

A Haskell Model of BSP Computations

Cezar Ionescu, Nicola Botta

An informal description of BSP computations

1. BSP belongs to the class of SPMD models
2. fixed number of processors, each with its own memory
3. one data exchange primitive, similar to the MPI all-to-all function.

An informal description of BSP computations

4. The computer can be in one of two states
 - (a) local computation: every processor is performing some computation using only the data in its own memory space and its own *processor id*
 - (b) communication: every processor is involved in the data exchange operation

A local computation stage followed by a data exchange is called a *superstep*, and a BSP program can be viewed as a sequence of supersteps.

The local computation stage

We need a description of the data associated with each processor:

$$d : Proc \rightarrow A$$

In Haskell

```
newtype D a = D (Proc → a)
```

Examples:

```
pid :: D Proc
```

```
pid = D id
```

```
D (const a) :: D a
```

The local computation stage

A local computation acts on the data associated with each processor *and* on the processor id:

$$f : A \times Proc \rightarrow B$$

$$apply : (A \times Proc \rightarrow B) \rightarrow D A \rightarrow D B$$

$$apply\ f\ da = db$$

$$\text{where } db\ p = f\ (da\ p, p)$$

The local computation stage

Alternative (Loulergue):

$$\mathit{apply}' : D (A \rightarrow B) \rightarrow D A \rightarrow D B$$

$$\mathit{apply}' \, df \, da = db$$

$$\mathbf{where} \, db \, p = (df \, p) (da \, p)$$

apply and apply' are equivalent: either can be defined in terms of the other.

The local computation stage

We adopt yet another version!

$bind : D A \rightarrow (A \rightarrow D B) \rightarrow D A$

$bind\ da\ f = db$

where $db\ p = (f\ (da\ p))\ p$

Either of *bind*, *apply* and *apply'* can be defined in terms of any of the other.

The local computation stage

We prefer *bind* because it shows that *D* is a *monad*, which is interesting, and allows us to take advantage of special Haskell notation.

instance *Monad D* **where**

return = D const

(▷) = bind

The local computation stage

The **do** notation:

```
apply f db = do  
  b ← db  
  p ← pid  
  return (f (b, p))
```

The communication stage

In the communication stage, every processor may send to or receive from any other processor one or more elements of some datatype a .

$$exch :: D [(a, Proc)] \rightarrow D [(a, Proc)]$$
$$exch (D dxs) = D dys$$

where

$$dys\ p = [(x, p') \mid p' \leftarrow Proc, \\ (x, p'') \leftarrow (dxs\ p'), p'' \equiv p]$$

The actual implementation of $exch$ will be very different from its description, (for example, $exch$ will be constrained to act on types which are instances of “Plain Old Data”)

The communication stage

Example: on every partition, compute the length of the “global” list.

$$dlen :: D [a] \rightarrow D Int$$

First, we compute the length of the local lists:

$$dlens :: D [a] \rightarrow D Int$$
$$dlens dxs = \mathbf{do}$$
$$xs \leftarrow dxs$$
$$\mathbf{return} (\mathit{length} xs)$$

Then, we prepare the result for *exch*:

$$dlens' :: D Int \rightarrow D [(Int, Proc)]$$
$$dlens' dl = \mathbf{do}$$
$$l \leftarrow dl$$
$$\mathbf{return} [(l, p) \mid p \leftarrow Proc]$$

The communication stage

After *exch*, we add locally all results obtained:

```
dlens'' :: D [(Int, Proc)] → D Int  
dlens'' dnps = do  
    nps ← dnps  
    return (sum (map fst nps))
```

The final program:

```
dlen = dlens'' ∘ exch ∘ dlens' ∘ dlens
```

The communication stage

Typical pattern seen in *dlens'*:

```
repl :: D a → D [(a, Proc)]  
repl dx = do  
    x ← dxs  
    return [(x, p) | p ← Proc]
```

We can formulate and prove properties of *exch*, for instance:

$exch \circ exch = id$

$exch \circ repl$ is a constant distributed value

An aside

The approach to program development outlined in “Algebra of Programming” [Bird97] often leads to programs that operate on sets:

The examples above indicate our approach to program derivation. Specify the problem using relations. After due process of calculation, we will end up with a relation that we want to implement. If this relation is a function [...], well and good. If not, then translate it into a set-valued function and, provided the function returns finite sets, implement it using suitable data structures, such as lists, trees, hash tables, and whatnot.

An aside

Main objective:

We would like to extend some of results of the “Algebra of Programming” by considering *distributed representations* of data structures and of sequential programs, especially of the main combinators, such as folds and unfolds.

Representations

When can we say that

$$dx :: D a$$

is a distributed representation of

$$x :: a$$

?

Representations

No general answer (the representation condition can be application dependent). But we can give typical examples:

$dX :: D (P \ A)$ is a d. r. of $X \subset A$ if

$$X = \cup \{ dX \ p \mid p \in Proc \}$$

$dxs :: D [a]$ is a d. r. of $xs :: [a]$ if

$$xs = concat [dxs \ p \mid p \leftarrow Proc]$$

$dx :: D \ Float$ is a d. r. of $x :: \Float$ if

$$x = (1 / np) * sum [dx \ p \mid p \leftarrow Proc]$$

Representations

In all these examples, we see representation as a (surjective) function

$$d2g :: D\ a \rightarrow a$$

We can define now a function $df :: D\ a \rightarrow D\ b$ to be a d. r. of $f :: a \rightarrow b$ if

$$d2g \circ df = f \circ d2g$$

Another example:

A function $df :: A \times D B \rightarrow D C$ is a d. r. of $f :: A \times B \rightarrow C$ if

$$d2g \circ df = f \circ (id \times d2g)$$

More generally:

$df :: F (D A) \rightarrow G (D B)$ is a d. r. of $f :: F A \rightarrow G B$ if

$$(G d2g) \circ df = f \circ (F d2g)$$

Folds and iteration

Iteration is described by the *foldn* function on *Nat*:

$$\textit{foldn} :: (a \rightarrow a) \rightarrow a \rightarrow \textit{Nat} \rightarrow a$$

$$\textit{foldn} f e 0 = e$$

$$\textit{foldn} f e (n + 1) = f (\textit{foldn} f e n)$$

$$\textit{foldn} f e n = f^n e$$

Assume that $df :: d\ a \rightarrow d\ a$ is a d. r. of f and $de :: d\ a$ is a d. r. of e . Is it the case that $\textit{foldn} df de$ is a distributed representation of $\textit{foldn} f e$?

Folds and iteration

The answer is yes: iterating a distributed representation of a function gives a distributed representation of iterating the function.

The result can be easily shown for arbitrary folds as an instance of the *fusion theorem* for catamorphisms.

What about unfolds?

Folds and iteration

Unbounded iteration on *Nat*:

$$\begin{aligned} \text{unfoldn} &:: (a \rightarrow \text{Bool}) \rightarrow (a \rightarrow a) \rightarrow a \rightarrow \text{Nat} \\ \text{unfoldn } p \ f \ e &= \mathbf{if } p \ e \ \mathbf{then } 0 \ \mathbf{else } 1 + \text{unfoldn } p \ f \ (f \ e) \end{aligned}$$

We can see that it is not sufficient to have df and de , one also needs a distributed representation of p !

But a d. r. of p would have to satisfy

$$\begin{aligned} dp &:: D \ a \rightarrow \text{Bool} \\ dp &= p \circ d2g \end{aligned}$$

and we have no way of implementing this in terms of the functions we have so far for $D \ a$ values (*bind*, *return* and *exch*).

Val

Solution:

$val :: D\ a \rightarrow a$
 $val\ dx = x, \mathbf{if}\ dx = return\ x$
 $\perp, otherwise$

With *val* we can implement *dp*, and we obtain, as before, that the unfold of a d. r. of *f* is a d. r. of the unfold of *f*.

Conclusions

The Haskell monadic model of BSP computations:

1. $D\ a$ the datatype of distributed values of type a
2. **instance** *Monad* D declaration for D allowing the use of the **do** notation and modeling the local computations stage
3. *exch* the only communication primitive, modeling the communication stage

Conclusions

4. definition of “distributed representations”
5. *val* a limited “escape” from the D monad for implementing unbounded iteration or unfolds.