# Generics in Data Parallel Haskell

Roman Leshchinskiy

Programming Languages and Systems
University of New South Wales

Joint work with
Manuel Chakravarty
Gabriele Keller
Simon Peyton Jones

# Programming in DPH

## Multiply sparse vector with dense vector

$$svvm :: [:(Int,\ Float):] \rightarrow [:Float:] \rightarrow Float$$
$$svvm\ v\ w = sumP\ [:x * (w\ !:\ i) \mid (i, x) \leftarrow v:]$$

# Programming in DPH

## Multiply sparse vector with dense vector

$$svvm :: [:(Int,\ Float):] \rightarrow [:Float:] \rightarrow Float$$
$$svvm\ v\ w = sumP\ [:x * (w\ !:\ i) \mid (i, x) \leftarrow v:]$$

## Multiply sparse matrix with dense vector

$$smvm :: [:[:(Int,\ Float):]:] \rightarrow [:Float:] \rightarrow Float$$
$$smvm\ m\ w = [:svvm\ v\ w \mid v \leftarrow m:]$$

# Programming in DPH

## Quicksort

```
qsort xs | lengthP xs ≤ 1 = xs
         | otherwise      = yss !: 0 +:+ eqs +:+ yss !: 1
  where
    m   = xs !: 0
    lts = [:y | y ← xs, y < m :]
    eqs = [:y | y ← xs, y == m:]
    gts = [:y | y ← xs, y > m :]
    yss = [: qsort ys | ys ← [:lts, gts:]:]
```

# Nested data parallelism

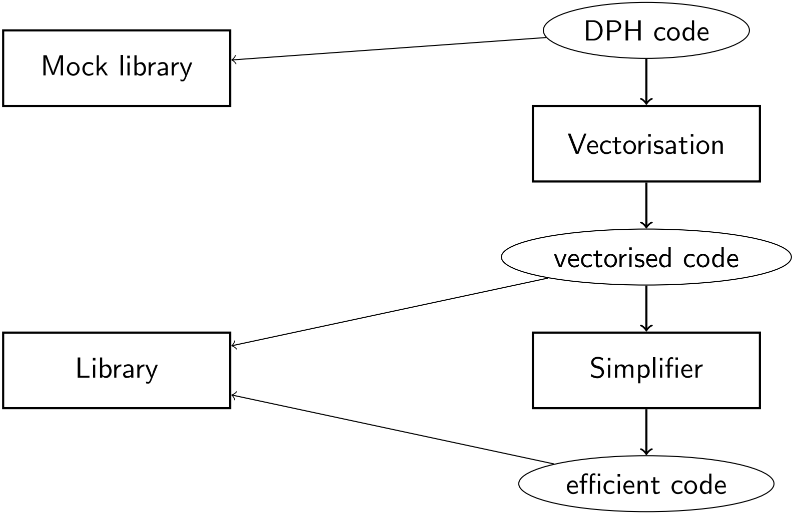
## Great for programmers

- single control flow
- implicit synchronisation and communication
- high-level specification of parallelism
- transparent integration into Haskell

## Bad for implementors

- extensive program transformations
- needs a very good optimiser

# Architecture



# Design

Library

Compiler

# Design

## Library

## Compiler

Type families

# Design

## Library

## Compiler

Type families

Rewrite rules

# Design

## Library

## Compiler

Type families

Simplifier

Rewrite rules

# Design

## Library

## Compiler

Simplifier

Type families

Rewrite rules

Parallel RTS

# Design

Library

Compiler

Type families

Simplifier

Rewrite rules

Parallel RTS

# Design

Library

Compiler

Simplifier

Type families

Rewrite rules

Vectoriser

Parallel RTS

# Design

Library

Compiler

Type families

Simplifier

Rewrite rules

Fusion

Vectoriser

Parallel RTS

# Design

Library

Fusion

Gang parallelism

Compiler

Type families

Simplifier

Rewrite rules

Vectoriser

Parallel RTS

# Design

## Library

Fusion

Gang parallelism

Type transformation

## Compiler

Type families

Simplifier

Rewrite rules

Vectoriser

Parallel RTS

# Design

## Library

Generic operations

Gang parallelism

Fusion

Type transformation

## Compiler

Type families

Simplifier

Rewrite rules

Vectoriser

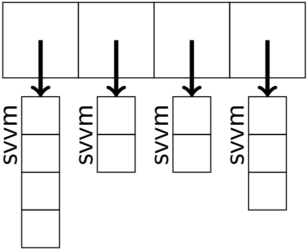
Parallel RTS

# Vectorisation – Basic idea

**Flattening transformation (Blelloch 1995)**

Convert a nested data parallel program into a flat data parallel program

- store nested arrays as flat arrays + segmenting information
- modify computations accordingly

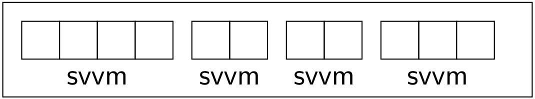


# Vectorisation – Basic idea

**Flattening transformation (Blelloch 1995)**

Convert a nested data parallel program into a flat data parallel program

- store nested arrays as flat arrays + segmenting information
- modify computations accordingly

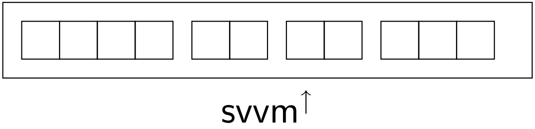


# Vectorisation – Basic idea

**Flattening transformation (Blelloch 1995)**

Convert a nested data parallel program into a flat data parallel program

- store nested arrays as flat arrays + segmenting information
- modify computations accordingly



# Representing arrays

### Non-parametric representation

$$
\begin{aligned}
&\textbf{data family } [{:}a{:}] \\
&\textbf{data instance } [{:}\mathit{Double}{:}] &&= \mathit{PDouble}\ \mathit{ByteArray} \\
&\textbf{data instance } [{:}(a,b){:}] &&= \mathit{PPair}\ [{:}a{:}]\ [{:}b{:}] \\
&\textbf{data instance } [{:}[{:}a{:}]{:}] &&= \mathit{PNest}\ \mathit{Segd}\ [{:}a{:}]
\end{aligned}
$$

- arrays of primitive types $\Longrightarrow$ memory blocks, no boxing!
- arrays of tuples $\Longrightarrow$ tuples of arrays
- nested arrays $\Longrightarrow$ flat data array $+$ segmenting information
- arrays of trees $\Longrightarrow$ trees of arrays

### Sparse matrices

$$
\begin{aligned}
&[{:}[{:}(0,1.3),(2,0.5){:}],[{:}(1,3.4){:}],[{:}{:}],[{:}(0,4.3),(1,2.2){:}]{:}] \\
&\Longrightarrow\ \mathit{PNest}\ \langle 2,1,0,2\rangle\ (\mathit{PPair}\ [{:}0,2,1,0,1{:}]\ [{:}1.3,0.5,3.4,4.3,2.2{:}])
\end{aligned}
$$

# Operations on parallel arrays

How do we implement $indexP :: [:a:] \rightarrow Int \rightarrow a$?

# Operations on parallel arrays

How do we implement $\mathit{indexP} :: [{:}a{:}] \rightarrow \mathit{Int} \rightarrow a$?

$$
\begin{array}{l}
\textbf{class}\ \mathit{PR}\ a\ \textbf{where} \\
\quad \mathit{indexPR} :: [{:}a{:}] \rightarrow \mathit{Int} \rightarrow a \\
\textbf{instance}\ \mathit{PR}\ \mathit{Double}\ \textbf{where} \\
\quad \mathit{indexPR}\ (\mathit{PDouble}\ \mathit{bytes})\ i = \mathit{indexU}\ \mathit{bytes}\ i \\
\textbf{instance}\ (\mathit{PR}\ a,\ \mathit{PR}\ b) \Rightarrow \mathit{PR}\ (a, b)\ \textbf{where} \\
\quad \mathit{indexPR}\ (\mathit{PPair}\ \mathit{xs}\ \mathit{ys})\ i = (\mathit{indexPR}\ \mathit{xs}\ i,\ \mathit{indexPR}\ \mathit{ys}\ i) \\
\\
\mathit{indexP}_V :: \mathit{PR}\ a \Rightarrow [{:}a{:}] \rightarrow \mathit{Int} \rightarrow a \\
\mathit{indexP}_V = \mathit{indexPR}
\end{array}
$$

# Operations on parallel arrays

How do we implement $\mathit{indexP} :: [{:}a{:}] \rightarrow \mathit{Int} \rightarrow a$?

$$\mathbf{data}\ \mathit{PR}\ a = \mathit{PR}\ \{\ \mathit{indexPR} :: [{:}a{:}] \rightarrow \mathit{Int} \rightarrow a\ \}$$

$$\begin{aligned}
&\mathit{dPR\_Double} :: \mathit{PR}\ \mathit{Double}\\
&\mathit{dPR\_Double} = \mathit{PR}\ \{\ \mathit{indexPR} = \lambda(\mathit{PDouble}\ \mathit{bs})\ i \rightarrow \mathit{indexU}\ \mathit{bs}\ i\ \}
\end{aligned}$$

$$\begin{aligned}
&\mathit{dPR\_Pair} :: \mathit{PR}\ a \rightarrow \mathit{PR}\ b \rightarrow \mathit{PR}\ (a, b)\\
&\mathit{dPR\_Pair}\ \mathit{pra}\ \mathit{prb} = \mathit{PR}\ \{\ \mathit{indexPR} = \lambda(\mathit{PPair}\ \mathit{xs}\ \mathit{ys})\ i \rightarrow\\
&\qquad\qquad (\mathit{indexPR}\ \mathit{pra}\ \mathit{xs}\ i,\ \mathit{indexPR}\ \mathit{prb}\ \mathit{ys}\ i)\ \}
\end{aligned}$$

$$\begin{aligned}
&\mathit{indexP}_V :: \mathit{PR}\ a \rightarrow [{:}a{:}] \rightarrow \mathit{Int} \rightarrow a\\
&\mathit{indexP}_V = \mathit{indexPR}
\end{aligned}$$

# Handling user-defined types

```
data Complex a = Complex a a
data instance [:Complex a:] = PComplex [:a:] [:a:]
```

# Handling user-defined types

$$
\begin{array}{l}
\textbf{data}\ \mathit{Complex}\ a \ = \ \mathit{Complex}\ a\ a \\
\textbf{data instance}\ [{:}\mathit{Complex}\ a{:}] \ = \ \mathit{PComplex}\ [{:}a{:}]\ [{:}a{:}]
\end{array}
$$

## Easy?

$$
\begin{array}{l}
\mathit{dPR\_Complex} \ :: \ \mathit{PR}\ a \ \rightarrow \ \mathit{PR}\ (\mathit{Complex}\ a) \\
\mathit{dPR\_Complex}\ \mathit{pa} \ = \ \mathit{PR}\ \{\ \mathit{indexPR} \ = \ \lambda(\mathit{PComplex}\ \mathit{xs}\ \mathit{ys})\ i \ \rightarrow \\
\qquad \mathit{Complex}\ (\mathit{indexPR}\ \mathit{pa}\ \mathit{xs}\ i)\ (\mathit{indexPR}\ \mathit{pa}\ \mathit{ys}\ i)\ \}
\end{array}
$$

# Handling user-defined types

$$\textbf{data}\ Complex\ a\ =\ Complex\ a\ a$$
$$\textbf{data instance}\ [{:}Complex\ a{:}]\ =\ PComplex\ [{:}a{:}]\ [{:}a{:}]$$

### Easy?

$$dPR\_Complex\ ::\ PR\ a\ \rightarrow\ PR\ (Complex\ a)$$
$$dPR\_Complex\ pa\ =\ PR\ \{\ indexPR\ =\ \lambda(PComplex\ xs\ ys)\ i\ \rightarrow$$
$$Complex\ (indexPR\ pa\ xs\ i)\ (indexPR\ pa\ ys\ i)\ \}$$

### Problem

- *PR* has 14 operations at the moment
- there will be more in the future
- compiler has to know how to generate them

# Handling user-defined types

```
data Complex a = Complex a a
data instance [:Complex a:] = PComplex [:a:] [:a:]
```

### Easy with generic programming!

- associate user-defined types with a generic product-sum representation
- fixed set of representation types
- vectoriser generates conversion functions
- “real” operations are implemented in the library

# Generic representations

**type family** $PRepr\ a$
**type instance** $PRepr\ (a, b)\ =\ (a, b)$
**type instance** $PRepr\ (Complex\ a\ a)\ =\ (a, a)$
**type instance** $PRepr\ (Maybe\ a)\ =\ Sum_2\ a\ ()$

# Generic representations

```
type family PRepr a
type instance PRepr (a, b)  =  (a, b)
type instance PRepr (Complex a a)  =  (a, a)
type instance PRepr (Maybe a)  =  Sum2 a ()
```

```
data PA a  =  PA {
  toPRepr        :: a → PRepr a
  fromPRepr      :: PRepr a → a
  toArrPRepr     :: [:a:] → [:PRepr a:]
  fromArrPRepr   :: [:PRepr a:] → [:a:]
  dictPRepr      :: PR (PRepr a)}
```

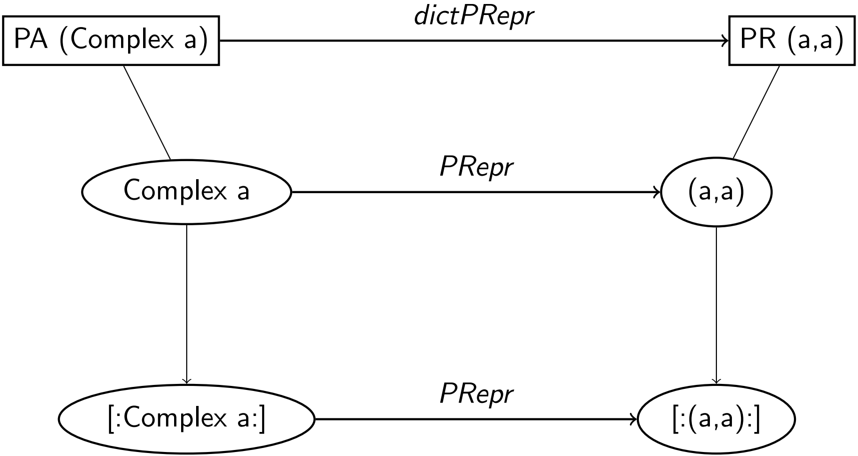
# Generic representations

```
type family PRepr a
type instance PRepr (a, b) = (a, b)
type instance PRepr (Complex a a) = (a, a)
type instance PRepr (Maybe a) = Sum2 a ()
```

```
data PA a = PA {
   toPRepr       :: a → PRepr a
   fromPRepr     :: PRepr a → a
   toArrPRepr    :: [:a:] → [:PRepr a:]
   fromArrPRepr :: [:PRepr a:] → [:a:]
   dictPRepr     :: PR (PRepr a)}
```

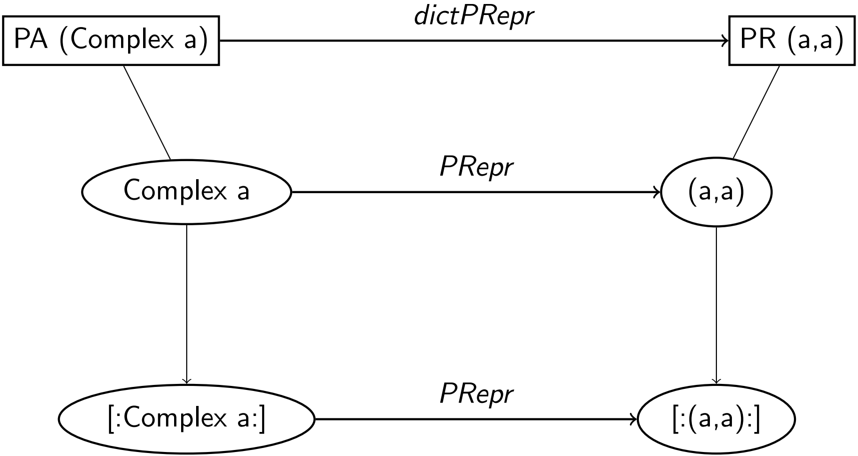
```
dPA_Complex :: PA a → PA (Complex a)
dPA_Complex pa = PA { toPRepr = λ(Complex x y) → (x, y), . . . }
```

# Generic operations



$$\begin{tikzcd}
\boxed{\text{PA (Complex a)}} \arrow[rr, "\mathit{dictPRepr}"] \arrow[d, no head] & & \boxed{\text{PR (a,a)}} \arrow[d, no head] \\
\text{Complex a} \arrow[rr, "\mathit{PRepr}"] \arrow[d] & & \text{(a,a)} \arrow[d] \\
\text{[:Complex a:]} \arrow[rr, "\mathit{PRepr}"] & & \text{[:(a,a):]}
\end{tikzcd}$$

# Generic operations



PA (Complex a)

*dictPRepr*

PR (a,a)

Complex a

*PRepr*

(a,a)

[:Complex a:]

*PRepr*

[:(a,a):]

**type instance** *PRepr* [:*a*:] = [:*PRepr a*:]

# Implementing generic operations

## Library

$$
\begin{aligned}
&\mathit{dPR\_Pair} :: \mathit{PR}\ a \to \mathit{PR}\ b \to \mathit{PR}\ (a, b)\\
&\mathit{dPR\_Pair}\ \mathit{pra}\ \mathit{prb} = \mathit{PR}\ \{\ \mathit{indexPR} = \lambda(\mathit{PPair}\ \mathit{xs}\ \mathit{ys})\ i \to\\
&\qquad\qquad (\mathit{indexPR}\ \mathit{pra}\ \mathit{xs}\ i,\ \mathit{indexPR}\ \mathit{prb}\ \mathit{ys}\ i)\ \}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{indexP}_V :: \mathit{PA}\ a \to [{:}a{:}] \to \mathit{Int} \to a\\
&\mathit{indexP}_V\ \mathit{pa}\ \mathit{xs}\ i = \mathit{fromPRepr}\ \mathit{pa}\\
&\qquad\qquad (\mathit{indexPR}\ (\mathit{dictPRepr}\ \mathit{pa})\ (\mathit{toArrPRepr}\ \mathit{pa}\ \mathit{xs})\ i)
\end{aligned}
$$

# Implementing generic operations

## Library

$$
\begin{aligned}
&\mathit{dPR\_Pair} \;::\; \mathit{PR}\,a \rightarrow \mathit{PR}\,b \rightarrow \mathit{PR}\,(a,b)\\
&\mathit{dPR\_Pair}\;\mathit{pra}\;\mathit{prb} \;=\; \mathit{PR}\,\{\,\mathit{indexPR} \;=\; \lambda(\mathit{PPair}\;\mathit{xs}\;\mathit{ys})\;i \rightarrow\\
&\qquad\qquad (\mathit{indexPR}\;\mathit{pra}\;\mathit{xs}\;i,\;\mathit{indexPR}\;\mathit{prb}\;\mathit{ys}\;i)\,\}
\end{aligned}
$$

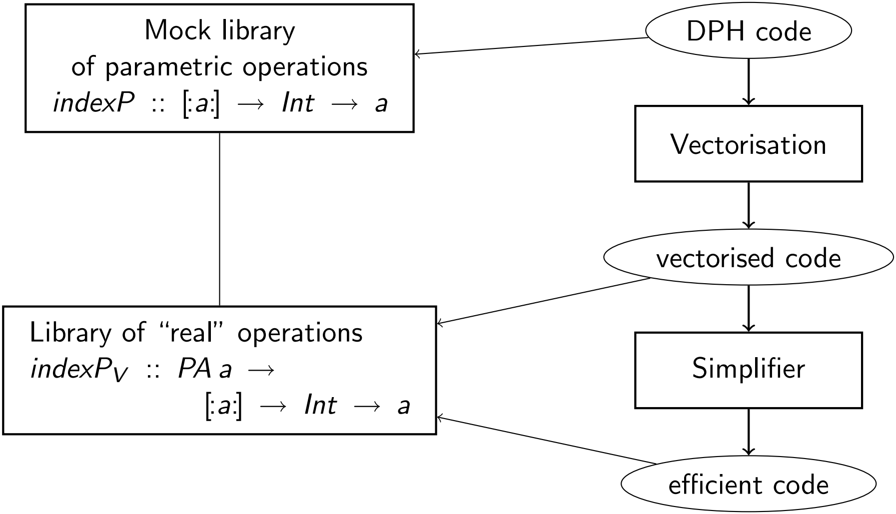
$$
\begin{aligned}
&\mathit{indexP}_V \;::\; \mathit{PA}\,a \rightarrow [{:}a{:}] \rightarrow \mathit{Int} \rightarrow a\\
&\mathit{indexP}_V\;\mathit{pa}\;\mathit{xs}\;i \;=\; \mathit{fromPRepr}\;\mathit{pa}\\
&\qquad\qquad (\mathit{indexPR}\,(\mathit{dictPRepr}\;\mathit{pa})\,(\mathit{toArrPRepr}\;\mathit{pa}\;\mathit{xs})\;i)
\end{aligned}
$$

## Vectoriser

$$
\begin{aligned}
&\textbf{data instance}\;[{:}\mathit{Complex}\;a{:}] \;=\; \mathit{Complex}\;[{:}a{:}]\;[{:}a{:}]\\
&\textbf{type instance}\;\mathit{PRepr}\,(\mathit{Complex}\;a) \;=\; (a,a)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{dPA\_Complex} \;::\; \mathit{PA}\,a \rightarrow \mathit{PA}\,(\mathit{Complex}\;a)\\
&\mathit{dPA\_Complex}\;\mathit{pa} \;=\; \mathit{PA}\,\{\,\mathit{toPRepr} \;=\; \lambda(\mathit{Complex}\;x\;y) \rightarrow (x,y)\;\ldots\,\}
\end{aligned}
$$

# Architecture – now with one more line!



Mock library
of parametric operations
$indexP :: [:a:] \to Int \to a$

DPH code

Vectorisation

vectorised code

Library of “real” operations
$indexP_V :: PA\ a \to [:a:] \to Int \to a$

Simplifier

efficient code

# Type families + generics are good ...

# Type families + generics are good ...

### ... for DPH

- minimise knowledge built into the vectoriser
- keep library code in the library
- very efficient

# Type families + generics are good ...

## ... for DPH

- minimise knowledge built into the vectoriser
- keep library code in the library
- very efficient
- we can give talks about it!

# Type families + generics are good ...

### ... for DPH

- minimise knowledge built into the vectoriser
- keep library code in the library
- very efficient
- we can give talks about it!

### ... for everything else

- new approach to implementing generics
- design pattern directly supported by the language
- no performance penalty
- we can write papers about it!