)   = 1
size (Node l r) = size l + size r
```

Done.

Goals of Cabal

- A build system for Haskell applications and libraries.
- Easy to use for developers and users.
- Specifically tailored to the needs of a "normal" Haskell package.
- Tracks dependencies between Haskell packages.
- A unified package description format that can be used by a database.
- Platform-independent.
- Compiler-independent.
- Generic support for preprocessors, inter-module dependencies, etc. (`make` replacement).

Cabal is still in development; some goals have been reached, others not quite.

Cabal

- Cabal is itself packaged using Cabal.
- Cabal is integrated into the set of packages shipped with GHC, so if you have GHC, you have Cabal as well.

Homepage

`http://haskell.org/cabal/`

A Cabal package description

(Show on Hackage.)

A Setup file

```haskell
import Distribution.Simple
main = defaultMain
```

In almost all cases, this together with a Cabal file is sufficient.

If you need to do extra stuff (for instance, install some additional files that have nothing to do with Haskell), there are variants of `defaultMain` that offer hooks.

Hackage

- Online Cabal package database.
- Everybody can upload their Cabal-based Haskell packages.
- Automated building of packages.
- Allows automatic online access to Haddock documentation.

`http://hackage.haskell.org/`

Well-Typed

Cabal vs. cabal-install

- The `Cabal` package provides a library with functions to support the packaging of Haskell libraries and tools.
- In particular, it specifies the format of the `.cabal` package description files.
- The `cabal-install` package provides a frontend to the `Cabal` library, providing the user with several commands to work with Cabal packages.
- Somewhat confusingly, the frontend contained in `cabal-install` is called `cabal`.
- The `cabal-install` package is contained in the Haskell Platform.

cabal-install

- A frontend to Cabal.
- Resolves dependencies of packages automatically, then downloads and installs all of them.
- Once cabal-install is present, installing a new library from Hackage is usually as easy as:

```
$ cabal update
$ cabal install <packagename>
```

- You can also run `cabal install` within a directory containing a `.cabal` file.

Creating your own Cabal package

- Create a directory.
- Write initial program.
- Put it into version control (Subversion, git, darcs).
- Add a `.cabal` file.
- Add a `Setup.hs` or `Setup.lhs` file.
- Build using Cabal.
- Generate Haddock documentation using Cabal.
- Add a test suite.
- Use your version control system to run test suite on every commit (currently preferred by the Haskell community: git and darcs).

Preparing a release

- Tag the version in your version control system.
- Create a tarball via Cabal (or darcs).
- Upload to HackageDB (supported by the cabal frontend).

More details

`http://en.wikibooks.org/wiki/Haskell/Packaging`

HLint

- A simple tool to improve your Haskell style.
- Developed by Neil Mitchell.
- Scans source code, provides suggestions.
- Makes use of generic programming (Uniplate).
- Suggests only correct transformations.
- New suggestions can be added, and some suggestions can be selectively disabled.
- Easy to install (via `cabal install`).

Demo

(Demo.)

GHC warnings

GHC can warn you about lots of potential mistakes:

- shadowing of identifier names,
- unused code,
- redundant module imports,
- non-exhaustive patterns in functions,
- use of deprecated functions,
- ...

By default, only a small fraction of these warnings are generated:

- use `-Wall` to enable all warnings,
- there are flags to selectively enable or disable specific sets of warnings.

Haddock

- Haddock is a documentation generator for Haskell (like JavaDoc, Doxygen, ...)
- Parses annotated Haskell files.
- Most of GHC's language extensions are supported.
- API documentation (mainly).
- Program documentation (possible).
- HTML output.

Haddock annotations

```
-- | This is (redundant) documentation for
-- function 'f'. The function 'f' is a badly
-- named substitute for the normal
-- /identity/ function 'id'.
f :: a → a
f x = x
```

- A `-- |` declaration affects the following top-level declaration.
- Single quotes as in `'f'` indicate the name of a Haskell function, and cause automatic hyperlinking. Referring to qualified names is also possible (even if the identifier is not normally in scope).
- Emphasis with forward slashes: `/identity/`.

More markup

Haddock supports several more forms of markup, for instance

- Sectioning to structure a module.
- Code blocks in documentation.
- References to whole modules.
- Itemized, enumerated, and definition lists.
- Hyperlinks.

Example

(Show on Hackage.)

Data Structures

The Practice of Haskell Programming

Andres Löh

Well-Typed

June 17, 2012

Overview

- Persistent data structures
- Arrays
- Trees, sets and finite maps
- Other useful data structures

Imperative vs. functional style

Given a finite map (associative map, dictionary) `m`.

Imperative style

foo.put (42, "Bar"); . . .

Functional style

let foo′ = insert 42 "Bar" m **in** . . .

What is the difference?

Imperative: destructive update

Functional: creation of a new value

Persistent data structures

Imperative languages:

- many operations make use of destructive updates
- after an update, the old version of the data structure is no longer available

Functional languages:

- most operations create a new data structure
- old versions are still available

Data structures where old version remain accessible are called **persistent**.

Persistent data structures

- In functional languages, most data structures are (automatically) persistent.
- In imperative languages, most data structures are not persistent (**ephemeral**).
- It is generally possible to also use ephemeral data structures in functional or persistent data structures in imperative languages.

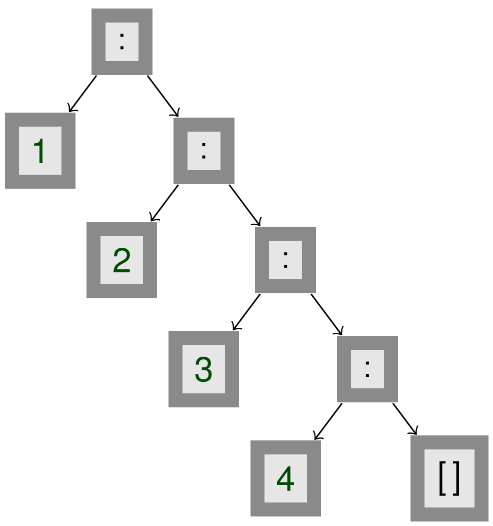
How do persistent data structures work?

Well-Typed

Example: Haskell lists

$[1, 2, 3, 4]$ is syntactic sugar for $1 : (2 : (3 : (4 : [])))$

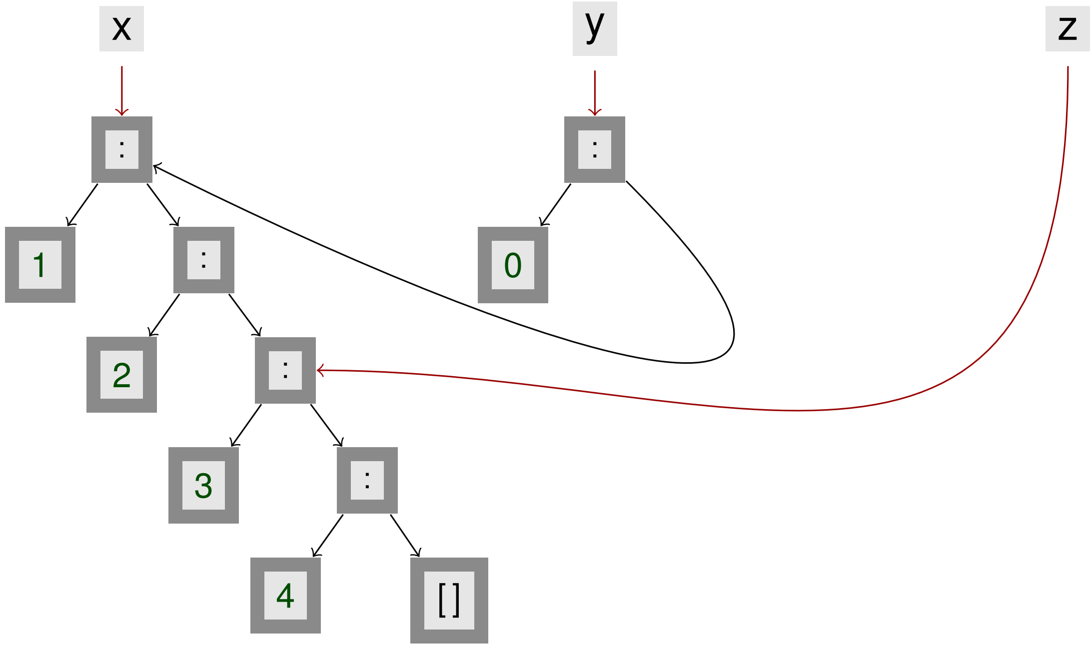
Representation in memory:



Well-Typed

Lists are persistent

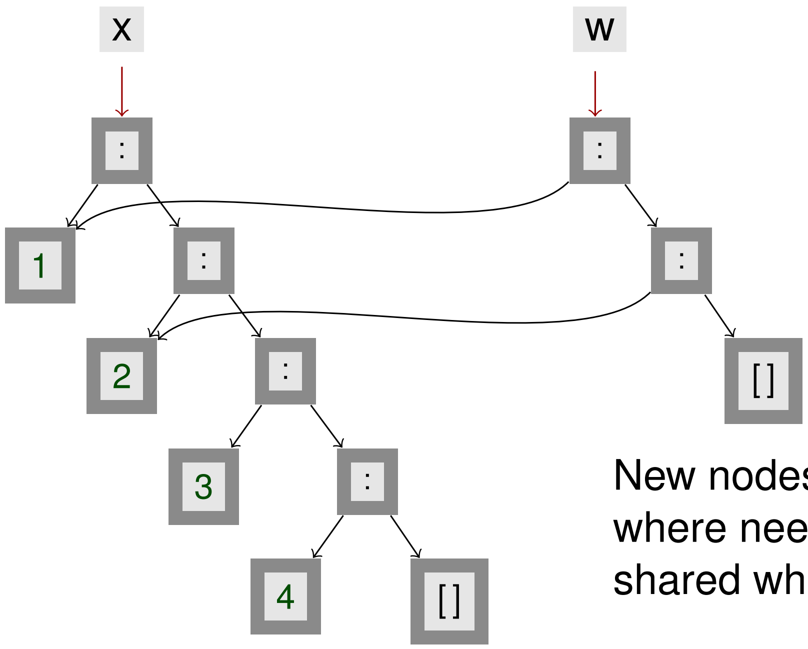
let x = [1, 2, 3, 4]; y = 0 : x; z = drop 3 y **in** . . .



Well-Typed

Lists are persistent – contd.

let x = [1, 2, 3, 4]; w = take 2 x **in** . . .



New nodes are allocated where needed; nodes are shared where possible.

Well-Typed

Implementation of persistent data structures

- Modifications of an existing structure take place by creating new nodes and pointers.
- Sometimes, parts of a structure have to be copied, because the old version must not be modified.

Of course, we want to copy as little as possible, and reuse as much as possible.

Well-Typed

Vacuum

Vacuum is a library originally developed by Matt Morrow:

- the library is a debugging tool,
- we can query and generate the internal graph representation of Haskell terms,
- useful to understand how Haskell terms are shared.

There are several visualization layers for vacuum available from Hackage. Unfortunately, many of them are somewhat tricky to build.

```
〉 view (let x = [1, 2] in x ++ x)
〉 view (let x = [1, 2, 3, 4] y = 0 : x; z = drop 3 y in (x, y, z))
〉 view (let x = [1, 2, 3, 4]; w = take 2 x in (x, w))
〉 view (repeat 1)
```


Well-Typed

Persistence and complexity

Some data structures show unexpected (i.e., bad) behaviour when used in a persistent setting:

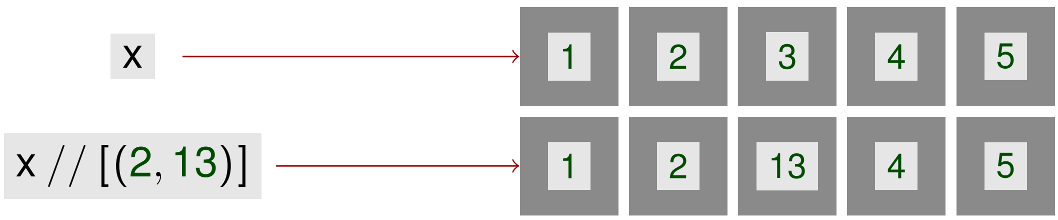
Haskell arrays:

```
let x = listArray (0, 4) [1, 2, 3, 4, 5] in x // [(2, 13)]
```

How expensive is the update operation?

- In an imperative language, we expect O(1), i.e., constant time.

Arrays



- Arrays are stored in a contiguous block of memory.
- This allows O(1) access to each element.
- In an imperative setting, a destructive update is also possible in O(1).
- But if a persistent update is desired, the whole array must be copied, which takes O(n), i.e., linear time.

Advice on arrays

Be careful when using them:

- stay away if you require a large number of incremental updates – finite maps are usually much better then;
- arrays can be useful if you have an essentially constant table that you need to access frequently;
- arrays can also be useful if you perform global updates on them anyway.

The `vector` package

There's a new, quite popular array package available from Hackage called `vector`:

- Developed by Roman Leshchinskiy.
- An interface capturing mutable and immutable arrays, boxed and unboxed arrays in a slightly more systematic way than the standard Haskell array interface allows.
- Support slicing operations.

Trees

- Arrays and hash tables are expensive in a functional (persistent) setting, because it is impossible to share substructures between different versions.
- Tree-shaped structures, however, are generally very suitable in a functional setting. Reuse of subtrees is easy to achieve. Most functional data structures therefore are some sort of trees.

Lists are trees, too – just a very peculiar variant.

Lists

- There is a lot of syntactic sugar for lists in Haskell. Thus, lists are used for a lot of different purposes.
- Lists are the default data structure in functional languages much as arrays are in imperative languages.
- However, lists support only **very few operations efficiently**.

Operations on lists

```
[]      :: [a]                              -- O(1)
(:)     :: a → [a] → [a]                    -- O(1)
head    :: [a] → a                          -- O(1)
tail    :: [a] → [a]                        -- O(1)
snoc    :: [a] → a → [a]                    -- O(n)
snoc    = λxs x → xs ++ [x]
(!!)    :: [a] → Int → a                    -- O(n)
(++)    :: [a] → [a] → [a]                  -- O(m), first list
reverse :: [a] → [a]                        -- O(n)
splitAt :: Int → [a] → ([a], [a])           -- O(n)
union   :: Eq a ⇒ [a] → [a] → [a]           -- O(mn)
elem    :: Eq a ⇒ a → [a] → Bool            -- O(n)
```


Guidelines for using lists

Lists are suitable for use if:

- most operations we need are **stack operations**,
- or the maximal size of the lists we deal with is relatively small,

A special case of stack-like access is if we traverse a large list linearly.

Lists are generally not suitable:

- for random access,
- for set operations such as union and intersection,
- to deal with (really) large amounts of texts as `String`.

What is better than lists?

Are there functional data structures that support a more efficient lookup operation than lists?

Yes, balanced search trees.

Can be used to implement finite maps and sets efficiently, and persistently.

Question

What is a (binary) search tree?

Well-Typed

Finite maps

- A finite map is a function with a finite domain (type of **keys**).
- Useful for a wide variety of applications (tables, environments, "arrays").
- Inefficient representation: `type Map a b = [(a, b)]`.

Well-Typed

An efficient implementation of finite maps

- Based on binary search trees.
- Available in `Data.Map` and `Data.IntMap` for `Int` as key type.
- Provided by the `containers` package that is part of the Haskell Platform.
- Keys are stored ordered in the tree, so that efficient lookup is possible.
- Requires the keys to be ordered.
- Inserting and removing elements can trigger rotations to rebalance the tree.
- Everything happens in a persistent setting.

Well-Typed

Sets

- Sets are a special case of finite maps: essentially,

```
type Set a = Map a ()
```

- A specialized set implementation is available in `Data.Set` and `Data.IntSet`, but the idea is the same as for finite maps.

Well-Typed

Finite map interface

This is an excerpt from the functions available in `Data.Map`:

```
data Map k a       -- abstract

insert  :: (Ord k) ⇒ k → a → Map k a → Map k a       -- O(log n)
lookup  :: (Ord k) ⇒ k → Map k a → Maybe a           -- O(log n)
delete  :: (Ord k) ⇒ k → Map k a → Map k a           -- O(log n)
update  :: (Ord k) ⇒ (a → Maybe a) →
                     k → Map k a → Map k a           -- O(log n)
union   :: (Ord k) ⇒ Map k a → Map k a → Map k a     -- O(m + n)
member  :: (Ord k) ⇒ k → Map k a → Bool              -- O(log n)
size    :: Map k a → Int                             -- O(1)
map     :: (a → b) → Map k a → Map k b               -- O(n)
```

The interface for `Set` is very similar.

Well-Typed

Implementation

In the following, we will sketch the implementation as it is available in `Data.Map`:

```
data Map k a = Tip
             | Bin !Size (Map k a) k a (Map k a)
type Size = Int
```

The `!` is a strictness annotation for extra efficiency. More about that later.

A map is

- either a leaf called `Tip`,
- or a binary node called `Bin` containing
  - the size of the tree,
  - the key value pair,
  - and a left and right subtree.

Well-Typed

Creating finite maps

```
empty :: Map k a
empty = Tip

singleton :: k → a → Map k a
singleton k x = bin Tip k x Tip
```

The function `bin` is an example of a **smart constructor** . . .

Well-Typed

Smart constructors

Smart constructors are wrappers around constructors that help to ensure that invariants of the data structure are maintained.

In this case: the `Size` argument of `Bin` should always reflect the actual size of the tree.

```
bin :: Map k a → k → a → Map k a → Map k a
bin l kx x r = Bin (size l + size r + 1) l kx x r
```

```
size :: Map k a → Int
size Tip                = 0
size (Bin sz _ _ _ _) = sz
```


Well-Typed

Finding an element

```
lookup :: Ord k ⇒ k → Map k a → Maybe a
lookup key Tip              = Nothing
lookup key (Bin _ l kx x r) =
  case compare key kx of
    LT → lookup key l
    GT → lookup key r
    EQ → Just x
```

Comparing two elements:

```
compare :: Ord a ⇒ a → a → Ordering
data Ordering = LT | EQ | GT
```


Well-Typed

Inserting an element

```
insert :: Ord k ⇒ k → a → Map k a → Map k a
insert kx x Tip               = singleton kx x    -- insert new
insert kx x (Bin sz l ky y r) =
  case compare kx ky of
    LT → balance (insert kx x l) ky y              r
    GT → balance              l  ky y (insert kx x r)
    EQ → Bin sz l kx x r    -- replace old
```

The function `balance` is an even smarter constructor with the same type as `bin`:

```
balance :: Map k a → k → a → Map k a → Map k a
```


Well-Typed

Balancing the tree

We could just define

```
balance = bin
```

and that would actually be correct.

Question

What is the problem, and when does it arise?

Balancing approach

- If the height of the two subtrees is not too different, we just use `Bin`.
- Otherwise, we perform a rotation.

Rotation

A rearrangement of the tree that preserves the search tree property.

Rotation

```
rotateL :: Map k a → k → a → Map k a → Map k a
rotateL l kx x r@(Bin _ ly _ _ ry)
  | size ly < ratio * size ry = singleL  l kx x r
  | otherwise                 = doubleL l kx x r
rotateL _ _ _ Tip = error "rotateL Tip"
```

Depending on the shape of the tree, either a simple (single) or a more complex (double) rotation is performed.

Well-Typed

Rotation – contd.

```
singleL :: Map k a → k → a → Map k a → Map k a
singleL t1 k1 x1 (Bin _ t2 k2 x2 t3) =
  bin (bin t1 k1 x1 t2) k2 x2 t3
```

```
doubleL :: Map k a → k → a → Map k a → Map k a
doubleL t1 k1 x1 (Bin _ (Bin _ t2 k2 x2 t3) k3 x3 t4) =
  bin (bin t1 k1 x1 t2) k2 x2 (bin t3 k3 x3 t4)
```

Note how easy it is to see that these rotations preserve the search tree property.

Well-Typed

Finger trees

- A balanced persistent tree structure.
- Supports search tree operations in logarithmic time.
- Supports cons and snoc in $O(1)$.
- Supports logarithmic splitting and union.

A universal data structure. A good choice for a wide range of applications.
Available in `Data.Sequence` from `containers` and in an extended version in the `fingertree` package on Hackage.

Byte strings and Text

Haskell strings are lists of characters.

Question

How much memory is needed to store a `String` that is three characters long?

There are other suitable datatypes for strings:

- Byte strings are stored as compact arrays (provided by `bytestring`. Mainly suitable for low-level or binary data.
- `Text` (provided by `text`) is a convenient datatype for text that wraps `bytestring` and deals with encoding issues.
- To prevent the typical array problems, a clever form of optimization called stream fusion is being used.

More on Hackage

On Hackage, there are several additional libraries for data structures.

Some examples: heaps, priority search queues, hash maps, heterogeneous lists, zippers, tries, graphs, quadtrees, . . .

Summary

- It is important to keep persistence in mind when thinking about functional data structures.
- Arrays should be used with care.
- Lists are ok for stack-like use or simple traversals.
- Good general-purpose data structures are sets, finite maps and sequences.

Testing

The Practice of Haskell Programming

Andres Löh

Well-Typed

June 17, 2012

What is testing about?

- Gain confidence in the correctness of your program.
- Show that common cases work correctly.
- Show that corner cases work correctly.
- **Testing cannot prove the absence of bugs.**
- Exception: Exhaustive testing.

Correctness

- When is a program correct?
- What is a specification?
- How to establish a relation between the specification and the implementation?
- What about bugs in the specification?

This lecture

QuickCheck, an automated testing library/tool for Haskell

Features:

- Describe properties as Haskell programs.
- Random test case generation.
- Test case generators can be adapted.

History

- Developed in 2000 by Koen Claessen and John Hughes.
- Copied to other programming languages: Common Lisp, Scheme, Erlang, Python, Ruby, SML, Clean, Java, Scala, F#
- Erlang version is sold by a company, QuviQ, founded by the authors of QuickCheck.

Well-Typed

A first version of the code

An attempt at insertion sort in Haskell:

```
sort :: [Int] → [Int]
sort []       = []
sort (x : xs) = insert x xs

insert :: Int → [Int] → [Int]
insert x []                   = [x]
insert x (y : ys) | x ⩽ y     = x : ys
                  | otherwise = y : insert x ys
```


Well-Typed

How to specify sorting?

A good specification is

- as precise as necessary,
- no more precise than necessary.

If we want to specify sorting, we should give a specification that distinguishes sorting from all other operations, but does not force us to use a particular sorting algorithm.

A first approximation

Certainly, sorting a list should not change its length.

```
sortPreservesLength :: [Int] → Bool
sortPreservesLength xs = length xs == length (sort xs)
```

We can test by invoking the function `quickCheck`:

```
⟩ quickCheck sortPreservesLength
*** Failed! Falsifiable (after 4 tests and 2 shrinks):
[0, 0]
```

Correcting the bug

```
sort :: [Int] → [Int]
sort []       = []
sort (x : xs) = insert x xs

insert :: Int → [Int] → [Int]
insert x []                    = [x]
insert x (y : ys) | x ⩽ y      = x : y : ys
                  | otherwise  = y : insert x ys
```


A new attempt

```
〉quickCheck sortPreservesLength
+++ OK, passed 100 tests.
```

Looks better. But have we tested enough?

Properties are first-class objects

```
(f `preserves` p) x = p x == p (f x)
sortPreservesLength = sort `preserves` length
idPreservesLength   = id `preserves` length
```

```
⟩ quickCheck idPreservesLength
+++ OK, passed 100 tests.
```

Clearly, the identity function does not sort the list.

When is a list sorted?

```
sorted :: [Int] → Bool
sorted []         = True
sorted (x : xs)   = ...
```

When is a list sorted?

$$
\begin{array}{ll}
\text{sorted} :: [\text{Int}] \to \text{Bool} & \\
\text{sorted}\ [\,] & = \text{True} \\
\text{sorted}\ (x : \boxed{[\,])} & = \ \ldots \\
\text{sorted}\ (x : \boxed{y : ys})\ & = \ \ldots
\end{array}
$$

Well–Typed

When is a list sorted?

$$
\begin{array}{ll}
\text{sorted} :: [\text{Int}] \to \text{Bool} & \\
\text{sorted}\ [\,] & = \text{True} \\
\text{sorted}\ (x : [\,]) & = \text{True} \\
\text{sorted}\ (x : y : ys) & = \ \ldots
\end{array}
$$

Well–Typed

When is a list sorted?

```
sorted :: [Int] → Bool
sorted []           = True
sorted (x : [])     = True
sorted (x : y : ys) = x < y && sorted (y : ys)
```


Well–Typed

Testing again

```
sortEnsuresSorted :: [Int] → Bool
sortEnsuresSorted xs = sorted (sort xs)
```

Or:

```
(f `ensures` p) x = p (f x)
sortEnsuresSorted = sort `ensures` sorted
```

```
〉quickCheck sortEnsuresSorted
*** Failed! Falsifiable (after 4 tests and 1 shrink):
[1, 1]
〉sort [1, 1]
[1, 1]
```

But this is correct. So what went wrong?

Well–Typed

Specifications can have bugs, too!

$\rangle$ sorted $[2, 2, 4]$
False

$$
\begin{aligned}
&\text{sorted} :: [\text{Int}] \to \text{Bool}\\
&\text{sorted}\ [\,] \qquad\qquad\ = \text{True}\\
&\text{sorted}\ (x : [\,]) \qquad = \text{True}\\
&\text{sorted}\ (x : y : ys) = x \boxed{\leqslant} y\ \&\&\ \text{sorted}\ (y : ys)
\end{aligned}
$$

Another attempt

$\rangle$ quickCheck sortEnsuresSorted
*** Failed! Falsifiable (after 5 tests and 4 shrinks):
$[0, 0, -1]$

There still seems to be a bug.

$\rangle$ sort $[0, 0, -1]$
$[0, 0, -1]$

Correcting again

```
sort :: [Int] → [Int]
sort []       = []
sort (x : xs) = insert x (sort xs)

insert :: Int → [Int] → [Int]
insert x []                 = [x]
insert x (y : ys) | x ⩽ y   = x : y : ys
                  | otherwise = y : insert x ys
```

```
〉 quickCheck sortEnsuresSorted
+++ OK, passed 100 tests.
```


Are we done yet?

Is sorting specified completely by saying that

- sorting preserves the length of the input list,
- the resulting list is sorted?

No, not quite

```
evilNoSort :: [Int] → [Int]
evilNoSort xs = replicate (length xs) 0
```

This function fulfills both specifications, but still does not sort. We need to make the relation between the input and output lists precise: both should contain the same elements – or one should be a permutation of the other.

Specifying sorting

```
f `permutes` xs    = f xs `elem` permutations xs
sortPermutes xs = sort `permutes` xs
```

Our sorting function fulfills this specification, but `evilNoSort` does not.

How to use QuickCheck

To use QuickCheck in your program:

```
import Test.QuickCheck
```

Define properties.

Then call `quickCheck` to test the properties.

```
quickCheck :: Testable prop ⇒ prop → IO ()
```


The type of `quickCheck`

The type of `quickCheck` is an **overloaded** type:

```
quickCheck :: Testable prop ⇒ prop → IO ()
```

- The argument of `quickCheck` is a property of type `prop`.
- The only restriction on the type `prop` is that it is in the `Testable` **type class**.
- When executed, `quickCheck` prints the results of the test to the screen – hence the `IO ()` result type.

Which properties are Testable ?

So far, all our properties have been of type $[\text{Int}] \to \text{Bool}$:

```
sortPreservesLength :: [Int] → Bool
sortEnsuresSorted   :: [Int] → Bool
sortPermutes        :: [Int] → Bool
```

When used on such properties, QuickCheck generates random integer lists and verifies that the result is True .

- If the result is True for 100 cases, this success is reported in a message.
- If the result is False for a case, the test case triggering the result is printed.

Other forms of properties

All these properties can be tested with quickCheck :

```
appendLength :: [a] → [a] → Bool
appendLength xs ys = length xs + length ys == length (xs ++ ys)
plusIsCommutative :: Int → Int → Bool
plusIsCommutative m n = m + n == n + m
takeDrop :: Int → [Int] → Bool
takeDrop n xs = take n xs ++ drop n xs == xs
dropTwice :: Int → Int → [Int] → Bool
dropTwice m n xs = drop m (drop n xs) == drop (m + n) xs
```

Other forms of properties – contd.

```
⟩ quickCheck takeDrop
+++ OK, passed 100 tests.
⟩ quickCheck dropTwice
*** Failed! Falsifiable (after 2 tests and 1 shrink):
1
−1
[0]
⟩ drop (−1) [0]
[0]
⟩ drop 1 (drop (−1) [0])
[]
⟩ drop (1 + (−1)) [0]
[0]
```


Well-Typed

Nullary properties

A property without arguments is also possible:

```
lengthEmpty :: Bool
lengthEmpty = length [] == 0
wrong :: Bool
wrong = False
⟩ quickCheck lengthEmpty
+++ OK, passed 100 tests.
⟩ quickCheck wrong
*** Failed! Falsifiable (after 1 test):
```

No random test cases are involved for nullary properties.
QuickCheck subsumes unit tests.

Well-Typed

Properties

Recall the type of `quickCheck`:

```
quickCheck :: Testable prop ⇒ prop → IO ()
```

We can now say more about when types are in `Testable`:

- testable properties usually are functions (with arbitrarily many arguments) resulting in a `Bool`

Are arbitrary argument types admissible?

No – QuickCheck has to know how to produce random test cases of such types.

Well-Typed

Properties – contd.

We can express the idea in Haskell using the type class language.

```
class Testable prop where
  property :: prop → Property
```

A `Bool` is testable:

```
instance Testable Bool where
  ...
```

If a type is testable, we can add another function argument, as long as we know how to generate and print test cases:

```
instance (Arbitrary a, Show a, Testable b) ⇒
         Testable (a → b) where
  ...
```


Well-Typed

Obtaining information about the test data

Question

Why is it important to know what data we actually test on?

A simple way is to use

`verboseCheck :: Testable prop ⇒ prop → IO ()`

rather than

`quickCheck    :: Testable prop ⇒ prop → IO ()`

Observations about random test data

- First test case are rather small.
- Test cases seem to increase in size over time.
- Duplicate test cases occur.

Often, `verboseCheck` is too much. We want to get information on the distribution of test cases according to a certain property.

The function collect

```
collect :: (Testable prop, Show a) ⇒ a → prop → Property
```

The function collect gathers statistics about test cases. This information is displayed when a test passes:

```
⟩ let sPL = sortPreservesLength
⟩ quickCheck (λxs → collect (null xs) (sPL xs))
+++ OK, passed 100 tests:
97% False .
 3% True .
```


The function collect – contd.

```
⟩ quickCheck (λxs → collect (length xs ‘div‘ 10) (sPL xs))
+++ OK, passed 100 tests:
29% 0
23% 1
14% 2
11% 3
 7% 4
 6% 5
 4% 9
 4% 6
 2% 7
```

The type Property

Recall the type of collect:

```
collect :: (Testable prop, Show a) ⇒ a → prop → Property
```

The type Property is QuickCheck-specific. It holds a more structural information about a property than a plain Bool ever could.

```
instance Testable Property where ...
```

Like Bool, a Property is testable, so for us, not much changes.

Well-Typed

Implications

The function insert preserves an ordered list:

```
implies :: Bool → Bool → Bool
implies x y = not x || y
```

A problematic property

```
insertPreservesOrdered :: Int → [Int] → Bool
insertPreservesOrdered x xs =
  sorted xs `implies` sorted (insert x xs)
```

Can you imagine why?

Well-Typed

Implications – contd.

```
⟩ quickCheck insertPreservesOrdered
+++ OK, passed 100 tests.
```

But:

```
⟩ let iPO = insertPreservesOrdered
⟩ quickCheck (λx xs → collect (sorted xs) (iPO x xs))
+++ OK, passed 100 tests:
88% False
12% True
```

For 88 test cases, `insert` has not actually been relevant for the result.

Well-Typed

Implications – contd.

The solution is to use the QuickCheck implication operator:

```
(==>) :: (Testable prop) => Bool → prop → Property
```

We see `Property` again – this type allows us to encode not only `True` or `False`, but also to reject the test case.

```
iPO :: Int → [Int] → Property
iPO x xs = sorted xs ==> sorted (insert x xs)
```

Now, lists that are not sorted are discarded and do not contribute towards the goal of 100 test cases.

Well-Typed

Implications – contd.

We can now easily run into a new problem:

```
〉quickCheck (λx xs → collect (sorted xs) (iPO x xs))
*** Gave up! Passed only 41 tests (100%  True ).
```

The chance that a random list is sorted is extremely small. QuickCheck will give up after a while if too few test cases pass the precondition.

Well-Typed

Generators

- Generators belong to an abstract data type `Gen` .
- We can define our own generators using another domain-specific language. The default generators for datatypes are specified by defining instances of class `Arbitrary` :

```
class Arbitrary a where
  arbitrary :: Gen a
  ...
```


Well-Typed

Building new generators

QuickCheck includes a library for the construction of new generators:

```
choose    :: Random a ⇒ (a, a) → Gen a
oneof     :: [Gen a] → Gen a
frequency :: [(Int, Gen a)] → Gen a
elements  :: [a] → Gen a
sized     :: (Int → Gen a) → Gen a
```

Quickly testing generators:

```
sample    :: Show a ⇒ Gen a → IO ()
```


Simple generators

```
instance Arbitrary Bool where
  arbitrary = choose (False, True)
instance (Arbitrary a, Arbitrary b) ⇒ Arbitrary (a, b) where
  arbitrary = pure (, ) <*> arbitrary <*> arbitrary
data Dir = North | East | South | West
instance Arbitrary Dir where
  arbitrary = elements [North, East, South, West]
```

Generating numbers

A simple possibility:

```
instance Arbitrary Int where
  arbitrary = choose (− 20, 20)
```

Better:

```
instance Arbitrary Int where
  arbitrary = sized (λn → choose (− n, n))
```

QuickCheck automatically increases the size gradually, up to the configured maximum value.

Well-Typed

How to generate sorted lists

Idea: Adapt the default generator for lists.

The following function turns a list of integers into a sorted list of integers:

```
mkSorted :: [Int] → [Int]
mkSorted []           = []
mkSorted [x]          = [x]
mkSorted (x : y : ys) = x : mkSorted (x + abs y : ys)
```

Example

```
⟩ mkSorted [1, 3, − 4, 0, 2]
[1, 4, 8, 8, 10]
```


Well-Typed

How to generate sorted lists – contd.

The original generator can be adapted as follows:

```
genSorted :: Gen [Int]
genSorted = pure mkSorted <*> arbitrary
```


Well-Typed

Using a custom generator

There is another function to construct properties provided by QuickCheck:

```
forAll :: (Show a, Testable b) ⇒ Gen a → (a → b) → Property
```

This is how we use it:

```
iPO :: Int → Property
iPO x = forAll genSorted
  (λxs → sorted xs ⟹ sorted (insert x xs))
```

And it works:

```
〉 quickCheck iPO
+++ OK, passed 100 tests.
```


Well-Typed

Summary

QuickCheck is a great tool:

- A domain-specific language for writing properties.
- Test data is generated automatically and randomly.
- Another domain-specific language to write custom generators.
- You should use it.

However, keep in mind that writing good tests still requires training, and that tests can have bugs, too.

Reachable uncovered code

Program code can be classified:

- **unreachable code**: code that simply is not used by the program, usually library code
- **reachable code**: code that can in principle be executed by the program

Reachable code can be classified further:

- **covered code**: code that is actually executed during a number of program executions (for instance, tests)
- **uncovered code**: code that is not executed during testing

Uncovered code is untested code – it could be executed, and it could do anything!

Introducing HPC

- HPC (Haskell Program Coverage) is a tool – integrated into GHC – that can identify uncovered code.
- Using HPC is extremely simple:
  - Compile your program with the flag `-fhpc`.
  - Run your program, possibly multiple times.
  - Run `hpc report` for a short coverage summary.
  - Run `hpc markup` to generate an annotated HTML version of your source code.

Well-Typed

What HPC does

- HPC can present your program source code in a color-coded fashion.
- Yellow code is uncovered code.
- Uncovered code is discovered down to the level of subexpressions! (Most tools for imperative language only give you line-based coverage analyis.)
- HPC also analyzes boolean expressions:
  - Boolean expressions that have always been True are displayed in green.
  - Boolean expressions that have always been False are displayed in red.

Well-Typed

QuickCheck and HPC

QuickCheck and HPC interact well!

- Use HPC to discover code that is not covered by your tests.
- Define new test properties such that more code is covered.
- Reaching 100% can be really difficult (why?), but strive for as much coverage as you can get.

More on testing

- QuickCheck can subsume unit tests, but QuickCheck is less suitable for testing `IO`-based code.
- There is a more classic unit test library for Haskell called `HUnit`.
- For small domains, exhaustive testing becomes a real option. The libraries `smallcheck` and `lazysmallcheck` migrate the ideas of QuickCheck to systematic exhaustive testing.
- Test suites can be integrated into Cabal packages.
- Tests can be integrated with Haddock documentation using `doctest`.

Evaluation

The Practice of Haskell Programming

Andres Löh

Well-Typed

June 17, 2012

Reduction

A subexpression that can be reduced is called a **redex**.

Most typical form of reduction in Haskell: replacing the left hand side of a function definition by a corresponding right hand side (this is essentially **beta reduction** from **lambda calculus**).

Question

What if there are multiple redexes in one term?

Multiple redexes

Many terms have multiple redexes.
How many redexes are in the following term?

$\mathsf{id}\ (\mathsf{id}\ (\lambda z \to \mathsf{id}\ z))$

$(\lambda x \to \lambda y \to x * x)\ (1 + 2)\ (3 + 4)$

Example

Let us play through the possible reductions for the following terms:

```
head (repeat 1)
```

```
let minimum xs = head (sort xs)
in minimum [4, 1, 3]
```

Haskell's lazy evaluation

In Haskell,

- expressions are only evaluated if actually required,
- the leftmost outermost redex is chosen to achieve this,
- sharing is introduced in order to prevent evaluating expressions multiple times.

If no redexes are left, an expression is in **normal form**. If the top-level of an expression is a constructor or lambda, then the expression is in **(weak) head normal form**.

Well–Typed

Common evaluation strategies

Call by value / strict evaluation

Most common. Arguments are reduced as far as possible before reducing a function application, usually left-to-right.

Call by name

Functions are reduced before their arguments. Used by some macro languages (TeX, for instance).

Call by need / lazy evaluation

Optimized version of "Call by name": function arguments are only reduced when needed, but shared if used multiple times.

λf g x $\rightarrow$ combine (f x) (g x)

Well–Typed

Church-Rosser

Theorem (Church-Rosser)

If a term e can be reduced to e_1 and e_2, there is a term e_3 such that both e_1 and e_2 can be reduced to e_3.

Corollary

Each term has at most one normal form.

Theorem

If a term has a normal form, then lazy evaluation arrives at this normal form.

Well-Typed

Non-termination

In Haskell, we can easily define non-terminating terms:

```
x :: a
x = x
```

Abnormal termination by means of a runtime exception is strongly related to non-termination:

```
undefined :: a
error     :: String → a
```

You can see a run-time exception as an “optimization” of a diverging computation.

Well-Typed

Strict functions vs. strict evaluation

A function `f` is called **strict** if `f undefined` does not terminate normally.

Note

In a **strict** language, all functions are strict.

In a **non-strict** language, such as Haskell, we have both strict and non-strict functions.

Well-Typed

Examples

The function `const` is strict in its first, but not in its second argument.

The function $(+)$ is strict in both its arguments.

The function `map` is not strict in its first argument, but strict in its second.

However, `map` shows that we often need more fine-grained information about evaluation.

Well-Typed

Lazy evaluation quiz

$(\lambda x \rightarrow x)$ True	$\rightsquigarrow^*$
$(\lambda x \rightarrow x)$ undefined	$\rightsquigarrow^*$
$(\lambda x \rightarrow ())$ undefined	$\rightsquigarrow^*$
$(\lambda x \rightarrow$ undefined$)$ ()	$\rightsquigarrow^*$
$(\lambda x\ f \rightarrow f\ x)$ undefined	$\rightsquigarrow^*$
(error "1") (error "2")	$\rightsquigarrow^*$
length (map undefined [1, 2])	$\rightsquigarrow^*$

Well-Typed

Example: the first 100 odd square numbers

```
example :: [Int]
example =                                         [1 ..]
```

We start by generating all numbers (lazy evaluation in action).

Well-Typed

Example: the first 100 odd square numbers

```
example :: [Int]
example =                          map (λx → x ∗ x)  [1 ..]
```

We use `map` to compute the square numbers.

Well-Typed

Example: the first 100 odd square numbers

```
example :: [Int]
example = (           filter odd ∘ map (λx → x ∗ x)) [1 ..]
```

We use function composition composition (and partial application) to subsequently filter the odd square numbers.

Well-Typed

Example: the first 100 odd square numbers

```
example :: [Int]
example = (take 100 ∘ filter odd ∘ map (λx → x ∗ x)) [1 ..]
```

Finally, we use composition again to take the first 100 elements of this list.

Well-Typed

What drives the evaluation?

If we type an expression in at the GHCi prompt:

- GHCi wants to print its result,
- and for printing, we need that expression in normal form,
- that then demands other expressions to be evaluated.

Similarly for a complete program.

Within a function, it is most often **pattern matching** that drives the evaluation:

- in order to produce part of the output, we have to select a case;
- in order to be able to choose a case, we have to evaluate some of the arguments just far enough.

Evaluating a term to **weak head normal form** (WHNF) reveals its outermost constructor and allows us to potentially make a choice in a pattern match.

Well-Typed

Haskell data in memory

As we've sketched in the data structures lecture:

- nearly all Haskell data lives on the heap,
- nearly all Haskell data is immutable,
- operations do not change data but rather create new data on the heap,
- a lot of data is shared.

Sharing is easy because everything is immutable.

Laziness on the heap

Bindings are not evaluated immediately:

- Instead, suspended computations (called **thunks**) are created on the heap.
- Thunks can be shared just as other subterms.
- If a thunk is required, it is evaluated and destructively updated on the heap.
- However, this is a safe and even desirable update – we don't change the value stored, we just change its representation.
- Other computations sharing the updated thunk won't have to recompute the expression.

Garbage collection

GHC uses a generational garbage collector:

- Optimized for lots of short-lived data, as is common in a purely functional language.
- New data is allocated in the "young" generation.
- The young generation is rather small and collected often.
- After a while, data that is still alive is moved to the "old" generation.
- The old generation is larger and collected rarely.
- The heap of a Haskell program can grow dynamically if more memory is needed.

The lifetime of data

Data is alive as long as there are references to it.

In a lazy setting, it is sometimes hard to predict how long we retain references to data.

Space leak

A data structure which grows bigger, or lives longer, than we expect.

As space is a limited resource, we might run (nearly) out of it. Consequences:

- more garbage collections cost extra time,
- swapping,
- program might get killed.

Computing a large sum

$$\begin{aligned} \mathsf{sum}_1\ [\,] &= 0 \\ \mathsf{sum}_1\ (x : xs) &= x + \mathsf{sum}_1\ xs \end{aligned}$$

- A straight-forward definition, following the standard pattern of defining functions on lists.
- What is the problem?
- If we try to evaluate this function for larger and larger input lists, we note that it takes more and more memory, and significant amounts of time, or we get an error indicating it runs out of stack space.
- But certainly we should be able to sum a list in (nearly) constant (stack) space? What is going on?

Obtaining more information

Haskell's run-time system (RTS) can be instructed to spit out additional information:

- RTS options can be passed to Haskell binaries on the command line by placing them after `+RTS` or enclosing them between `+RTS` and `-RTS`.
- Many RTS flags require the binary to be compiled (or rather linked) using the `-rtsopts` GHC flag.
- You can obtain info about available RTS flags by invoking a compiled binary with `+RTS --help`.
- Very interesting are GC statistics (available in various amounts of detail via `-t`, `-s` or `-S`).
- You can increase the stack space by saying something like `-K50M` or `-K500M`.

GC statistics

```
$ ./Sum1 10000000 +RTS -s -K500M
50000005000000
   1,532,401,936 bytes allocated in the heap
     788,992,048 bytes copied during GC
     457,301,152 bytes maximum residency (10 sample(s))
         740,216 bytes maximum slop
             633 MB total memory in use (0 MB lost due to fragmentation)

                                    Tot time (elapsed)  Avg pause  Max pause
  Gen  0      2299 colls,     0 par    0.83s    0.83s     0.0004s    0.0008s
  Gen  1        10 colls,     0 par    0.60s    0.60s     0.0602s    0.2877s

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    0.46s  (  0.46s elapsed)
  GC      time    1.43s  (  1.43s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    1.89s  (  1.88s elapsed)

  %GC     time      75.8%  (75.8% elapsed)

  Alloc rate    3,352,283,510 bytes per MUT second

  Productivity  24.2% of total user, 24.3% of total elapsed
```

MUT (mutator) time is good, GC time is bad.

Maximum residency and percentage of GC time are revealing.

Heap profiling

More detailed information can be obtained using heap profiling.

- Requires recompilation of the program (makes program larger and overall slower).
- All used libraries must have profiling versions, too.
- In your cabal-install `config` file, put

```
-- library-profiling: True
```

  for the future.
- Compile a program with profiling enabled:

```
$ ghc --make -prof -auto-all -rtsopts Sum1
```

  The `-auto-all` is optional. It is more important for larger programs where you not only want to know **how much** space is being used, but also **where** it is being used.

Heap profiling – contd.

- Run with profiling enabled:

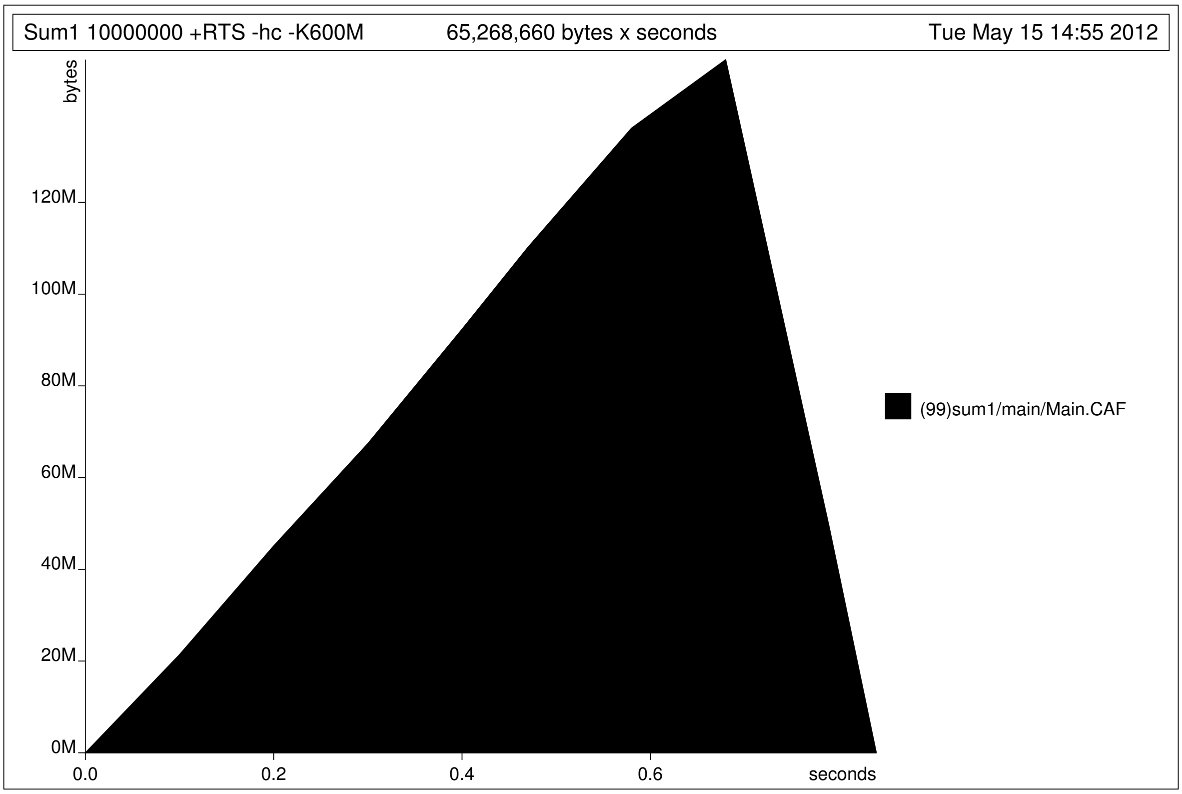
```
$ ./Sum1 10000000 +RTS -K800M -hc
```

  Again, there are many different -h flags.
- The -hc is for cost-center profiling.
- A very simplistic form of heap profiling via just -h is available even without compiling the program for profiling. It would be sufficient here!
- Files `Sum1.prof` and `Sum1.hp` are produced.
- The .hp file can be transformed into PostScript format using the `hp2ps` tool.

```
$ hp2ps Sum1.hp
```


Well-Typed

Heap profile for $\texttt{sum}_1$



Well-Typed

The problem

$$\begin{aligned}
& \mathrm{sum}_1\ [1,2,3,4,\ldots] \\
\equiv\ & \quad \{\ \text{Definition of}\ \ \mathrm{sum}_1\ \ \} \\
& 1 + \mathrm{sum}_1\ [2,3,4,\ldots] \\
\equiv\ & \quad \{\ \text{Definition of}\ \ \mathrm{sum}_1\ \ \} \\
& 1 + (2 + \mathrm{sum}_1\ [3,4,\ldots]) \\
\equiv\ & \quad \{\ \text{Definition of}\ \ \mathrm{sum}_1\ \ \} \\
& 1 + (2 + (3 + \mathrm{sum}_1\ [4,\ldots])) \\
\equiv\ & \\
& \ldots
\end{aligned}$$

The whole recursion has to be unfolded before the first addition can be reduced!

Well-Typed

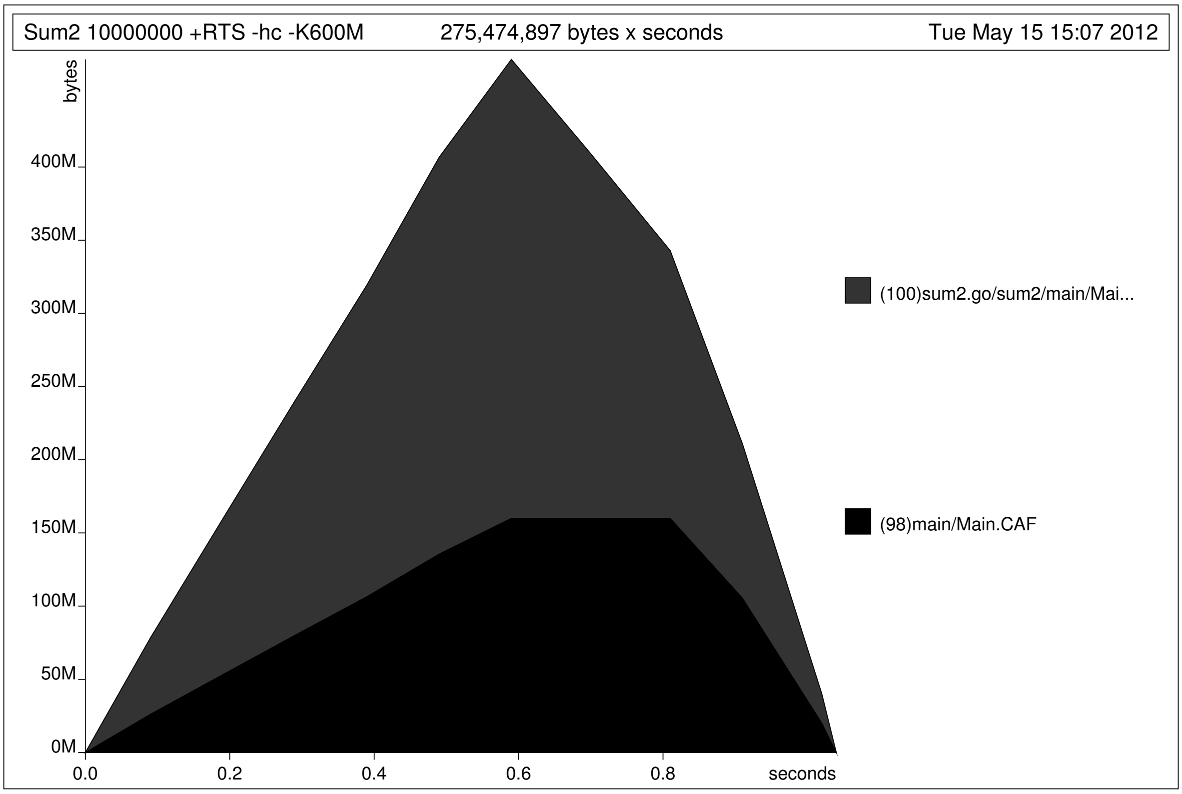
Attempting a tail-recursive version

```
sum2 xs = go 0 xs
  where
    go acc []       = acc
    go acc (x : xs) = go (acc + x) xs
```

We hope that tail-recursion improves stack usage, and might thereby improve space behaviour as well, but . . .

Well-Typed

Heap profile for sum_2



Well-Typed

The new problem

$$
\begin{aligned}
& \mathsf{sum}_2\ [1,2,3,4,\ldots] \\
\equiv\ & \quad \{\ \text{Definition of } \mathsf{sum}_2\ \} \\
& \mathsf{sum}_2'\ 0\ [1,2,3,4,\ldots] \\
\equiv\ & \quad \{\ \text{Definition of } \mathsf{sum}_2\ \} \\
& \mathsf{sum}_2'\ (0+1)\ [2,3,4,\ldots] \\
\equiv\ & \quad \{\ \text{Definition of } \mathsf{sum}_2\ \} \\
& \mathsf{sum}_2'\ ((0+1)+2)\ [3,4,\ldots] \\
& \ldots \\
\equiv\ & \\
& \mathsf{sum}_2'\ (\ldots((0+1)+2)\ldots)\ [\,] \\
\equiv\ & \quad \{\ \text{Definition of } \mathsf{sum}_2\ \} \\
& (\ldots(0+1)+2)\ldots)
\end{aligned}
$$

We still build up the whole addition, but now in an accumulating argument! Evaluating that still takes stack!

Well-Typed

We need more control

Sometimes, we want to make things stricter than they are by default. Here:

- we have a computation that will be evaluated anyway,
- storing it in delayed form costs much more space than storing its result.

Forcing evaluation

Haskell has the following primitive function

```
seq :: a → b → b    -- primitive
```

The call `seq x y` is strict in `x` and returns `y`.

The function `seq` can be used to define strict function application:

```
($!) :: (a → b) → a → b
f $! x = x `seq` f x
```

Recall sharing!

Forcing quiz

The function `seq` only evaluates to WHNF (i.e., a lambda abstraction, literal or constructor application).

$$
\begin{array}{ll}
(\lambda x \to ())\ \$!\ \mathsf{undefined} & \rightsquigarrow^* \\
\mathsf{seq}\ (\mathsf{error}\ "1", \mathsf{error}\ "2")\ () & \rightsquigarrow^* \\
\mathsf{snd}\ \$!\ (\mathsf{error}\ "1", \mathsf{error}\ "2") & \rightsquigarrow^* \\
(\lambda x \to ())\ \$!\ (\lambda x \to \mathsf{undefined}) & \rightsquigarrow^* \\
\mathsf{error}\ "1"\ \$!\ \mathsf{error}\ "2" & \rightsquigarrow^* \\
\mathsf{length}\ \$!\ \mathsf{map}\ \mathsf{undefined}\ [1, 2] & \rightsquigarrow^* \\
\mathsf{seq}\ (\mathsf{error}\ "1" + \mathsf{error}\ "2")\ () & \rightsquigarrow^* \\
\mathsf{seq}\ (1 : \mathsf{undefined})\ () & \rightsquigarrow^*
\end{array}
$$

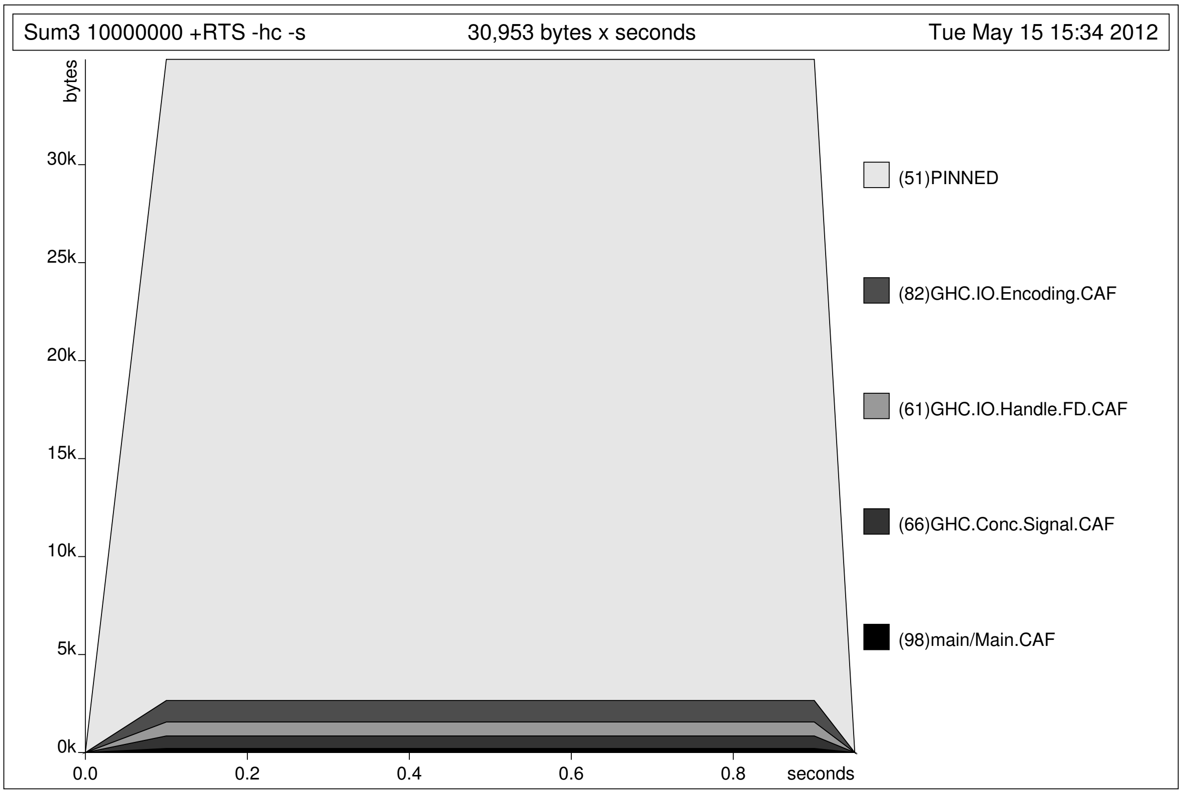
Well-Typed

Using `seq` to force the addition

```
sum₃ xs = go 0 xs
  where
    go acc []       = acc
    go acc (x : xs) = (go $! acc + x) xs
```


Well-Typed

Heap profile for `sum3`



GC statistics

```
$ ./Sum3 10000000 +RTS -hc -s
50000005000000
   2,560,118,208 bytes allocated in the heap
         714,144 bytes copied during GC
          62,104 bytes maximum residency (10 sample(s))
          26,344 bytes maximum slop
               1 MB total memory in use (0 MB lost due to fragmentation)

                                    Tot time (elapsed)  Avg pause  Max pause
  Gen  0      4873 colls,     0 par    0.02s    0.02s     0.0000s    0.0000s
  Gen  1        10 colls,     0 par    0.00s    0.00s     0.0001s    0.0001s

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    0.95s  (  0.95s elapsed)
  GC      time    0.02s  (  0.02s elapsed)
  RP      time    0.00s  (  0.00s elapsed)
  PROF    time    0.00s  (  0.00s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    0.98s  (  0.98s elapsed)

  %GC     time       2.3%  (2.2% elapsed)

  Alloc rate    2,684,947,785 bytes per MUT second

  Productivity  97.6% of total user, 97.6% of total elapsed
```

Look at the maximum residency and GC time / productivity now.

Standard recursion patterns

The three versions of `sum` we have seen correspond to using `foldr`, `foldl` and `foldl'`, respectively:

$$\begin{aligned}
\mathrm{sum}_1 &= \mathrm{foldr}\ \ (+)\ 0\\
\mathrm{sum}_2 &= \mathrm{foldl}\ \ (+)\ 0\\
\mathrm{sum}_3 &= \mathrm{foldl}'\ (+)\ 0
\end{aligned}$$

Well-Typed

Question

Is using `foldl'`/strictness always preferable?

For example, what about defining `map` ...

Well-Typed

Rules of thumb

- If you expect partial results or want to use infinite lists, use `foldr`.

  Examples: `map`, `filter`.

- If the operator is strict, use `foldl'`.

  Examples: `sum`, `product`.

- Otherwise, use `foldl`.

  Examples: `reverse`.

- Use the GHC optimizer by passing `-O`. GHC performs strictness analysis to optimize your code – but don't rely on it to always figure out everything!

Well-Typed

(Embedded) Domain-Specific Languages

The Practice of Haskell Programming

Andres Löh

Well-Typed

June 20, 2012

What is an (E)DSL?

- DSL = domain-specific language (fuzzy concept)
- EDSL = **embedded** DSL

In essence, EDSLs are just Haskell libraries:

- a limited set of types and functions;
- certain rules for composing sensible expressions out of these building blocks;
- often a certain unique look and feel;
- often understandable without having to know (all about) the host language.

DSLs vs. EDSLs

DSLs

- complete design freedom,
- limited syntax, thus easy to understand, usable by non-programmers,
- requires dedicated compiler, development tools,
- hard to extend with general-purpose features.

EDSLs

- design tied to capabilities of host language,
- compiler and general-purpose features for free,
- complexity of host language available but exposed,
- several EDSLs can be combined and used together.

Haskell (or rather: Hackage) is full of EDSLs!

database queries
pretty-printing
workflows
parallelism
testing
web applications
(de)serialization
parsing
JavaScript
animations
hardware descriptions
data accessors / lenses
music
(attribute) grammars
HTML
concurrency
GUIs
array computations
images

Why?

Several reasons:

- syntactic freedom (user-defined operators and priorities, overloading, overloaded literals, `do` -notation, . . .),
- higher-order functions,
- lazy evaluation,
- strong type system, type inference,
- algebraic datatypes,
- explicit effects,
- good partial evaluator,
- user-defined optimizations (rewrite rules).

EDSLs already encountered

QuickCheck:

- the language to construct **properties**,
- the language to construct **generators**.

In Ralf's lecture:

- an EDSL for describing **music**.

In Simon's lecture: EDSLs for

- computations with **software transactional memory**;
- describing **parallel computations**.

Parsing

Let us look at a classic EDSL example: **parsing**.

Goal:

- a library for describing parsers,
- no generation approach,
- high degree of abstraction,
- easy to use.

Parser interface

Type of parsers producing a result of type `a`:

```
data Parser a   -- abstract
```

Succeed always, consume nothing:

```
pure :: a → Parser a
```

Consume a single matching character:

```
satisfy :: (Char → Bool) → Parser Char
```

Parser interface – contd.

Change the type of the result:

$(<\$>) :: (a \to b) \to \text{Parser } a \to \text{Parser } b$

Parse one thing followed by another:

$(<*>) :: \text{Parser } (a \to b) \to \text{Parser } a \to \text{Parser } b$

Parse one thing or another:

$(<|>) :: \text{Parser } a \to \text{Parser } a \to \text{Parser } a$

Well–Typed

The type of $(<*>)$

Why not:

$(<\times>) :: \text{Parser } a \to \text{Parser } b \to \text{Parser } (a, b)$

Consider:

```
data X = X A B C
pA :: Parser A
pB :: Parser B
pC :: Parser C
```

$(\lambda((a, b), c) \to X\ a\ b\ c) <\$> (pA <\times> pB <\times> pC) :: \text{Parser } X$

$\text{pure } X <*> pA <*> pB <*> pC :: \text{Parser } X$

Well–Typed

An fmap function on parsers

The `(<$>)` operator has the same type as `fmap`:

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

```
instance Functor Parser where
  fmap f p = pure f <*> p
```

```
    X <$> pA <*> pB <*> pC :: Parser X
```


Derived parser combinators

Parsing many occurrences.

BNF:

```
⟨ps⟩ ::= ⟨p⟩ ⟨ps⟩
      |  ε
```

Haskell:

```
many :: Parser a -> Parser [a]
many p  =  (:) <$> p <*> many p
           <|> pure []
```

```
ident :: Parser String
ident = (:) <$> satisfy isAlpha <*> many (satisfy isAlphaNum)
```

Parsing balanced parentheses

BNF:

$$
\begin{aligned}
\langle \text{bal} \rangle &::= (\ \langle \text{bal} \rangle\ )\ \langle \text{bal} \rangle \\
&\quad\ \mid\ \varepsilon
\end{aligned}
$$

Haskell:

```haskell
type Total = Int

sym :: Char -> Parser Char
sym x = satisfy (== x)

bal :: Parser Total
bal   =   (\_ m _ n -> (1 + m) + n)
    <$> sym '(' <*> bal <*> sym ')' <*> bal
    <|> pure 0
```


Well-Typed

Parsing balanced parentheses

BNF:

$$
\begin{aligned}
\langle \text{bal} \rangle &::= (\ \langle \text{bal} \rangle\ )\ \langle \text{bal} \rangle \\
&\quad\ \mid\ \varepsilon
\end{aligned}
$$

Haskell:

```haskell
data Bal = Nest [Bal]

sym :: Char -> Parser Char
sym x = satisfy (== x)

bal :: Parser [Bal]
bal   =   (\_ x _ xs -> Nest x : xs)
    <$> sym '(' <*> bal <*> sym ')' <*> bal
    <|> pure []
```


Well-Typed

Parsing balanced parentheses

BNF:

$$\langle \text{bal} \rangle ::= (\ \langle \text{bal} \rangle\ )\ \langle \text{bal} \rangle \\ \qquad\qquad \mid\ \varepsilon$$

Haskell:

```
data Bal = Nest Bal Bal | Nil

sym :: Char → Parser Char
sym x = satisfy (== x)

bal :: Parser Bal
bal  =  (λ_ x _ xs → Nest x xs)
     <$> sym '(' <*> bal <*> sym ')' <*> bal
     <|> pure Nil
```


List-of-successes semantics

A simple semantics of parsers:

- parsers transform an input string,
- they consume some (but not necessarily all) input,
- they also produce a result –
- well, actually they can fail (no result) or be ambiguous (several results), too.

Let us try to just use this as an implementation:

```
type Parser a = String → [(a, String)]
```

Using functions as semantics is quite common.

Implementing simple parser combinators

$$\begin{aligned}&\mathsf{pure} :: a \rightarrow \mathsf{String} \rightarrow [(a, \mathsf{String})]\\&\mathsf{pure}\ x\ ys = [(x, ys)]\end{aligned}$$

$$\begin{aligned}&\mathsf{satisfy} :: (\mathsf{Char} \rightarrow \mathsf{Bool}) \rightarrow \mathsf{Parser\ Char}\\&\mathsf{satisfy}\ p\ (y : ys) \mid p\ y = [(y, ys)]\\&\mathsf{satisfy}\ p\ \_ \qquad\qquad\;\; = [\,]\end{aligned}$$

$$\begin{aligned}&(<\$>) :: (a \rightarrow b) \rightarrow \mathsf{Parser}\ a \rightarrow \mathsf{Parser}\ b\\&(f <\$> p)\ xs = [(f\ r, ys) \mid (r, ys) \leftarrow p\ xs]\end{aligned}$$

$$\begin{aligned}&(<*>) :: \mathsf{Parser}\ (a \rightarrow b) \rightarrow \mathsf{Parser}\ a \rightarrow \mathsf{Parser}\ b\\&(p <*> q)\ xs = [(f\ r, zs) \mid (f, ys) \leftarrow p\ xs,\\&\qquad\qquad\qquad\qquad\qquad\quad (r, zs) \leftarrow q\ ys]\end{aligned}$$

$$\begin{aligned}&(<|>) :: \mathsf{Parser}\ a \rightarrow \mathsf{Parser}\ a \rightarrow \mathsf{Parser}\ a\\&(p <|> q)\ xs = p\ xs +\!\!+\ q\ xs\end{aligned}$$

Demo

(Demo.)

Disadvantages of our parsers

- We always compute all alternatives. Potentially lots of backtracking, inefficient.
- Absolutely no error messages.
- Tied to `String` as input type.

But these problems can be fixed. For example:

- `parsec`, limited lookahead, good error messages;
- `Text.ParserCombinators.ReadP`, trying several choices "in parallel",
- `uu-parsinglib`, more sophisticated variant of `ReadP`, error messages, incremental parsing.

Abstracting from interfaces

Many EDSLs are focused on one (or several) **parameterized** type(s):

```
Parser a
Gen    a
STM    a
Par    a
```

- All these types can be seen as enriched, effectful versions of the underlying type `a`.
- We always need a way to relate plain types and their effectful versions.
- We always need a way to combine effectful terms.

Monads

```
class Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
  fail   :: String → m a    -- controversial
```

Monads are common: they allow

- embedding via `return`,
- sequencing via `(>>=)`,
- later parts of the computation can depend on earlier results.

All of `Gen`, `STM` and `Par` are monads.

Advantages of a common interface

Next to the familiarity, we can define various functions in terms of just the interface and then reuse them on all types that implement the interface:

```
sequence :: Monad m ⇒ [m a] → m [a]
sequence []       = return []
sequence (m : ms) =
  m >>= λx → sequence ms >>= λxs → return (x : xs)
```

```
foldM :: Monad m ⇒ (a → b → m a) → a → [b] → m a
foldM op e []       = return e
foldM op e (x : xs) = foldM op e xs >>= λr → op r x
```

For the monad interface, we additionally get to use `do` notation.

Functors

Even more fundamental than the monad interface is the functor interface:

```
class Functor f where
  fmap :: (a → b) → f a → f b
(<$>) = fmap
```

Every monad is a functor:

```
liftM :: Monad m ⇒ (a → b) → m a → m b
liftM f m = m >>= λx → return (f x)
```

However, this isn't automatically exploited in Haskell's class system.

Types that are just functors are not very suitable for EDSLs, because there is no way to combine enriched terms . . .

Effectful application

But if we look at our parser interface, we find a different way of combining results:

```
(<*>) :: Parser (a → b) → Parser a → Parser b
```

- Another form of sequencing.
- While the results are combined, the second computation cannot depend on the result of the first.
- So `(<*>)` is more restrictive than `(>>=)`.
- The interface with embedding (`pure`, like `return`) and `(<*>)` is called `Applicative`.

Applicative functors

$$\textbf{class}\ \text{Functor}\ f \Rightarrow \text{Applicative}\ f\ \textbf{where}$$
$$\quad \text{pure} \quad :: a \to f\ a$$
$$\quad (<\!*\!>) :: f\ (a \to b) \to f\ a \to f\ b$$

The $(<\!|\!>)$ combinator is also more generally useful:

$$\textbf{class}\ \text{Applicative}\ f \Rightarrow \text{Alternative}\ f\ \textbf{where}$$
$$\quad \text{empty} :: f\ a$$
$$\quad (<\!|\!>) :: f\ a \to f\ a \to f\ a$$

Question: How do we define empty for our parsers?

Well-Typed

Common applicative functions

Again, there are functions that require just the Applicative (and Alternative) interfaces.

Recall for example many :

$$\text{many} :: \text{Alternative}\ f \Rightarrow f\ a \to f\ [a]$$
$$\text{many}\ p \;=\; (:)\ <\!\$\!>\ p\ <\!*\!>\ \text{many}\ p$$
$$\qquad\qquad <\!|\!>\ \text{pure}\ [\,]$$

Well-Typed

Monads are applicative

Many computations do not need the full power of `(>>=)`.

```
ap :: Monad m ⇒ m (a → b) → m a → m b
ap mf mx = mf >>= λf → mx >>= λx → return (f x)
```

Once again, an `Applicative` instance isn't automatically defined for every monad – but most common monads have `Functor` and `Applicative` instances.

Stylistic comparison

All three are equivalent:

```
comp = do
  x ← f1
  y ← f2
  return (something x y)
```

```
comp = liftM2 something f1 f2
```

```
comp = something <$> f1 <*> f2
```

Degree of embedding

Shallow embedding
EDSL constructs are directly represented by their semantics.

Deep embedding
EDSL constructs are represented by their abstract syntax, and interpreted in a separate stage.

Note:

- These are two extreme points in a spectrum.
- Most EDSLs use something in between (but close to one end).

Examples

Classic example of a shallow embedding:

```
type Parser a = String → [(a, String)]
```

The Haskore music language is using a (relatively) deep embedding:

```
data Music = Note Pitch Dur [NoteAttribute]
           | Rest Dur
           | Music :+: Music
           | Music :=: Music
           | Tempo (Ratio Int) Music
           | Trans Int Music
           | Instr IName Music
           | Player PName Music
           | Phrase [PhraseAttribute] Music
```

Shallow vs. deep

Shallow

- Working directly with the (denotational) semantics is often very concise and elegant.
- Relatively easy to use all Haskell features (sharing, recursion).
- Difficult to debug and/or analyze, because we are limited to a single interpretation.

Deep

- Full control over the AST, many different interpretations possible.
- Allows on-the-fly runtime optimization and conversion.
- We can visualize and debug the AST.
- Hard(er) to use Haskell's sharing and recursion.

An example

Let us embed an almost trivially simple language of arithmetic expressions:

```
data Expr    -- abstract
(⊕) :: Expr → Expr → Expr
one :: Expr
eval :: Expr → Int
```

An example
Shallow implementation

```
type Expr = Int
(⊕)  = (+)
one  = 1
eval = id
```

- We pick a semantics of expressions: an `Int`.
- We directly implement language constructs by their semantics.
- Very easy to do, but limited to a single interpretation.

Well-Typed

An example
Deep implementation

```
(⊕) :: Expr → Expr → Expr
one :: Expr
eval :: Expr → Int
```

```
data Expr = Pl Expr Expr | One
(⊕) = Pl
one = One
eval (Pl e1 e2) = eval e1 + eval e2
eval One        = 1
```

We are no longer tied to one interpretation ...

Well-Typed

Showing expressions

```
disp :: Expr → String
disp (Pl e₁ e₂) = "(" ++ disp e₁ ++ " + " ++ disp e₂ ++ ")"
disp One        = "1"
```

Similarly, we could:

- transform the expression,
- optimize the expression,
- generate some code for the expression in another language,
- ...

Turning a concept into data

Moving from shallow towards deep is an important functional design pattern:

- introducing data types is easy,
- former functions become constructors,
- as a result, the structure of terms becomes observable.

A user-defined abstraction

```
tree :: Int → Expr
tree 0 = one
tree n = let shared = tree (n − 1) in shared ⊕ shared
```

With the shallow embedding, this is fine:

- We reuse Haskell's sharing.
- What we share is just an integer.

But now in the deep setting . . .

```
tree :: Int → Expr
tree 0 = one
tree n = let shared = tree (n − 1) in shared ⊕ shared
```

The call `disp (tree 3)` results in

```
"(((1 + 1) + (1 + 1)) + ((1 + 1) + (1 + 1)))"
```

Sharing is destroyed! We don't want to wait for `eval (tree 30)` !

Parsing our expression language

Our parser combinators work, but they cannot handle left-recursive grammars:

```
⟨expr⟩ ::= 1
         | 1 + ⟨expr⟩   -- ok
```


Parsing our expression language

Our parser combinators work, but they cannot handle left-recursive grammars:

```
⟨expr⟩ ::= 1
         | ⟨expr⟩ + 1   -- not ok
```

Resulting parser:

```
expr :: Parser Expr
expr  =   const One                <$> sym '1'
      <|> (λ_ _ e → Pl One e₂) <$> expr <*> sym '+' <*> sym '1'
```

- This parser will loop.
- In the second alternative, `expr` is called again before any input has been consumed.

Left recursion and parser combinators

For parsers, not being able to handle left recursion is not actually that serious a problem:

- left recursion can relatively easily be removed,
- common cases of left recursion can be abstracted into specific parser combinators (`chainl`).

Nevertheless, for some EDSL applications we would like to **preserve** or **observe** recursion and sharing.

Well-Typed

Making sharing (and recursion) explicit

In practice, many EDSLs require preserving and observing sharing and recursion.

We need:

- a way to explicitly represent sharing in our representation,
- a way to conveniently produce terms in that representation.

Unfortunately, we have time left for only a short look at the options.

Well-Typed

Recall `vacuum`

The `vacuum` package:

- queried GHC's internal representation of data,
- in order to produce visualizations of terms that reveal the sharing.

Perhaps we can use a similar hack to recover implicit sharing in EDSL terms?

Well-Typed

Introducing `data-reify`

The `data-reify` package offers such a function to recover implicit sharing:

```
reifyGraph :: MuRef s ⇒ s → IO (Graph (DeRef s))
```

Unfortunately, that looks a bit for complicated than `vacuum`'s:

```
view :: a → IO ()
```

Question: Why?

Well-Typed

Using `data-reify`

The `MuRef` class is about revealing the recursive structure of our type:

- we need the option to point at a **marker** rather than an actual value,
- so wherever we have a recursive subterm, we need flexibility.

Example:

```
data Expr    = Pl    Expr Expr | One
data ExprF e = PlusF e    e    | OneF
```

Now:

- the type `ExprF Int` is an expression with integers instead of subexpressions,
- the type `ExprF Expr` is isomorphic to the original `Expr` type.

Well–Typed

Instantiating `MuRef`

```
instance MuRef Expr where
  type DeRef Expr = ExprF
  mapDeRef f One         = pure OneF
  mapDeRef f (Pl e1 e2) = PlusF <$> f e1 <*> f e2
```

In `mapDeRef`, we have to explain how to turn an `Expr` into an `ExprF u`, for some applicative function `f`.

The type of `mapDeRef` is somewhat scary:

```
mapDeRef :: (Applicative f, MuRef a) ⇒
            (∀b.(MuRef b, DeRef a ~ DeRef b) ⇒ b → f u) →
            a → f (DeRef a u)
```


Well–Typed

Using reifyGraph

```
⟩ reifyGraph (tree 3)
let [(1, PlusF 2 2), (2, PlusF 3 3), (3, PlusF 4 4), (4, OneF)] in 1
```

Note that this is a pretty-printed version of this type:

```
data Graph e = Graph [(Unique, e Unique)] Unique
type Unique = Int
```


Working with explicitly shared structures

In practice, working with Graph ExprF rather than Expr can be awkward:

- looking up labels in a list,
- an extra indirection even for unshared subtrees,
- possibility to introduce duplicate or dangling labels.

There are other options for handling names and binding, including:

- using string-based names,
- using De-Bruijn-indices,
- using (parametric) higher-order abstract syntax.

None of these options come entirely for free, but if you have to observe recursion and sharing, paying a certain price is unavoidable.

Summary

EDSLs:

- are ubiquitous in Haskell,
- often share monadic or applicative interfaces,
- can use shallow or deep embeddings.

It is not difficult to design your own EDSL.

Features we have not covered in detail:

- adding new effect by changing the underlying monad or applicative functor,
- observing sharing and binding,
- expressing advanced invariants using the type system,
- optimizing by using GHC rewrite rules.

Data-parallel arrays

The Practice of Haskell Programming

Andres Löh

Well-Typed

June 21, 2012

The plan for today

- Unboxed types (type internals, prerequisite).
- The Repa library.

The internals of basic types

```
⟩ :i Int
data Int = GHC.Types.I# GHC.Prim.Int#
```

Aha, so GHC thinks `Int` is yet another datatype?

- The `GHC.Types` and `GHC.Prim` are just module names.
- So there's one constructor, called `I#`.
- And one argument, of type `Int#`.

What is an `Int#`?

The internals of basic types – contd.

To get names like `Int#` even through the parser, we have to enable the `MagicHash` language extension ...

```
⟩ :i GHC.Prim.Int#
data GHC.Prim.Int#   -- Defined in ‘ GHC.Prim ’
```

So this one seems to be really primitive.

Boxed vs. unboxed types

The type `Int#` is the type of **unboxed** integers:

- unboxed integers are essentially machine integers,
- their memory representation is just bits encoding an integer.

An `Int` is a **boxed** integer:

- it wraps the unboxed integer in an additional pointer,
- thereby introducing an indirection.

Boxed vs. unboxed types – contd.

Pro unboxed:

- no indirection,
- faster,
- less space.

Pro boxed:

- only boxed types admit laziness,
- only boxed types admit polymorphism.

Boxing makes all types look alike, making it compatible with thunks and polymorphisms.

Operations on unboxed types

Everything is monomorphic:

```
3#      :: Int#
3##     :: Word#
3.0#    :: Float#
3.0##   :: Double#
'c'#    :: Char#

(+#)       :: Int#    → Int#    → Int#
plusWord#  :: Word#   → Word#   → Word#
plusFloat# :: Float#  → Float#  → Float#
(+##)      :: Double# → Double# → Double#
```


Well-Typed

The kind of unboxed types

GHC uses Haskell's **kind** system to distinguish boxed from unboxed types:

```
〉:k Int
Int :: *
〉:k []
[] :: * → *
〉:k Int#
Int# :: #
```

- Kinds are the types of types.
- Just like programs are type-checked, they're also kind-checked.
- You can get kind errors.

Well-Typed

Kind errors

All these expressions produce kind errors:

```
⟩ let x = undefined :: [ ]
⟩ 3# +# 2
⟩ id 3#
⟩ [3#]
```


Well-Typed

Unpacking strict fields

You typically don't have to use unboxed types directly:

```
data X = C ... {-# UNPACK #-} !Int ...
```

If you have a strict, single-constructor field in a datatype, then the "unpack" pragma instructs GHC:

- to avoid the indirection introduced by the constructor,
- thereby in this case inlining the unboxed `Int#` inside.

Well-Typed

Introducing Repa

A library for data-parallelism in Haskell:

- implemented as an EDSL,
- based on adaptive unboxed arrays,
- offers "delayed" arrays,
- arrays can be re-shaped,
- makes use of advanced type system features,
- offers high-level parallelism.

Repa's arrays

Repa's array type looks as follows:

```
data family Array r sh e    -- abstract
```

There are a number of things worth noting:

- the type is a **data family** – does not affect how we use it, but means that the representation of the array can depend on the parameters (for example, the element type);
- there are **three** type arguments;
- the final is the element type;
- the first denotes the **representation** of the array;
- the second the **shape**.

But what are **representation** and **shape**?

Array shapes

Repa can represent multi-dimensional arrays:

- as a first approximation, the **shape** of an array describes its **dimension**;
- the shape also describes the type of an array **index**.

```
data Z = Z                  -- similar to the () type, Z for "zero"
data t :. h = !t :. !h      -- similar to (,), but strict
type DIM0 = Z
type DIM1 = DIM0 :. Int
type DIM2 = DIM1 :. Int
...
```

So `DIM2` is the type of strict pairs of integers.

Array representations

Repa distinguishes two fundamentally different states an array can be in:

- a **manifest** array is an array that is represented as a block in memory, as we'd expect;
- a **delayed** array is not a real array at all, but merely a computation that describes how to compute each of the elements.

Let's look at the "why" and the delayed representation in a moment.

The standard **manifest** representation is denoted by a type argument `U` (for unboxed).

Creating manifest arrays

```
fromListUnboxed
  :: (Shape sh, Unbox a) ⇒ sh → [a] → Array U sh a
```

Example:

```
〉 fromListUnboxed (Z :. 10 :: DIM1) [1 . . 10 :: Int]
AUnboxed (Z :. 10) (fromList [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
〉 fromListUnboxed (Z :. 2 :. 5 :: DIM2) [1 . . 10 :: Int]
AUnboxed ((Z :. 2) :. 5) (fromList [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

The shape argument provides the dimensions and size of the array; the list must match the size of the shape:

```
〉 size (Z :. 2 :. 5 :: DIM2)
10
```


The `Unbox` class

The `fromListUnboxed` function creates an **adaptive unboxed** array.

The `Unbox` class is defined in the `vector` package:

```
class Unbox a
instance Unbox Int
instance Unbox Float
instance Unbox Double
instance Unbox Char
instance Unbox Bool
instance (Unbox a, Unbox b) ⇒ Unbox (a, b)
```

- Choose an efficient representation depending on element type.
- Represent arrays of tuples as tuples of arrays.

What if our type is not in `Unbox`?

Two options:

- define an `Unbox` instance (tedious, but generally possible);
- use a less efficient manifest array representation (`V`).

For the purposes of this lecture, base types and `U` are sufficient.

Well-Typed

Array access

```
extent :: (Shape sh, Source r e) ⇒ Array r sh e → sh
(!)    :: (Shape sh, Source r e) ⇒ Array r sh e → sh → e
```

```
example :: Array U DIM2 Int
example = fromListUnboxed (Z :. 2 :. 5 :: DIM2) [1 .. 10 :: Int]
```

```
〉 extent example
(Z :. 2) :. 5
〉 x ! (Z :. 1 :. 3)
9
```


Well-Typed

The Source class

The class Source keeps track which element types are allowed for which representation:

```
class Source r e
instance Unbox a ⇒ Source U a
instance           Source V a
```

The unboxed representation is only valid for elements in the Unbox class.

Well-Typed

Operations on arrays

```
map     :: (Shape sh, Source r a) ⇒
           (a → b) → Array r sh a → Array D sh b
extract :: (Shape sh, Source r e) ⇒
           sh → sh → Array r sh e → Array D sh e
(++)    :: (Shape sh, Source r1 e, Source r2 e) ⇒
           Array r1 (sh :. Int) e → Array r2 (sh :. Int) →
           Array D (sh :. Int) e
(*^)    :: (Num c, Shape sh, Source r1 c, Source r2 c) ⇒
           Array r1 sh c → Array r2 sh c → Array D sh c
```

Note:

- What does the shape requirement on (++) tell us?
- All these functions return **delayed** arrays (D).

Well-Typed

Why delayed arrays?

Recall "map fusion":

$$(\mathrm{map}\ f \circ \mathrm{map}\ g)\ xs == \mathrm{map}\ (f \circ g)\ xs$$

- For lists, rather than traversing a list several times, we can traverse it once and do several operations at once.
- However, lists can be traversed one by one. Even if we don't fuse the computations, we only allocate the intermediate cons-cells for the cons-cells we evaluate in the end.
- For arrays, we have to make a full intermediate copy for every traversal, so performing fusion becomes essential – so important that we'd like to make it **explicit** in the type system.

Well-Typed

Delayed arrays

Delayed arrays are internally represented simply as functions:

```
data instance Array D sh e = ADelayed !sh (sh → e)
```

- Delayed arrays aren't really arrays at all.
- Operating on an array does not create a new array.
- Performing another operation on a delayed array just performs function composition.
- If we want to have a manifest array again, we have to **explicitly force** the array.

Well-Typed

Creating delayed arrays

From a function:

```
fromFunction :: sh → (sh → a) → Array D sh a
```

Directly maps to `ADelayed`.

From an arbitrary Repa array:

```
delay :: (Shape sh, Source r e) ⇒ Array r sh e → Array D sh e
```


Well-Typed

The implementation of `map`

```
map :: (Shape sh, Source r a)
    ⇒ (a → b) → Array r sh a → Array D sh b
map f arr = case delay arr of
              ADelayed sh g → ADelayed sh (f ∘ g)
```

Many other functions are only slightly more complicated:

- think about pointwise multiplication `(*^)`,
- or the more general `zipWith`.

Well-Typed

Forcing delayed arrays

Sequentially:

$$\text{computeS} :: (\text{Fill r1 r2 sh e}) \Rightarrow \text{Array r1 sh e} \rightarrow \text{Array r2 sh e}$$

In parallel:

$$\text{computeP} :: (\text{Monad m}, \text{Source r2 e}, \text{Fill r1 r2 sh e}) \Rightarrow \text{Array r1 sh e} \rightarrow \text{m (Array r2 sh e)}$$

The `Fill` class encodes which representations can be converted into which others. The interesting case is:

$$\textbf{instance}\ (\text{Unbox e}, \text{Shape sh}) \Rightarrow \text{Fill D U sh e}$$

"Automatic" parallelism

Behind the scenes:

- Repa starts a gang of threads.
- Depending on the number of available cores, Repa assigns chunks of the array to be computed by different threads.
- The chunking and scheduling and synchronization don't have to concern the user.
- But: Repa **only** supports **flat** data-parallelism! If the delayed computations forced by `computeP` are themselves parallel, Repa will fall back to sequential computation.

Well-Typed

Reducing arrays

Reductions or folds are also available in both sequential and parallel variants:

$$
\begin{aligned}
&\mathsf{sumS} &&:: (\mathsf{Num}\ a, \mathsf{Shape}\ sh, \mathsf{Source}\ r\ a, \mathsf{Unbox}\ a, \mathsf{Elt}\ a) \Rightarrow \\
& && \quad \mathsf{Array}\ r\ (sh :. \mathsf{Int})\ a \rightarrow \mathsf{Array}\ U\ sh\ a \\
&\mathsf{sumP} &&:: (\mathsf{Monad}\ m, \mathsf{Num}\ a, \mathsf{Shape}\ sh, \mathsf{Source}\ r\ a, \mathsf{Unbox}\ a, \mathsf{Elt}\ a) \Rightarrow \\
& && \quad \mathsf{Array}\ r\ (sh :. \mathsf{Int})\ a \rightarrow m\ (\mathsf{Array}\ U\ sh\ a) \\
&\mathsf{sumAllS} &&:: (\mathsf{Num}\ a, \mathsf{Shape}\ sh, \mathsf{Source}\ r\ a, \mathsf{Unbox}\ a, \mathsf{Elt}\ a) \Rightarrow \\
& && \quad \mathsf{Array}\ r\ sh\ a \rightarrow a \\
&\mathsf{sumAllP} &&:: (\mathsf{Monad}\ m, \mathsf{Num}\ a, \mathsf{Shape}\ sh, \mathsf{Source}\ r\ a, \mathsf{Unbox}\ a, \mathsf{Elt}\ a) \Rightarrow \\
& && \quad \mathsf{Array}\ r\ sh\ a \rightarrow m\ a \\
&\mathsf{foldS} &&:: (\mathsf{Shape}\ sh, \mathsf{Source}\ r\ a, \mathsf{Unbox}\ a, \mathsf{Elt}\ a) \Rightarrow \\
& && \quad (a \rightarrow a \rightarrow a) \rightarrow a \rightarrow \mathsf{Array}\ r\ (sh :. \mathsf{Int})\ a \rightarrow \mathsf{Array}\ U\ sh\ a \\
&\mathsf{foldP} &&:: (\mathsf{Monad}\ m, \mathsf{Shape}\ sh, \mathsf{Source}\ r\ a, \mathsf{Unbox}\ a, \mathsf{Elt}\ a) \Rightarrow \\
& && \quad (a \rightarrow a \rightarrow a) \rightarrow a \rightarrow \mathsf{Array}\ r\ (sh :. \mathsf{Int})\ a \rightarrow m\ (\mathsf{Array}\ U\ sh\ a)
\end{aligned}
$$

The constraint `Elt` is comparable to `Unbox`.

Examples

```
example :: Array U DIM2 Int
example = fromListUnboxed (Z :. 2 :. 5) [1 .. 10]
```

```
〉 computeS (map (+ 1) example) :: Array U DIM2 Int
AUnboxed ((Z :. 2) :. 5) (fromList [2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
〉 computeUnboxedS (extract (Z :. 0 :. 1) (Z :. 2 :. 3) example
AUnboxed ((Z :. 2) :. 3) (fromList [2, 3, 4, 7, 8, 9])
〉 sumS it
AUnboxed (Z :. 2) (fromList [9, 24])
〉 sumS it
AUnboxed Z (fromList [33])
〉 sumAllS example
55
```

Goal

- Implement naive matrix multiplication.
- Benefit from parallelism.
- Learn about a few more Repa functions.

This is taken from the `repa-example` package which contains more than just this example.

Start with the types

We want something like this:

```
mmultP :: Monad m ⇒
          Array U DIM2 Double → Array U DIM2 Double →
          m (Array U DIM2 Double)
```

- We inherit the `Monad` constraint from the use of a parallel compute function.
- We work with two-dimensional arrays, it's an additional prerequisite that the dimensions match.

Strategy

We get two matrices of shapes `Z :. h1 :. w1` and `Z :. h2 :. w2`:

- we expect `w1` and `h2` to be equal,
- the resulting matrix will have shape `Z :. h1 :. w2`,
- we have to traverse the rows of the first and the columns of the second matrix, yielding one-dimensional arrays,
- for each of these pairs, we have to take the sum of the products,
- and these results determine the values of the result matrix.

Some observations:

- the result is given by a **function**,
- we need a way to **slice** rows or columns out of a matrix,

Well-Typed

Starting top-down

```
mmultP :: Monad m ⇒
          Array U DIM2 Double → Array U DIM2 Double →
          m (Array U DIM2 Double)
mmultP m1 m2 =
  do
    let (Z :. h1 :. w1) = extent m1
    let (Z :. h2 :. w2) = extent m2
    computeP (fromFunction   (Z :. h1 :. w2)
                             (λ(Z :. r   :. c   ) → ...)
```


Well-Typed

Slicing

A quite useful function offered by Repa is `backpermute`:

```
backpermute :: (Shape sh1, Shape sh2, Source r e) ⇒
               sh2 →                 -- new shape
               (sh2 → sh1) →         -- map new index to old index
               Array r sh1 e → Array D sh2 e
```

- We compute a delayed array simply by saying how each index can be computed in terms of an old index.
- This is trivial to implement in terms of `fromFunction`.

Well-Typed

Slicing – contd.

We can use `backpermute` to slice rows and columns.

```
sliceCol   :: Source r e ⇒ Int → Array r DIM2 e → Array D DIM1 e
sliceCol   c a =
  let (Z :. h :. w) = extent a
  in  backpermute (Z :. h ) (λ(Z :. r ) → (Z :. r :. c)) a
sliceRow   :: Source r e ⇒ Int → Array r DIM2 e → Array D DIM1 e
sliceRow r  a =
  let (Z :. h :. w) = extent a
  in  backpermute (Z :. w) (λ(Z :. c) → (Z :. r :. c)) a
```

```
〉 computeUnboxedS (sliceCol 3 example)
AUnboxed (Z :. 2) (fromList [4, 9])
```

Note that `sliceCol` and `sliceRow` do not actually create a new array unless we force it!

Well-Typed

Slicing – contd.

Repa itself offers are more general slicing function (but it's based on the same idea):

```
slice :: (Slice sl, Shape (SliceShape sl), Shape (FullShape sl),
          Source r e) ⇒
         Array r (FullShape sl) e → sl → Array D (SliceShape sl) e
```

A member of class `Slice`:

- looks similar to a member of class `Shape`,
- but describes **two** shapes at once, the orginal and the sliced.

```
sliceCol, sliceRow :: Source r e ⇒
                      Int → Array r DIM2 e → Array D DIM1 e
sliceCol c  a = slice a (Z :. All :. c  )
sliceRow r a = slice a (Z :. r    :. All)
```


Well-Typed

Putting everything together

```
mmultP :: Monad m ⇒
          Array U DIM2 Double → Array U DIM2 Double →
          m (Array U DIM2 Double)
mmultP m1 m2 =
  do
    let (Z :. h1 :. w1) = extent m1
    let (Z :. h2 :. w2) = extent m2
    computeP (fromFunction (Z :. h1 :. w2)
                (λ(Z :. r :. c) →
                   sumAllS (sliceRow r m1 *^ sliceCol c m2)
                ))
```

That's all. Note that we compute no intermediate arrays.

Well-Typed

Testing it

(Demo.)

Well–Typed

Summary

- The true magic of Repa is in the `computeP` -like functions, where parallelism is automatically handled.
- Haskell's type system is used in various ways:
  - Adapt the representation of unboxed arrays to element types.
  - Keep track of the shape of an array, to make fusion explicit.
  - Keep track of the state of an array.
- We have seen yet another embedded domain-specific language:
  - for efficient array computations,
  - allowing high-level deterministic parallelism,
  - where the types direct us towards correct use.
- A large part of Repa's implementation is actually quite understandable.

Well–Typed