Journal of Higher Order and Symbolic Computing manuscript No.
(will be inserted by the editor)

More Dependent Types for Distributed Arrays

Wouter Swierstra

the date of receipt and acceptance should be inserted later

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. This paper shows how a dependently-typed programming language can aid in the development of these algorithms and statically ensure that 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 presented here facilitates the testing of and reasoning about algorithms that manipulate 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 performance gain that multicore processors offer [1,6–8,12,21]. 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 designed specifically for distributed computing, such as X10 [8], require all arrays operations to be local. Any attempt to access non-local

Wouter Swierstra
Vector Fabrics
Paradijslaan 28, Eindhoven
E-mail: wouter@vectorfabrics.com

data results in an exception. To preclude such errors, X10's type system guarantees that programs only access local parts of a distributed array [9,17]. The proposed type system is fairly intricate and consists of a substantial number of type rules that keep track of locality information. Proving meta-theoretical results about such a system, such as the decidability of type checking, is no trivial result.

This paper explores 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 meta-theoretical results. Instead, this paper shows how to tailor a powerful type system to enforce certain properties—resulting in a *domain-specific embedded type system*. This type system immediately inherits all the desirable properties of our dependently-typed host type system, such as subject reduction, decidable type checking, and principle typing. Functional programmers have studied domain-specific embedded languages for years [11]; the time is ripe to take these ideas one step further.

Previous work described a pure specification of several parts of the IO monad [23], the interface between pure functional languages such as Haskell [19] and the 'real world.' By providing functional, executable specifications you can test, debug, and reason about impure programs as if they were pure. When you release the final version of your code, you can replace these pure specifications with their impure, more efficient, counterparts. In the presence of dependent types, these specifications can provide even stronger static guarantees about our programs. To this end, this paper makes several contributions:

- We will begin by giving a pure specification of arrays (Section 3). This 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. To make this specification usable, we must overcome a problem related to the *weakening* of address locations.
- Distributed arrays pose more of a challenge. Section 4 shows how to enforce locality constraints, while still providing programmers with place-shifting operators. The pure specification 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 will see how programmers may write their own locality-aware control structures (Section 4.3), how to implement simple distributed algorithms using these control structures (Section 4.4), and how to define combinators that describe data distributions (Section 4.5). We will conclude by discussing further work and the limitations of this approach (Section 5).

Throughout this paper, I will use the dependently-typed programming language Agda [16] 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. In the coming section, I will briefly introduce the syntax of Agda 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 [20]. For example, consider the following definition of the natural numbers.

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

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:

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

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

Polymorphic lists are slightly more interesting than natural numbers:

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

To uniformly parameterise a data type, we can write additional arguments to the left of the colon. 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 of kind $\star$.

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

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

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 corresponds to 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 could also define *append* as in any other functional language:

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

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 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 automatically instantiated in Haskell. By enclosing a function's argument in curly brackets, we can explicitly instantiate or pattern match on an implicit argument.

Besides polymorphic data types, Agda also supports *indexed families*, a dependently typed analogue of Haskell's GADTs. 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:

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

The type *Fin n* corresponds to a finite type with n distinct values. For example, *Fin* 1 is isomorphic to the unit type; *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 *Fin* 0 is uninhabited. For every n, the *Fs* constructor embeds *Fin n* into *Fin* (*Succ n*); the *Fz* constructor, on the other hand, adds a single new element to *Fin* (*Succ n*) that was not in *Fin n*. This inductive argument shows that *Fin n* does indeed have n elements.

Agda has many other features, such as records and a 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, 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. To program with these operators, we need to resolve a few technical problems. Finally, we will 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 using a universe construction, as is done in my PhD thesis [22].

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.

$$
\begin{aligned}
&\mathit{Data} : \star\\
&\mathit{Data} = \mathit{Nat}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{Array} : \mathit{Nat} \to \star\\
&\mathit{Array}\ n = \mathit{Fin}\ n \to \mathit{Data}
\end{aligned}
$$

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 explicitly state 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:

$$
\begin{aligned}
&\mathit{Shape} : \star\\
&\mathit{Shape} = \mathit{List}\ \mathit{Nat}
\end{aligned}
$$

The *Shape* of the heap is 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 layout $\mathit{Cons}\ n\ ns$.

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Heap} : \mathit{Shape} \to \star\ \mathbf{where}\\
&\quad \mathit{Empty} : \mathit{Heap}\ \mathit{Nil}\\
&\quad \mathit{Alloc} : \{n : \mathit{Nat}\} \to \{ns : \mathit{Shape}\} \to \mathit{Array}\ n \to \mathit{Heap}\ ns \to \mathit{Heap}\ (\mathit{Cons}\ n\ ns)
\end{aligned}
$$

Finally, we will want to model references, denoting locations in the heap. A value of type $\mathit{Loc}\ n\ ns$ 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; any existing reference can be modified to denote the same location in a larger heap using the *Pop* constructor.

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Loc} : \mathit{Nat} \to \mathit{Shape} \to \star\ \mathbf{where}\\
&\quad \mathit{Top} : \{n : \mathit{Nat}\} \to \{ns : \mathit{Shape}\} \to \mathit{Loc}\ n\ (\mathit{Cons}\ n\ ns)\\
&\quad \mathit{Pop} :\ \mathbf{forall}\ \ \{n\ k\ ns\} \to \mathit{Loc}\ n\ ns \to \mathit{Loc}\ n\ (\mathit{Cons}\ k\ ns)
\end{aligned}
$$

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. When we use the **forall**-notation, the types of n, k, and ns are inferred from the rest of the signature by the Agda type checker. Alternatively, we could also have written the more verbose:

$$
\begin{aligned}
&\mathit{Pop} : \{n : \mathit{Nat}\} \to \{k : \mathit{Nat}\} \to \{ns : \mathit{Shape}\} \to\\
&\quad \mathit{Loc}\ n\ ns \to \mathit{Loc}\ n\ (\mathit{Cons}\ k\ ns)
\end{aligned}
$$

We will occasionally use the **forall**-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 an 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* [3]—that is, a monad with *return* and bind operators that satisfy certain coherence conditions with respect to the *Shape* indices.

```
return : forall {a ns} → 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, ≫=, 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 enforced by the type of the bind operator.

To facilitate programming using these array operations, we could define the functions *readArray* and *newArray* as follows:

```
readArray : forall {n ns} → Loc n ns → Fin n → IO Data ns ns
readArray a i = Read a i Return
newArray : forall {ns} → (n : Nat) → IO (Loc n (Cons n ns)) ns (Cons n ns)
newArray n = New n Return
```

There is a slight problem with these definitions. 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. This problem is best illustrated with an example.

Consider the following sequence of array manipulations:

$$\begin{array}{l} \mathit{newArray}\ 4 \ggg\!= \lambda \mathit{array1} \rightarrow \\ \mathit{newArray}\ 8 \ggg\!= \lambda \mathit{array2} \rightarrow \\ \mathit{readArray}\ \mathit{array1}\ \mathit{Fz} \end{array}$$

This allocates two arrays, before performing reading an index from the first array. The first array, stored at *array1*, has type *Loc* 4 (*Cons* 4 *Nil*); the second array has type *Loc* 8 (*Cons* 8 (*Cons* 4 *Nil*)). When we want to read from the first array after the second allocation, the type checker expects an array location in a heap of shape (*Cons* 8 (*Cons* 4 *Nil*), and not *Cons* 4 *Nil*. Fortunately, adding a *Pop* constructor around *array1* yields a type correct program:

$$\begin{array}{l} \mathit{newArray}\ 4 \ggg\!= \lambda \mathit{array}_1 \rightarrow \\ \mathit{newArray}\ 8 \ggg\!= \lambda \mathit{array}_2 \rightarrow \\ \mathit{readArray}\ (\mathit{Pop}\ \mathit{array}_1)\ \mathit{Fz} \end{array}$$

We will refer to this as *weakening* the reference array_1, in line with the usual terminology used for the weakening rules present in many logics.

Adding these extra *Pop* constructors by hand is tiresome and error-prone. In the coming sections, we will see how this process can be automated to some degree.

3.1 Manual weakening

Before we revisit our smart constructors, we need to develop a bit of machinery. We need to decide how many *Pop* constructors are necessary to weaken a location to denote a position in a larger heap. Put differently, we need to prove that one shape is a suffix of a larger shape. One way to represent such a proof is using an inductive data type:

$$\begin{array}{l} \mathbf{data}\ \mathit{IsSuffix} : \mathit{Shape} \rightarrow \mathit{Shape} \rightarrow \star\ \mathbf{where} \\ \quad \mathit{Base} : \mathbf{forall}\ \{\mathit{ns}\} \rightarrow \mathit{IsSuffix}\ \mathit{ns}\ \mathit{ns} \\ \quad \mathit{Step} : \mathbf{forall}\ \{m\ \mathit{ns}\ \mathit{ms}\} \rightarrow \mathit{IsSuffix}\ \mathit{ns}\ \mathit{ms} \rightarrow \mathit{IsSuffix}\ \mathit{ns}\ (\mathit{Cons}\ m\ \mathit{ms}) \end{array}$$

Every proof that *ns* is a suffix of *ms* is built from two constructors *Base* and *Step*. The base case corresponds to stating that every list is a suffix of itself; the step case states that if *ns* is a suffix of *ms* then *ns* is also a suffix of *Cons m ms* for any natural number *m*.

If we have such a proof that *ns* is a suffix of *ms*, we can weaken any location of type *Loc n ns* to make a new location of type *Loc n ms*. The weaken function does exactly this:

$$\begin{array}{l} \mathit{weaken} : \mathbf{forall}\ \{n\ \mathit{ns}\ \mathit{ms}\} \rightarrow \mathit{IsSuffix}\ \mathit{ns}\ \mathit{ms} \rightarrow \mathit{Loc}\ n\ \mathit{ns} \rightarrow \mathit{Loc}\ n\ \mathit{ms} \\ \mathit{weaken}\ \mathit{Base}\ \mathit{loc} \qquad = \mathit{loc} \\ \mathit{weaken}\ (\mathit{Step}\ i)\ \mathit{loc} = \mathit{Pop}\ (\mathit{weaken}\ i\ \mathit{loc}) \end{array}$$

It proceeds by induction over the proof argument, adding a *Pop* constructor for every step. While the *weaken* function adds the necessary *Pop* constructors, there is still no way to compute its proof argument automatically. There is an alternative representation of such proofs that does facilitate such automation.

3.2 An alternative proof

In type theory, one way to formulate the proposition that two values are equal is by constructing an inhabitant of the following type:

$$\begin{array}{l}\textbf{data}\ \_\equiv\_\ \{a:\star\}\ (x:a) : a \to \star\ \textbf{where}\\ \quad \mathit{Refl} : x \equiv x\end{array}$$

That is, the only canonical proof that two objects are equal is by reflexivity. This type plays a fundamental role in intensional type theory [15].

Whenever we pattern match on an equality proof, we learn how two values are related. For example, suppose we want to write the following function:

$$\mathit{subst} : \{a:\star\} \to (P : a \to \star) \to (x : a) \to (y : a) \to x \equiv y \to P\ x \to P\ y$$

What patterns should we write on the left-hand side of the definition? Clearly, any argument of type $x \equiv y$ must be *Refl*. As soon as we match on that argument, however, we learn something about x and y, i.e., they must be the same. We will write this as follows:

$$\mathit{subst}\ P\ x\ .x\ \mathit{Refl}\ \mathit{px} = \ldots$$

The pattern $.x$ means 'the value of this argument can only be equal to x.' Such 'dot-patterns' appear naturally once you start programming with data types indexed by values.

We can decide whenever two shapes are equal, that is, we can define a function with the following type:

$$\mathit{decEqShape} : (s : \mathit{Shape}) \to (t : \mathit{Shape}) \to \mathit{Either}\ (s \equiv t)\ ((s \equiv t) \to \bot)$$

Here *Either* denotes the disjoint sum of two types and $\bot$ is the empty type. The definition of *decEqShape* is fairly unremarkable. We traverse the lists s and t, comparing every element. If the elements of both lists are equal at every position, the entire lists are equal; if the lists have different lengths or store different elements, the two shapes are distinct.

Using this function, we can define the following function that checks whether or not one list is a suffix of another:

$$\begin{array}{l}\_\leqslant\_ : \mathit{Shape} \to \mathit{Shape} \to \mathit{Bool}\\ \mathit{Nil} \leqslant s \qquad\qquad = \mathit{True}\\ \mathit{Cons}\ u\ s \leqslant \mathit{Nil} = \mathit{False}\\ \mathit{Cons}\ u\ s \leqslant \mathit{Cons}\ v\ t\ \textbf{with}\ \mathit{decEqShape}\ (\mathit{Cons}\ u\ s)\ (\mathit{Cons}\ v\ t)\\ \mathit{Cons}\ u\ s \leqslant \mathit{Cons}\ .u\ .s \mid \mathit{Inl}\ \mathit{Refl} = \mathit{True}\\ \mathit{Cons}\ u\ s \leqslant \mathit{Cons}\ v\ t \ \ \mid \mathit{Inr}\ q = (\mathit{Cons}\ u\ s) \leqslant t\end{array}$$

Here we use a **with** clause, a local pattern match similar to Haskell's **case** expressions. In the inductive step, we make a call to the auxiliary function *decEqShape*. On the next lines we repeat the left-hand side of the function definition, followed by the new patterns *Inl Refl* and *Inr q*. We separate the new patterns introduced by the **with** clause from the original function arguments by a vertical bar. If we learn that the two lists are equal, it follows that the first list is a suffix of the second. Otherwise, we continue with a recursive call.

Now we can reflect any *Bool* into $\star$ as follows:

$$\begin{aligned}&\mathit{So} : \mathit{Bool} \to \star\\ &\mathit{So}\ \mathit{True}\ = ()\\ &\mathit{So}\ \mathit{False} = \bot\end{aligned}$$

Using this definition, we give an alternative definition of the *IsSuffix* predicate: a shape s is a suffix of a shape t if and only if $\mathit{So}\ (s \leqslant t)$ is inhabited. This alternative definition is equivalent to the definition using *IsSuffix*. Proving the equivalence in one direction corresponds to defining a function with the following type:

$$\mathit{equiv} : (s : \mathit{Shape}) \to (t : \mathit{Shape}) \to \mathit{So}\ (s \leqslant t) \to \mathit{IsSuffix}\ s\ t$$

The definition is reasonably straightforward: we pattern match on s and t, using the proof argument to kill off any impossible branches.

So why go through all this effort to write an equivalent representation of the inductive *IsSuffix* data type? The two forms of proof are useful for different reasons:

IsSuffix $s\ t$ – By defining the proof as an inductive data type, we can pattern match on proofs. We use this to define the *weaken* function. Unfortunately, we need to write an inhabitant of *IsSuffix* by hand to pass to the weaken function.

So $(s \leqslant t)$ – By defining the $\leqslant$-operator we have written a function that decides when one list is a suffix of another. In particular, for any pair of *closed* shapes $s \leqslant t$ reduces to either *True* or *False*. Correspondingly, the type *So* $(s \leqslant t)$ is either trivial or uninhabited.

The central idea behind our smart constructors is to weaken using the *inductive representation*, but to require a *trivial implicit witness*.

3.3 Smart constructors

With this in mind, we revise our smart constructors for reading from and writing to references. These smart constructors now require an implicit proof that s is a suffix of t. Using our *equiv* function, we can compute the inductive representation of such a proof. Using this inductive representation, we can weaken locations as necessary.

$$\begin{aligned}&\mathit{readArray} : \ \textbf{forall}\ \ \{n\ ns\ ms\} \to \{p : \mathit{So}\ (ns \leqslant ms)\} \to\\ &\quad \mathit{Loc}\ n\ ns \to \mathit{Fin}\ n \to \mathit{IO}\ \mathit{Data}\ ms\ ms\\ &\mathit{readArray}\ \{n\}\ \{ns\}\ \{ms\}\ \{p\}\ a\ i = \mathit{Read}\ (\mathit{weaken}\ (\mathit{equiv}\ ns\ ms\ p)\ a)\ i\ \mathit{Return}\end{aligned}$$

The beauty of this solution is that Agda will automatically instantiate implicit arguments of type (). In other words, for any closed *IO* term a programmer need not worry about passing proof arguments. For example, we can now write:

$$\begin{aligned}&f : \mathit{IO}\ \mathit{Nat}\ \mathit{Nil}\ (\mathit{Cons}\ 8\ (\mathit{Cons}\ 4\ \mathit{Nil}))\\ &f = \mathit{newArray}\ 4 \ggg\!= \lambda \mathit{array}_1 \to\\ &\qquad \mathit{newArray}\ 8 \ggg\!= \lambda \mathit{array}_2 \to\\ &\qquad \mathit{readArray}\ \mathit{array}_1\ \mathit{Fz}\end{aligned}$$

The smart constructors will now automatically weaken array_1 after array_2 has been allocated.

Unfortunately, not all in the garden is rosy. If we have an open term, Agda will warn you that it cannot find a suitable proof argument. For example, consider the following function that reads the first and only location of any array storing a single element:

$readOne : \{ns : Shape\} \rightarrow Loc\ 1\ ns \rightarrow IO\ Nat\ ns\ ns$
$readOne\ a = readArray\ a\ Fz$

While any particular call to *readOne* is safe, Agda fails to automatically deduce that $So\ (ns \leqslant ns)$ reduces to the unit type. Even though it is clearly true for any particular choice of *ns* and we can fill in the proof argument manually, the techniques we have outlined here fail to provide the required proof automatically.

Without more language support it is unlikely that we can avoid this restriction.

3.4 Denotational model

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

$Array : Nat \rightarrow \star$
$Array\ n = Fin\ n \rightarrow Data$

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

$lookup : \textbf{forall}\ \{n\ ns\} \rightarrow Loc\ n\ ns \rightarrow Fin\ n \rightarrow Heap\ ns \rightarrow Data$
$lookup\ Top\qquad i\ (Alloc\ a\ \_) = a\ i$
$lookup\ (Pop\ k)\ i\ (Alloc\ \_\ h) = lookup\ k\ i\ h$

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 $Fin\ n \rightarrow Data$; 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.

$updateArray : \{n : Nat\} \rightarrow Fin\ n \rightarrow Data \rightarrow Array\ n \rightarrow Array\ n$
$updateArray\ i\ d\ a = \lambda j \rightarrow \textbf{if}\ i \equiv j\ \textbf{then}\ d\ \textbf{else}\ a\ j$

$updateHeap : \textbf{forall}\ \{n\ ns\} \rightarrow$
$\quad Loc\ n\ ns \rightarrow Fin\ n \rightarrow Data \rightarrow Heap\ ns \rightarrow Heap\ ns$
$updateHeap\ Top\qquad i\ x\ (Alloc\ a\ h) = Alloc\ (updateArray\ i\ x\ a)\ h$
$updateHeap\ (Pop\ k)\ i\ x\ (Alloc\ a\ h) = Alloc\ a\ (updateHeap\ k\ i\ x\ h)$

The *updateArray* function overwrites the data stored at a single index. The function *updateHeap* 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.

We now 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 will add additional *Pop* constructors when new memory is allocated.

We refer to this specification as a denotational model. Agda is a programming language based on a consistent type theory. In a sense, the *run* function constitutes a denotational semantics of mutable arrays. By implementing these semantics in Agda, we build an executable denotational model in Agda's type theory.

3.5 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 ≫= λval_i →
             readArray a j ≫= λval_j →
             writeArray a i val_j ≫
             writeArray a j val_i
```

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 : forall {n ns} →
  (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 parameterised:

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

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

$Place : \star$
$Place = Fin\ placeCount$

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* $(a : \star) : Shape \rightarrow Place \rightarrow Shape \rightarrow \star$ **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.

4.1 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 [9], 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
```

We have implemented several of X10's combinators for defining distributions, which we will present shortly.

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

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

```
New : forall {p ns ms} →
  (n : Nat) →
  (Loc n (Cons n ns) → DIO a (Cons n ns) p ms) →
  DIO a ns p ms
```

Finally, we add a constructor for a place-shifting operator. Using this *At* operator lets us execute a computation at another place.

$$\begin{array}{l} \mathit{At} : \textbf{forall}\ \{p\ ns\ ms\ ps\} \rightarrow \\ \quad (q : \mathit{Place}) \rightarrow \mathit{DIO}\ ()\ ns\ q\ ms \rightarrow \mathit{DIO}\ a\ ms\ p\ ps \rightarrow \mathit{DIO}\ a\ ns\ p\ ps \end{array}$$

Note that we will discard the result of the computation that is executed at another place. We therefore require this computation 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:

$$\begin{array}{l} \mathit{return} : \textbf{forall}\ \{ns\ a\ p\} \rightarrow a \rightarrow \mathit{DIO}\ a\ ns\ p\ ns \\ \_ \gg\!= \_ : \textbf{forall}\ \{ns\ ms\ ks\ a\ b\ p\} \rightarrow \\ \quad \mathit{DIO}\ A\ ns\ p\ ms \rightarrow (A \rightarrow \mathit{DIO}\ B\ ms\ p\ ks) \rightarrow \mathit{DIO}\ B\ ns\ p\ ks \end{array}$$

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

4.2 Denotational model

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

$$\begin{array}{ll} \multicolumn{2}{l}{\mathit{run} : \textbf{forall}\ \{a\ ns\ ms\} \rightarrow} \\ \multicolumn{2}{l}{\quad (p : \mathit{Place}) \rightarrow \mathit{DIO}\ a\ ns\ p\ ms \rightarrow \mathit{Heap}\ ns \rightarrow \mathit{Pair}\ a\ (\mathit{Heap}\ ms)} \\ \mathit{run}\ p\ (\mathit{Return}\ x)\ h & = \mathit{pair}\ x\ h \\ \mathit{run}\ .(\mathit{distr}\ l\ i)\ (\mathit{Read}\ l\ i\ \mathit{rd})\ h & = \mathit{run}\ (\mathit{distr}\ l\ i)\ (\mathit{rd}\ (\mathit{lookup}\ l\ i\ h))\ h \\ \mathit{run}\ .(\mathit{distr}\ l\ i)\ (\mathit{Write}\ l\ i\ x\ \mathit{wr})\ h & = \textbf{let}\ h' = \mathit{updateHeap}\ l\ i\ x\ h \\ & \ \textbf{in}\ \mathit{run}\ (\mathit{distr}\ l\ i)\ \mathit{wr}\ h' \\ \mathit{run}\ p\ (\mathit{New}\ n\ \mathit{io})\ h & = \mathit{run}\ p\ (\mathit{io}\ \mathit{Top})\ (\mathit{Alloc}\ (\lambda i \rightarrow \mathit{Zero})\ h) \\ \mathit{run}\ p\ (\mathit{At}\ q\ \mathit{io1}\ \mathit{io2})\ h & = \mathit{run}\ p\ \mathit{io2}\ (\mathit{snd}\ (\mathit{run}\ q\ \mathit{io1}\ h)) \end{array}$$

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*, as is indicated by the dot-pattern.

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 [10], 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 [23], 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.

4.3 Locality-aware combinators

Using the place-shifting operator *at*, we can define several locality-aware control structures. With our first-class distribution and definition of *Place*, we believe there is no need to define more primitive operations.

The distributed map, for example, applies a function to all the elements of a distributed array at the place where they are stored. We define it in terms of an auxiliary function, *for*, that iterates over all the indices of an array:

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

Besides *dmap*, we implement two other combinators: *forallplaces* and *ateach*. 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.

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

4.4 Example

We will now show how to write a simple algorithm that sums all the elements of a distributed array. To do so efficiently, we first locally sum all the values at every place. To compute the total sum of all the elements of the array, we add together all these local sums. In what follows, we will need the following auxiliary function, *increment*:

$$
\begin{aligned}
&\mathit{increment} : \textbf{forall}\ \{n\ \mathit{ns}\ p\} \rightarrow\\
&\quad (l : \mathit{Loc}\ n\ \mathit{ns}) \rightarrow (i : \mathit{Fin}\ n) \rightarrow \mathit{Nat} \rightarrow (\mathit{distr}\ l\ i \equiv p) \rightarrow \mathit{DIO}\ ()\ \mathit{ns}\ p\ \mathit{ns}\\
&\mathit{increment}\ l\ i\ x\ \mathit{Refl} = \mathit{readArray}\ l\ i \gg\!= \lambda y \rightarrow \mathit{writeArray}\ l\ i\ (x + y)
\end{aligned}
$$

Note that *increment* is a bit more general than strictly necessary. We could return a computation at *distr l i*, but instead we choose to be a little more general: *increment* can be executed at any place, as long as we have a proof that this place is equal to *distr l i*. The ≡-type is inhabited by single constructor *Refl*.

We can use the *increment* function to define a simple sequential *sum* function:

```
sum : forall {n ns p} → Loc n ns → Loc 1 ns → DIO () ns p ns
sum l out = ateach l (λi → readArray l i >>= λn →
                                at (distr out Fz) (increment out Fz n Refl))
```

The *sum* function takes an array as its argument, together with a reference to a single-celled array, *out*. It reads every element of the array, and increments *out* accordingly.

Finally, we can use both these functions to define a parallel sum:

```
psum : forall {n ns} →
  (l : Loc n ns) → (localSums : Loc placeCount ns) →
  ((i : Place) → distr localSums i ≡ i) →
  (out : Loc 1 ns) → DIO Nat ns (distr out Fz) ns
psum l localSums locDistr out =
  ateach l (λi → (readArray l i >>= λn →
                    increment localSums (distr l i) n (locDistr (distr l i))))
  >> sum localSums out
  >> readArray out Fz
```

The *psum* function takes four arguments: the array *l* whose elements you would like to sum; an array *localSums* that will store the intermediate sums; an assumption regarding the distribution of this array; and finally, the single-celled array to which we write the result. For every index *i* of the array *l*, we read the value stored at index *i*, and increment the corresponding local sum. We then add together the local sums using our previous sequential *sum* function, and return the final result. We use our assumption about the distribution of the *localSums* array when calling the increment function. Without this assumption, we would have to use the place-shifting operation *at* to update a (potentially) non-local array index.

There are several interesting issues that these examples highlight. First of all, as our *at* function only works on computations returning a unit type, the results of intermediate computations must be collected in intermediate arrays.

More importantly, however, whenever we want to rely on properties of the global distribution, we need to make explicit assumptions in the form of proof arguments. This is rather unfortunate: it would be interesting to research how a specific distribution can be associated with an array when it is created. This would hopefully allow for a more fine-grained treatment of distributions and eliminate the need for explicit proof arguments.

4.5 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 *Fin n → 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.

```
constDistr : {n : Nat} → Place → Point n → Place
constDistr p = λi → p
```

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

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

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

Although the definition of *compose* is a bit tricky, there is a particularly elegant definition in the literature using *views* [2]. Of course, we can iterate the *compose* operator:

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

In the case where k is equal to *Zero*, we need to define a function of type *Fin Zero* $\to$ *Place*. As *Fin 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 seen how to implement a domain-specific library for distributed arrays, together with an embedded type system that guarantees all array access operations are both safe and local. In contrast to existing work [10], there is no specific set of type rules; instead, equivalent properties are enforced by a general-purpose language with dependent types. We have defined semantics for our library in the form of a total, functional specification. Although this 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, this approach can be extended to other domains: a dependently-typed language accommodates domain specific libraries with their own embedded type systems [18].

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

Furthermore, the pure model is rather naive. It would be interesting to explore a more refined model, where every place maintains its own heap. As our example in the previous section illustrated, assuming the presence of a global distribution does not scale well. Decorating every array with a distribution upon its creation should help provide locality-information when it is needed.

We have not discussed how code in the *IO* or *DIO* monad is actually compiled. At the moment, Agda can only be compiled to Haskell. Agda does provide several pragmas to customise how Agda functions are translated to their Haskell counterparts. The ongoing effort to support data parallelism in Haskell [4, 5] may therefore provide us with a most welcome foothold.

The domain-specific language for manipulating arrays presented in this paper contains an operation for the *allocation* of memory, but no operation for memory *deallocation*. The system presented here assumes garbage collection is handled automatically. This is not a fundamental limitation of this approach: in principle, dependently typed programming languages can enforce any invariant that can be described in constructive predicate logic. In particular, we could extend the shape data type not only capture the amount of data that has been allocated, but also record the locations that are still valid. Any memory access operation then needs to provide a proof that the location being accessed has not been deallocated. Where this is certainly feasible, there is still a substantial challenge in *engineering* a solution that does not put a strain on the programmer to provide explicit proofs or clutter type signatures with too much information.

There are many features of X10 that have not been discussed here at all. Most notably, I have refrained from modelling many of X10's constructs that enable asynchronous communication between locations, even though I would like to do so in the future.

Finally, it is necessary to explore larger examples to acquire a better understanding of how this approach scales. At the moment, I cannot predict how efficient the resulting code will be; nor do I know how difficult it will be to reason about large, realistic distributed algorithms.

Acknowledgements

A previous version of this article was published at the Symposium on Trends in Functional Programming in 2008. Thorsten Altenkirch, my PhD supervisor, suggested this line of research and helped prepare our original submission, for which I am grateful. I would also like to thank the anonymous reviewers for their valuable comments on both versions of this paper.

References

1. 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.
2. Thorsten Altenkirch, Conor McBride, and Peter Morris. Generic programming with dependent types. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *LNCS*. Springer-Verlag, 2007.
3. Robert Atkey. Parameterised notions of computation. In *Proceedings of the Workshop on Mathematically Structured Functional Programming*, 2006.
4. 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.

5. 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.
6. Brad Chamberlain, Steve Deitz, Mary Beth Hribar, and Wayne Wong. Chapel. Technical report, Cray Inc., 2005.
7. 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), 2000.
8. 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*, 2005.
9. Christian Grothoff, Jens Palsberg, and Vijay Saraswat. Safe arrays via regions and dependent types. Submitted for publication.
10. Christian Grothoff, Jens Palsberg, and Vijay Saraswat. A type system for distributed arrays. Unpublished draft.
11. Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.
12. Ben Liblit and Alexander Aiken. Type systems for distributed data structures. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 199–213, 2000.
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. To appear in the Journal of Functional Programming.
15. Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, 1990.
16. Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.
17. Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 457–474, 2008.
18. Nicolas Oury and Wouter Swierstra. The power of Pi. In *ICFP '08: Proceedings of the Thirteenth ACM SIGPLAN International Conference on Functional Programming*, 2008.
19. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.
20. Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, 2006.
21. Sven-Bodo Scholz. Single Assignment C — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.
22. Wouter Swierstra. *A Functional Specification of Effects*. PhD thesis, University of Nottingham, 2009.
23. 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.