Dependent Types for Distributed Arrays

Wouter Swierstra and Thorsten Altenkirch

University of Nottingham
{wss,txa}@cs.nott.ac.uk

Abstract. Locality-aware algorithms over distributed arrays can be very difficult to write. Yet such algorithms are becoming more and more important as desktop machines boast more and more processors. We show how a dependently-typed programming language can help develop such algorithms by hosting a domain-specific embedded type system that ensures every well-typed program will only ever access local data. Such static guarantees can help catch programming errors early on in the development cycle and maximise the potential speedup that multicore machines offer. At the same time, the functional specification of effects we provide facilitates the testing of and reasoning about algorithms on distributed arrays.

1 Introduction

Computer processors are not becoming significantly faster. To satisfy the demand for more and more computational power, manufacturers are now assembling computers with multiple microprocessors. It is hard to exaggerate the impact this will have on software development: tomorrow's programming languages must embrace parallel programming on multicore machines.

Researchers have proposed several new languages to maximise the potential speedup that multicore processors offer [2, 7–9]. Although all these languages are different, they share the central notion of a *distributed array*, where the elements of an array may be distributed over separate processors or even over separate machines. To write efficient code, programmers must ensure that processors only access *local* parts of a distributed array—it is much faster to access data stored locally than remote data on another core.

When writing such locality-aware algorithms it is all too easy to make subtle mistakes. Programming languages such as X10 require all arrays operations to be local [9]. Any attempt to access non-local data results in an exception. To preclude such errors, Grothoff *et al.* have designed a type system for a small core language resembling X10 that is specifically designed to guarantee that programs only access local parts of a distributed array [10]. Yet it is somehow disappointing that these static guarantees require a custom-built type system to solve a very specific problem.

In this paper, we explore an alternative avenue of research. Designing and implementing a type system from scratch is a lot of work. New type systems typically require extensive proofs of various soundness, completeness, principle

typing, and decidability theorems. Instead, we show how to tailor a sufficiently expressive type system to enforce certain properties—resulting in a *domain-specific embedded type system*. We no longer need to prove any meta-theoretical results, but immediately inherit all the desirable properties of our host type system. Functional programmers have studied domain-specific embedded languages for years [12]; the time is ripe to take these ideas one step further.

In previous work [19], we described a pure specification of several parts of the IO monad, the interface between pure functional languages such as Haskell [17] and the 'real world.' By providing functional, executable specifications we can test, debug, and reason about impure programs as if they were pure. When we now shift to a dependently-typed programming language, we can show how our specifications can provide stronger static guarantees about our programs. To this end, we make several novel contributions:

- We begin by giving a pure specification of arrays (Section 3). Due to our rich type system, the specification is *total*: there is no way to access unallocated memory; there are no 'array index out of bounds' exceptions. As a result, these specifications can not only be used to *program* with, but also facilitate *formal proofs* about array algorithms.
- Distributed arrays pose more of a challenge (Section 4). Not only do we attend to locality constraints, but we must also accommodate place-shifting operators. The pure specification we present is, once again, executable and total: it can be interpreted both as a domain-specific embedded language for writing algorithms on distributed arrays and as an executable denotational model for specifying and proving properties of such algorithms.
- Finally, we demonstrate how programmers may write their own locality-aware control structures using the primitives we have defined. We also implement several combinators that enable programmers to define their own distributions, specifying how large arrays must be distributed over different processors. We conclude by discussing how our specifications may be extended to cope with multidimensional arrays with more complex regions (Section 5).

Throughout this paper, we will use the dependently-typed programming language Agda [1, 15] as a vehicle of explanation. In fact using lhs2TeX [13], the sources of this paper generate an Agda program that can be compiled and executed.[1] We will briefly introduce the syntax by means of several examples, as it may be unfamiliar to many readers.

2 An overview of Agda

Data types in Agda can be defined using a similar syntax to that for Generalized Algebraic Data Types, or GADTs, in Haskell [**?**]. For example, consider the following definition of the natural numbers.

[1] The resulting code is available from the first author's website.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Nat} : \star\ \textbf{where} \\
\quad \mathit{Zero} : \mathit{Nat} \\
\quad \mathit{Succ} : \mathit{Nat} \to \mathit{Nat}
\end{array}
$$

There is one important difference with Haskell. We must explicitly state the *kind* of the data type that we are introducing; in particular, the declaration $\mathit{Nat} : \star$ states that *Nat* is a base type.

We can define functions by pattern matching and recursion, just as in any other functional language. To define addition of natural numbers, for instance, we could write:

$$
\begin{array}{l}
\_ + \_ : \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \\
\mathit{Zero} + m \quad = m \\
\mathit{Succ}\ n + m = \mathit{Succ}\ (n + m)
\end{array}
$$

Note that Agda uses underscores to denote the positions of arguments when defining new operators.

Polymorphic lists are slightly more interesting than natural numbers:

$$
\begin{array}{l}
\textbf{data}\ \mathit{List}\ (a : \star) : \star\ \textbf{where} \\
\quad \mathit{Nil} \quad : \mathit{List}\ a \\
\quad \mathit{Cons} : a \to \mathit{List}\ a \to \mathit{List}\ a
\end{array}
$$

To uniformly parameterise a data type, we can write additional arguments to the left of the copula. In this case, we add $(a : \star)$ to our data type declaration to state that lists are type *constructors*, parameterised over a type variable a that has kind $\star$.

Just as we defined addition for natural numbers, we can define an operator that appends one list to another:

$$
\begin{array}{l}
\mathit{append} : (a : \star) \to \mathit{List}\ a \to \mathit{List}\ a \to \mathit{List}\ a \\
\mathit{append}\ a\ \mathit{Nil}\ ys \qquad\quad = ys \\
\mathit{append}\ a\ (\mathit{Cons}\ x\ xs)\ ys = \mathit{Cons}\ x\ (\mathit{append}\ a\ xs\ ys)
\end{array}
$$

The *append* function is polymorphic. In Agda, such polymorphism can be introduced via the *dependent function space*, written $(x : a) \to y$, where the variable x may occur in the type y. This particular example of the dependent function space is not terribly interesting: it merely expresses parametric polymorphism. Later we will encounter more interesting examples, where *types* depend on *values*.

One drawback of using the dependent function space for such parametric polymorphism, is that we must explicitly instantiate polymorphic functions. For example, the recursive call to *append* in the *Cons* case takes a type as its first argument. Fortunately, Agda allows us to mark certain arguments as *implicit*. Using implicit arguments, we can define the $+\!\!+$ operator just as in any other functional language:

$$
\begin{array}{l}
\_ +\!\!+ \_ : \{\, a : \star \,\} \to \mathit{List}\ a \to \mathit{List}\ a \to \mathit{List}\ a \\
\mathit{Nil} +\!\!+ ys \qquad\quad = ys \\
(\mathit{Cons}\ x\ xs) +\!\!+ ys = \mathit{Cons}\ x\ (xs +\!\!+ ys)
\end{array}
$$

Arguments enclosed in curly brackets, such as $\{a : \star\}$, are implicit: we do not write a to the left of the equals sign and do not pass a type argument when we call the make a recursive call. The Agda type checker will automatically instantiate this function whenever we call it, much in the same way as type variables are instantiated in Haskell.

Besides polymorphic data types, Agda also supports *indexed families*. Like Haskell's GADTs, indexed families allow a data type's constructors to have different codomains. Indexed families, however, are more general as they also capture data types that are indexed by *values* instead of types. For example, we can define the family of finite types:

$$\begin{aligned}&\textbf{data}\ \mathit{Fin} : \mathit{Nat} \to \star\ \textbf{where}\\ &\quad \mathit{Fz} : \{\, n : \mathit{Nat} \,\} \to \mathit{Fin}\ (\mathit{Succ}\ n)\\ &\quad \mathit{Fs} : \{\, n : \mathit{Nat} \,\} \to \mathit{Fin}\ n \to \mathit{Fin}\ (\mathit{Succ}\ n)\end{aligned}$$

The type $\mathit{Fin}\ n$ corresponds to a finite type with n distinct values. For example, $\mathit{Fin}\ 1$ is isomorphic to the unit type; $\mathit{Fin}\ 2$ is isomorphic to Bool. Note that the argument n is left implicit in both the constructors of Fin. From the types of these constructors, it is easy to see that $\mathit{Fin}\ 0$ is uninhabited. For every n, the Fs constructor embeds $\mathit{Fin}\ n$ into $\mathit{Fin}\ (\mathit{Succ}\ n)$; the Fz constructor, on the other hand, adds a single new element to $\mathit{Fin}\ (\mathit{Succ}\ n)$ that was not in $\mathit{Fin}\ n$. By induction it is easy to see that $\mathit{Fin}\ n$ does indeed have n elements.

Agda has many other features, such as records and a rich module system, that we will hardly use in this paper. Although there are a few more concepts we will need, we will discuss them as they pop up in later sections.

3 Mutable arrays

With this brief Agda tutorial under our belt, we can start our specification of mutable arrays. We will specify three different operations on arrays: the creation of new arrays; reading from an array; and updating a value stored in an array. Before we can define the behaviour of these operations, however, we need to introduce several data types to describe the layout and contents of memory. Using these data types, we can proceed by defining an IO type that captures the syntax of array operations. Finally, we define a *run* function that describes how the array operations affect the heap, assigning semantics to our syntax. This semantics can be used to simulate and reason about computations on mutable arrays in a pure functional language. When compiled, however, these operations should be replaced by their more efficient, low-level counterparts.

To keep things simple, we will only work with flat arrays storing natural numbers. This is, of course, a drastic oversimplification. The techniques we present here, however, can be adapted to cover multidimensional arrays that may store different types of data (Section 5).

To avoid confusion between numbers denoting the size of an array and the data stored in an array, we introduce the Data type synonym. Throughout the

rest of this paper, we will use *Data* to refer to the data stored in arrays; the *Nat* type will always refer to the size of an array.

```
Data : ⋆
Data = Nat
```

Using the *Fin* type, we can give a functional specification of arrays of a fixed size by mapping every index to the corresponding datum.

```
Array : Nat → ⋆
Array n = Fin n → Data
```

How should we represent the heap? We need to be a bit careful—as the heap will store arrays of different sizes, its type should make it clear how many arrays it stores and how large each array is. To accomplish this, we begin by introducing a data type representing the *shape* of the heap:

```
Shape : ⋆
Shape = List Nat
```

The *Shape* of the heap is simply a list of natural numbers, representing the size of the arrays stored in memory.

We can now define a *Heap* data type that is indexed by a *Shape*. The *Empty* constructor corresponds to an empty heap; the *Alloc* constructor adds an array of size n to any heap of shape ns to build a larger heap with the larger layout $Cons\ n\ ns$.

```
data Heap : Shape → ⋆ where
  Empty : Heap Nil
  Alloc : {n : Nat} → {ns : Shape} →
    Array n → Heap ns → Heap (Cons n ns)
```

Finally, we will want to model references, denoting locations in the heap. A value of type $Loc\ n\ ns$ corresponds to a reference to an array of size n in a heap with shape ns. The *Loc* data type shares a great deal of structure with the *Fin* type. Every non-empty heap has a *Top* reference; we can weaken any existing reference to denote the same location in a larger heap using the *Pop* constructor.

```
data Loc : Nat → Shape → ⋆ where
  Top : {n : Nat} → {ns : Shape} → Loc n (Cons n ns)
  Pop : forall {n k ns} → Loc n ns → Loc n (Cons k ns)
```

Note that in the type signature of the *Pop* constructor, we omit the types of three implicit arguments and quantify over them using the **forall** keyword. Alternatively, we could also have written the more verbose:

```
Pop : {n : Nat} → {k : Nat} → {ns : Shape} →
  Loc n ns → Loc n (Cons k ns)
```

When we use the **forall** keyword, the types of n, k, and ns are inferred from the rest of the signature by the Agda type checker. We will use this shorthand notation to make large type signatures somewhat more legible.

With these data types in place, we can define a data type capturing the syntax of the permissible operations on arrays. Crucially, the *IO* type is indexed by *two* shapes: a value of type $IO\ a\ ns\ ms$ denotes a computation that takes a heap of shape ns to a heap of shape ms and returns a result of type a. This pattern of indexing operations by an initial and final 'state' is a common pattern in dependently-typed programming [14].

```
data IO (a : ⋆) : Shape → Shape → ⋆ where
  Return : {ns : Shape} → a → IO a ns ns
  Write : forall {n ns ms} →
    Loc n ns → Fin n → Data → IO a ns ms → IO a ns ms
  Read : forall {n ns ms} →
    Loc n ns → Fin n → (Data → IO a ns ms) → IO a ns ms
  New : forall {ns ms} →
    (n : Nat) → (Loc n (Cons n ns) → IO a (Cons n ns) ms) →
    IO a ns ms
```

The *IO* type has four constructors. The *Return* constructor returns a pure value of type a without modifying the heap. The *Write* constructor takes four arguments: the location of an array of size n; an index in that array; the value to write at that index; and the rest of the computation. Similarly, reading from an array requires a reference to an array and a suitable index. Instead of requiring the data to be written, however, the last argument of the *Read* constructor may refer to data that has been read. Finally, the *New* constructor actually changes the size of the heap. Given a number n, it allocates an array of size n on the heap; the second argument of *New* may then use this fresh reference to continue the computation in a larger heap.

The *IO* data type is a *parameterised monad* [4]—that is, a monad with *return* and bind operators that satisfy certain coherence conditions regarding how they may affect the heap.

```
return : {a : ⋆} → {ns : Shape} → a → IO a ns ns
return x = Return x

_>>=_ : forall {a b ns ms ks} →
  IO a ns ms → (a → IO b ms ks) → IO b ns ks
Return x >>= f       = f x
Write a i x wr >>= f = Write a i x (wr >>= f)
Read a i rd >>= f    = Read a i (λx → rd x >>= f)
New n io >>= f       = New n (λa → io a >>= f)
```

The return of the *IO* data type lifts a pure value into a computation that can run on a heap of any size. Furthermore, *return* does not modify the shape of the heap. The bind operator, $\gg\!=$, can be used to compose monadic computations.

To sequence two computations, the heap resulting from the first computation must be a suitable starting point for the second computation. This condition is ensured by the type of the bind operator:

To actually write programs using these arrays, we need to introduce a smart constructor for the *IO* data type. For example, we could define the *readArray* function as follows:

```
readArray : forall {n ns} → Loc n ns → Fin n → IO Data ns ns
readArray a i = Read a i Return
```

There is a slight problem with this definition. As we allocate new memory, the size of the heap changes; correspondingly, we must explicitly modify any existing pointers to denote locations in a larger heap. We can achieve this by revising the above definition slightly, ensuring that an additional *Pop* constructor is wrapped around any existing references. For the purpose of this paper, however, we will ignore this technicality: the above definition will suffice.

Denotational model

We have described the syntax of array computations using the *IO* data type, we have not specified how these computation *behave*. Recall that we can model arrays as functions from indices to natural numbers:

```
Array : Nat → ⋆
Array n = Fin n → Data
```

Before specifying the behaviour of *IO* computations, we define a several auxiliary functions to update an array and lookup a value stored in an array.

```
lookup : forall {n ns} → (l : Loc n ns) → Fin n → Heap ns → Data
lookup Top      i (Alloc a _) = a i
lookup (Pop k) i (Alloc _ h) = lookup k i h
```

The *lookup* function takes a reference to an array l, an index i in the array at location l, and a heap, and returns the value stored in the array at index i. It dereferences l, resulting in a function of type $Fin\ n \to Data$; the value stored at index i is the result of applying this function to i.

Next, we define a pair of functions to update the contents of an array.

```
updateArray : {n : Nat} → Fin n → Data → Array n → Array n
updateArray i d a = λj → if i ≡ j then d else a j

updateHeap : forall {n ns} →
  Loc n ns → Fin n → Data → Heap ns → Heap ns
updateHeap Top      i x (Alloc a h) = Alloc (updateArray i x a) h
updateHeap (Pop k) i x (Alloc a h) = Alloc a (updateHeap k i x h)
```

The *updateArray* function overwrites the data stored at a single index. The *updateHeap* function updates a single index of an array stored in the heap. It

proceeds by dereferencing the location on the heap where the desired array is stored and updates it accordingly, leaving the rest of the heap unchanged.

Now we finally have all the pieces in place to assign semantics to *IO* computations. The *run* function below takes a computation of type *IO a ns ms* and an initial heap of shape *ns* as arguments, and returns a pair consisting of the result of the computation and the final heap of shape *ms*.

```
data Pair (a : ⋆) (b : ⋆) : ⋆ where
  pair : a → b → Pair a b
run : forall {a ns ms} → IO a ns ms → Heap ns → Pair a (Heap ms)
run (Return x) h        = pair x h
run (Read a i rd) h     = run (rd (lookup a i h)) h
run (Write a i x wr) h = run wr (updateHeap a i x h)
run (New n io) h        = run (io Top) (Alloc (λi → Zero) h)
```

The *Return* constructor simply pairs the result and heap; in the *Read* case, we lookup the data from the heap and recurse with the same heap; for the *Write* constructor, we recurse with an appropriately modified heap; finally, when a new array is created, we extend the heap with a new array that stores *Zero* at every index, and continue recursively. Note that, by convention, the *Top* constructor always refers to the most recently created reference. Our smart constructors should add additional *Pop* constructors when new memory is allocated.

Example Using our smart constructors and the monad operators, we can now define functions that manipulate arrays. For example, the swap function exchanges the value stored at two indices:

```
swap : forall {n ns} → Loc n ns → Fin n → Fin n → IO () ns ns
swap a i j =
  readArray a i >>= λvali →
  readArray a j >>= λvalj →
  writeArray a i valj >>
  writeArray a j vali
```

In a dependently-typed programming language such as Agda, we can prove properties of our code. For example, we may want to show that swapping the contents of any two array indices twice, leaves the heap intact :

```
swapProp : {n : Nat}{ns : Shape} →
  (l : Loc n ns) → (i : Fin n) → (j : Fin n) → (h : Heap ns) →
  (h ≡ snd (run (swap l i j >> swap l i j) h))
```

The proof requires a lemma about how *updateHeap* and *lookupHeap* interact and is not terribly interesting in itself. The fact that we can formalise such properties and have our proof verified by a computer is much more exciting.

4 Distributed arrays

Arrays are usually represented by a continuous block of memory. *Distributed arrays*, however, can be distributed over different *places*—where every place may correspond to a different core on a multiprocessor machine, a different machine on the same network, or any other configuration of interconnected computers.

We begin by determining the type of places, where data is stored and code is executed. Obviously, we do not want to fix the type of all possible places prematurely: you may want to execute the same program in different environments. Yet regardless of the exact number of places, there are certain operations you will always want to perform, such as iterating over all places, or checking when two places are equal.

We therefore choose to abstract over the *number* of places in the module we will define in the coming section. Agda allows modules to be parametrised as follows:

```
module DistrArray (placeCount : Nat) where
```

When we import the *DistrArray* module, we are obliged to choose the number of places. Typically, there will one place for every available processor. From this number, we can define a data type corresponding to the available places:

```
Place : ⋆
Place = Fin placeCount
```

The key idea underlying our model of locality-aware algorithms is to index computations by the place where they are executed. The new type declaration for the *IO* monad corresponding to operations on *distributed* arrays will become:

```
data DIO (a : ⋆) : Shape → Place → Shape → ⋆ where
  ...
```

You may want to think of a value of type *DIO a ns p ms* as a computation that can be executed at place *p* and will take a heap of shape *ns* to a heap of shape *ms*, yielding a final value of type *a*.

We strive to ensure that any well-typed program written in the *DIO* monad will never access data that is not local. The specification of distributed arrays now poses a twofold problem: we want to ensure that the array manipulations from the previous section are 'locality-aware,' that is, we must somehow restrict the array indices that can be accessed from a certain place; furthermore, X10 facilitates several *place-shifting* operations that change the place where certain chunks of code are executed. As we shall see in the rest of this section, both these issues can be resolved quite naturally.

Regions, Points, and Distributed Arrays

Before we define the *DIO* monad, we need to introduce several new concepts. In what follows, we will try to stick closely to X10's terminology for distributed

arrays. Every array is said to have a *region* associated with it. A region is a set of valid index points. A *distribution* specifies a place for every index point in a region.

Once again, we will only treat flat arrays storing natural numbers and defer any discussion about how to deal with more complicated data structures for the moment. In this simple case, a region merely determines the size of the array.

```
Region : ⋆
Region = Nat
```

As we have seen in the previous section, we can model array indices using the *Fin* data type:

```
Point : Region → ⋆
Point n = Fin n
```

To model distributed arrays, we now need to consider the distribution that specifies *where* this data is stored. In line with existing work [10], we assume the existence of a fixed distribution. Agda's **postulate** expression allows us to assume the existence of a distribution, without providing its definition.

```
postulate
  distr : forall {n ns} → Loc n ns → Point n → Place
```

At the end of this section, we will discuss how programmers can define their own distributions.

Now that we have all the required auxiliary data types, we proceed by defining the *DIO* monad.

```
data DIO (a : ⋆) : Shape → Place → Shape → ⋆ where
```

As it is a bit more complex than the data types we have seen so far, we will discuss every constructor individually.

The *Return* constructor is analogous to one we have seen previously for the *IO* monad: it lifts any pure value into the *DIO* monad.

```
Return : {p : Place} → {ns : Shape} → a → DIO a ns p ns
```

The *Read* and *Write* operations are more interesting. Although they correspond closely to the operations we have seen in the previous section, their type now keeps track of the place where they are executed. Any read or write operation to point *pt* of an array *l* can *only* be executed at the place specified by the distribution. This invariant is enforced by the types of our constructors:

```
Read : forall {n ns ms} →
  (l : Loc n ns) → (pt : Point n) →
  (Data → DIO a ns (distr l pt) ms) →
  DIO a ns (distr l pt) ms
```

$$
\begin{aligned}
&\mathit{Write} : \mathbf{forall}\ \{n\ ns\ ms\} \to \\
&\quad (l : \mathit{Loc}\ n\ ns) \to (pt : \mathit{Point}\ n) \to \mathit{Data} \to \\
&\quad \mathit{DIO}\ a\ ns\ (\mathit{distr}\ l\ pt)\ ms \to \\
&\quad \mathit{DIO}\ a\ ns\ (\mathit{distr}\ l\ pt)\ ms
\end{aligned}
$$

In contrast to *Read* and $Write$, new arrays can be allocated at any place.

$$
\begin{aligned}
&\mathit{New} : \mathbf{forall}\ \{p\ ns\ ms\} \to \\
&\quad (n : \mathit{Nat}) \to \\
&\quad (\mathit{Loc}\ n\ (\mathit{Cons}\ n\ ns) \to \mathit{DIO}\ a\ (\mathit{Cons}\ n\ ns)\ p\ ms) \to \\
&\quad \mathit{DIO}\ a\ ns\ p\ ms
\end{aligned}
$$

Finally, we add a constructor for the place-shifting operator At:

$$
\begin{aligned}
&\mathit{At} : \mathbf{forall}\ \{p\ ns\ ms\ ps\} \to \\
&\quad (q : \mathit{Place}) \to \mathit{DIO}\ ()\ ns\ q\ ms \to \mathit{DIO}\ a\ ms\ p\ ps \to \mathit{DIO}\ a\ ns\ p\ ps
\end{aligned}
$$

The At operator lets us execute a computation at another place. As we will discard the result of this computation, we require it to return an element of the unit type.

We can add our smart constructors for each these operations, as we have done in the previous section. We can also show that *DIO* is indeed a parameterised monad. We have omitted the definitions of the *return* and bind operators for the sake of brevity:

$$
\begin{aligned}
&\mathit{return} : \mathbf{forall}\ \{ns\ a\ p\} \to a \to \mathit{DIO}\ a\ ns\ p\ ns \\
&\_ \gg\!= \_ : \mathbf{forall}\ \{ns\ ms\ ks\ a\ b\ p\} \to \\
&\quad \mathit{DIO}\ A\ ns\ p\ ms \to (A \to \mathit{DIO}\ B\ ms\ p\ ks) \to \mathit{DIO}\ B\ ns\ p\ ks
\end{aligned}
$$

It is worth noting that the bind operator $\gg\!=$ can only be used to sequence operations at the same place.

We can use our primitives to define more interesting functions over distributed arrays. The distributed map, for example, applies a function to all the elements of a distributed array at the place where they are stored.

$$
\begin{aligned}
&\mathit{for} : \mathbf{forall}\ \{n\ ns\ p\} \to (\mathit{Point}\ n \to \mathit{DIO}\ ()\ ns\ p\ ns) \to \mathit{DIO}\ ()\ ns\ p\ ns \\
&\mathit{for}\ \{\mathit{Succ}\ k\}\ dio = dio\ \mathit{Fz} \gg (\mathit{for}\ \{k\}\ (dio\ .\ \mathit{Fs})) \\
&\mathit{for}\ \{\mathit{Zero}\}\ dio \quad = \mathit{return}\ () \\
&\mathit{dmap} : \mathbf{forall}\ \{n\ ns\ p\} \to (\mathit{Data} \to \mathit{Data}) \to \mathit{Loc}\ n\ ns \to \mathit{DIO}\ ()\ ns\ p\ ns \\
&\mathit{dmap}\ f\ l = \mathit{for}\ (\lambda i \to \mathit{at}\ (\mathit{distr}\ l\ i)\ (\mathit{readArray}\ l\ i \gg\!= \lambda x \to \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{writeArray}\ l\ i\ (f\ x)))
\end{aligned}
$$

We use the *for* function defined above to iterate over the indices of an array. The *dmap* function illustrates how these primitives we have presented can be used to write distributed algorithms.

Denotational model

To run a computation in the *DIO* monad, we stick quite closely to the *run* function defined in the previous section. Our new *run* function, however, must be locality-aware. To that end, we parameterise the *run* function explicitly by the place where the computation is executed.

$$
\begin{aligned}
&\mathit{run} : \mathbf{forall}\ \{\, a\ \mathit{ns}\ \mathit{ms} \,\} \rightarrow \\
&\quad (p : \mathit{Place}) \rightarrow \mathit{DIO}\ a\ \mathit{ns}\ p\ \mathit{ms} \rightarrow \mathit{Heap}\ \mathit{ns} \rightarrow \mathit{Pair}\ a\ (\mathit{Heap}\ \mathit{ms}) \\
&\mathit{run}\ p\ (\mathit{Return}\ x)\ h &&= \mathit{pair}\ x\ h \\
&\mathit{run}\ .(\mathit{distr}\ l\ i)\ (\mathit{Read}\ l\ i\ \mathit{rd})\ h &&= \mathit{run}\ (\mathit{distr}\ l\ i)\ (\mathit{rd}\ (\mathit{lookup}\ l\ i\ h))\ h \\
&\mathit{run}\ .(\mathit{distr}\ l\ i)\ (\mathit{Write}\ l\ i\ x\ \mathit{wr})\ h &&= \mathbf{let}\ h' = \mathit{updateHeap}\ l\ i\ x\ h \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathbf{in}\ \mathit{run}\ (\mathit{distr}\ l\ i)\ \mathit{wr}\ h' \\
&\mathit{run}\ p\ (\mathit{New}\ n\ \mathit{io})\ h &&= \mathit{run}\ p\ (\mathit{io}\ \mathit{Top})\ (\mathit{Alloc}\ (\lambda i \rightarrow \mathit{Zero})\ h) \\
&\mathit{run}\ p\ (\mathit{At}\ q\ \mathit{io1}\ \mathit{io2})\ h &&= \mathit{run}\ p\ \mathit{io2}\ (\mathit{snd}\ (\mathit{run}\ q\ \mathit{io1}\ h))
\end{aligned}
$$

Now we can see that the *Read* and *Write* operations may not be executed at *any* place. Recall that the *Read* and *Write* constructors both return computations at the place *distr l i*. When we pattern match on a *Read* or *Write*, we know exactly what the place argument of the *run* function must be. Correspondingly, we do not pattern match on the place argument—we know that the place can only be *distr l i*. Agda's syntax allows us to prefix expressions by a single period, provided we know that there is only one possible value an argument may take. This may be unfamiliar to many functional programmers who are used to thinking of patterns being built-up from variables and constructors—*distr l i* is an expression, not a pattern! The situation is somewhat similar to pattern matching on GADTs in Haskell, which introduces equalities between *types*. The *DIO* monad, however, is indexed by values. As a result, pattern matching in the presence of dependent types may introduce equalities between *values*.

The other difference with respect to the previous *run* function, is the new case for the *At* constructor. In that case, we sequence the two computations *io1* and *io2*. To do so, we first execute the *io1* at *q*, but discard its result; we continue executing the second computation *io2* with the heap resulting from the execution of *io1* at the location *p*. Conform to previous proposals [**?**], we have assumed that *io1* and *io2* are performed synchronously—executing *io1* before continuing with the rest of the computation. Using techniques to model concurrency that we have presented previously [19], we believe we could give a more refined treatment of the X10's *globally asynchronous/locally synchronous* semantics and provide specifications for X10's clocks, *finish*, and *force* constructs.

Other operations

Using the place-shifting operator *at*, we can define several other operations to manipulate places and regions. With our first-class distribution and definition of *Place*, we believe there is no need to define more primitive operations. We show how two new control structures, *forallplaces* and *ateach*, can be defined in terms of the functions we have seen so far.

The *forallplaces* operation executes its argument computation at all available places. We define it using the *for* function to iterate over all places. The *ateach* function, on the other hand, is a generalisation of the distributed map operation. It iterates over an array, executing its argument operation once for every index of the array, at the place where that index is stored.

$$
\begin{aligned}
&\mathit{forallplaces} : \mathbf{forall}\ \{p\ \mathit{ns}\} \rightarrow \\
&\quad ((q : \mathit{Place}) \rightarrow \mathit{DIO}\ ()\ \mathit{ns}\ q\ \mathit{ns}) \rightarrow \mathit{DIO}\ ()\ \mathit{ns}\ p\ \mathit{ns} \\
&\mathit{forallplaces}\ \mathit{io} = \mathit{for}\ (\lambda i \rightarrow \mathit{at}\ i\ (\mathit{io}\ i))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ateach} : \mathbf{forall}\ \{n\ \mathit{ns}\ p\} \rightarrow \\
&\quad (l : \mathit{Loc}\ n\ \mathit{ns}) \rightarrow ((\mathit{pt} : \mathit{Point}\ n) \rightarrow \mathit{DIO}\ ()\ \mathit{ns}\ (\mathit{distr}\ l\ \mathit{pt})\ \mathit{ns}) \rightarrow \\
&\quad \mathit{DIO}\ ()\ \mathit{ns}\ p\ \mathit{ns} \\
&\mathit{ateach}\ l\ \mathit{io} = \mathit{for}\ (\lambda i \rightarrow \mathit{at}\ (\mathit{distr}\ l\ i)\ (\mathit{io}\ i))
\end{aligned}
$$

Defining distributions

Throughout this section we have assumed the existence of a distribution, specifying how an array is distributed over the available places. Several built-in distributions are provided by X10, some of which we will now define. For the moment, we focus on defining a distribution of a single array, that is, functions of type $\mathit{Fin}\ n \rightarrow \mathit{Place}$.

There are two atomic distributions: the constant distribution maps all the points of an array to a single place; the *unique* distribution maps the i-th index of an array to the i-th place.

$$
\begin{aligned}
&\mathit{constDistr} : \{n : \mathit{Nat}\} \rightarrow \mathit{Place} \rightarrow \mathit{Point}\ n \rightarrow \mathit{Place} \\
&\mathit{constDistr}\ p = \lambda i \rightarrow p
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unique} : \mathit{Point}\ \mathit{placeCount} \rightarrow \mathit{Place} \\
&\mathit{unique}\ i = i
\end{aligned}
$$

We can compose any two such distributions, to form a larger distribution:

$$
\begin{aligned}
&\mathit{compose} : \mathbf{forall}\ \{n\ m\} \rightarrow \\
&\quad (\mathit{Point}\ n \rightarrow \mathit{Place}) \rightarrow (\mathit{Point}\ m \rightarrow \mathit{Place}) \rightarrow (\mathit{Point}\ (n + m) \rightarrow \mathit{Place})
\end{aligned}
$$

Although the definition of *compose* is a bit tricky, there is a particularly elegant definition in the literature using *views* [**?**]. Of course, we can iterate the *compose* operator:

$$
\begin{aligned}
&\mathit{cycle} : \{n : \mathit{Nat}\} \rightarrow \\
&\quad (k : \mathit{Nat}) \rightarrow (\mathit{Point}\ n \rightarrow \mathit{Place}) \rightarrow (\mathit{Point}\ (k * n) \rightarrow \mathit{Place}) \\
&\mathit{cycle}\ (\mathit{Succ}\ k)\ f\ i = \mathit{compose}\ f\ (\mathit{cycle}\ k\ f)\ i
\end{aligned}
$$

In the case where k is equal to *Zero*, we need to define a function of type $\mathit{Fin}\ \mathit{Zero} \rightarrow \mathit{Place}$. As $\mathit{Fin}\ \mathit{Zero}$ has no inhabitants, this function will never be applied and we may omit the definition accordingly.

Using similar combinators, we can define distributions over all arrays, exploiting the obvious similarity between *Loc* and *Fin*. There are several other distributions supported by X10, that can be implemented along the same lines.

5 Discussion

Using a dependently-typed host language, we have shown how to implemented a domain-specific library for distributed arrays, together with an embedded type system that guarantees all array access operations are both safe and local. We have provided semantics for our library in the form of a total, functional specification. Although our semantics may not take the form of deduction rules, they are no less precise or concise. Besides these functional specifications are both executable and amenable to computer-aided formal verification. More generally, we hope that this approach can be extended to other domains: a dependently-typed language accommodates domain specific libraries with their own embedded type systems.

Having said this, there are clearly several serious limitations of this work as it stands. First and foremost, we have assumed that every array only stores natural numbers, disallowing more complex structures such as multidimensional arrays. This can be easily fixed by defining a more elaborate *Shape* data type. In its most general form, we could choose our *Shape* data type as a list of types; a heap then corresponds to a list of values of the right type.[2] We decided to restrict ourself to this more simple case for the purpose of presentation. We also believe that there is no fundamental obstacle preventing us from incorporating the rich region calculus offered by X10 in the same fashion.

Secondly, we have not discussed how code in the *IO* or *DIO* monad is actually compiled. To actually exploit the locality information our types carry, we would need to customise the existing Agda to Haskell compiler. The ongoing effort to support data parallelism in Haskell [5, 6] could hopefully provide us with a most welcome foothold.

Furthermore, we have assumed a fixed distribution. We would like to investigate a more flexible approach, allowing distributions to be associated with arrays as they are created, rather than require a global distribution to be chosen before a program is executed.

Finally, there are many features of X10 that we have not discussed here at all. Most notably, we have refrained from modelling many of X10's constructs that enable asynchronous communication between locations, even though we would like to do so in the future. Nonetheless, we believe this paper demonstrates the viability of our approach and provides a first stepping-stone for such further research.

Acknowledgements We wish to express our gratitude to Jens Palsberg for our interesting discussions; to Ulf Norell for his fantastic new incarnation of Agda; and to Mauro Jaskelioff, Nicolas Oury, and Liyang HU for their helpful comments on a draft version of this paper.

[2] There are some technical details involving 'size problems' that are beyond the scope of this paper. The standard technique of introducing a universe, closed under natural numbers and arrays, should resolve these issues.

References

1. Agda 2. `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php`.
2. Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification. Technical report, Sun Microsystems, Inc., 2005.
3. Thorsten Altenkirch, Conor McBride, and Peter Morris. Generic programming with dependent types. Draft lecture notes for the Summer School ond Generic Programming, August 2006.
4. Robert Atkey. Parameterised notions of computation. In *Proceedings of the Workshop on Mathematically Structured Functional Programming*, 2006.
5. Manuel M.T. Chakravarty, Gabriele Keller, Roman Lechtchinsky, and Wolf Pfannenstiel. Nepal – Nested Data-Parallelism in Haskell. In *Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference*, volume LNCS 2150, 2001.
6. Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: a status report. *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, 2007.
7. Brad Chamberlain, Steve Deitz, Mary Beth Hribar, and Wayne Wong. Chapel. Technical report, Cray Inc., 2005.
8. Bradford L. Chamberlain, Sung-Eun Choi, E. Christopher Lewis, Calvin Lin, Lawrence Snyder, and Derrick Weathersby. ZPL: A machine independent programming language for parallel computers. *Software Engineering*, 26(3):197–211, 2000.
9. Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005.
10. Christian Grothoff, Jens Palsberg, and Vijay Saraswat. Safe arrays via regions and dependent types. Submitted for publication.
11. Paul N. Hilfinger, Dan Oscar Bonachea, Kaushik Datta, David Gay, Susan L. Graham, Benjamin Robert Liblit, Geoffrey Pike, Jimmy Zhigang Su, and Katherine A. Yelick. Titanium language reference manual, version 2.19. Technical Report UCB/EECS-2005-15, EECS Department, University of California, Berkeley, Nov 2005.
12. Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.
13. Andres Löh. lhs2tex. `http://people.cs.uu.nl/andres/lhs2tex/`.
14. James McKinna and Joel Wright. A type-correct, stack-safe, provably correct, expression compiler in Epigram. Submitted to the Journal of Functional Programming, 2006.
15. Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.
16. Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.
17. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.
18. Herb Sutter. The free lunch is over. *Dr. Dobb's Journal*, 30(3), 2005.
19. Wouter Swierstra and Thorsten Altenkirch. Beauty in the beast: a functional semantics of the awkward squad. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, 2007.