Agda Meets Accelerate

Peter Thiemann[1] and Manuel M. T. Chakravarty[2]

[1] University of Freiburg, Germany,
thiemann@informatik.uni-freiburg.de
[2] University of New South Wales, Sydney, Australia,
chak@cse.unsw.edu.au

Abstract. Embedded languages in Haskell benefit from a range of type extensions, such as type families, that are subsumed by dependent types. However, even with those type extensions, embedded languages for data parallel programming lack desirable static guarantees, such as static bounds checks in indexing and collective permutation operations.
This observation raises the question whether an embedded language for data parallel programming would benefit from fully-fledged dependent types, such as those available in Agda. We explored that question by designing and implementing an Agda frontend to Accelerate, a Haskell-embedded language for data parallel programming aimed at GPUs. We discuss the potential of dependent types in this domain, describe some of the limitations that we encountered, and share some insights from our preliminary implementation.

Keywords: programming with dependent types, data parallelism

1 Introduction

Generative approaches to programming parallel hardware promise to combine high-level programming models with high performance. They are particularly attractive for targeting restricted architectures that cannot efficiently execute code aimed at conventional multicore CPUs. One prime example are GPUs (graphics processor units), which require a high degree of data parallelism, restricted control flow, and custom tailored data access patterns to be efficient. Previous work —for example, Accelerator [17], Copperhead [2], and Accelerate [3]— demonstrates that embedded array languages with a custom code generator can meet those GPU constraints with carefully designed language constructs.

Given a host language with an expressive type system, it is attractive to leverage that type system to express static properties of the embedded language. For example, Accelerate, an embedded array language for Haskell, uses Haskell's recent support for type-level programming like GADTs and type families in that manner [3]. This design choice is desirable for approaches relying on run-time code generation: each potential fault at application run time should be discovered by a compile-time fault in the embedded language. Moreover, static guarantees hold the potential to improve the predictability of parallel performance.

Dependent types [9] are an established approach to certified programming, where invariants are established in the form of types and proven at compile time. Many of Haskell's type-level extensions used in Accelerate approximate aspects of dependently-typed programming. Hence, it is natural to ask whether fully-fledged dependent types, such as those provided by Agda, improve the specification of an embedded language like Accelerate, whether they increase the scope of static guarantees, and whether they may be leveraged to predict performance more accurately.

This paper is a first investigation into this topic. It reports on a partial port of Accelerate to a new, dependently-typed host language, Agda [1, 10]. Agda is particularly suited to this port because of its foreign function interface to Haskell, which enables it to directly invoke the functionality of Accelerate. The main contributions of this paper are the following:

- We identify and discuss the challenges of combining generative embedded languages with dependent typing (Section 4).
- We propose predicated arrays to overcome some of these challenges (Section 5).
- We outline an implementation of the main parts of Accelerate in Agda using the Agda-Haskell FFI for code execution (Section 6).

Overall, our investigation has the following structure. After recalling some background on Agda and Accelerate in Section 2 and describing related work in Section 3, Section 4 discusses potential uses of dependent types in an array-oriented data parallel language and how they were realized in our implementation. Section 5 considers conceptual problems and limitations that we ran into when constructing the Agda frontend for Accelerate. Section 6 explains some technical details of the implementation and discusses some example code.
Source code is available at https://github.com/mchakravarty/accelerate-agda.

2 Background

2.1 Agda

Agda [1,10] is a dependently-typed functional programming language. Its basis is a dependently-typed lambda calculus extended with inductive data type families, dependent records, and parameterized modules. At the same time, Agda is also a proof assistant for interactively constructing proofs in an intuitionistic type theory based on the work of Per Martin-Löf [9].

One attractive feature of Agda's inductive data type families is the ability to construct indexed data types. A familiar example for such an indexed data type is the type `Vec A n` of vectors of fixed length `n` and elements of type `A`. This vector data types can be equipped with an access operation that restricts the index to the actual length of the vector at compile time.[3]

[3] An identifier can be an almost arbitrary string of Unicode characters except spaces, parentheses, and curly braces. Agda also supports mixfix syntax with the position of arguments indicated by underscores in the defining occurrence of an identifier.

```
data Nat : Set where
  zero : Nat
  suc  : Nat -> Nat

data Vec (A : Set) : Nat -> Set where
  []   : Vec A zero
  _::_ : {n : Nat} -> A -> Vec A n -> Vec A (suc n)
```

The above defines the type `Nat` of natural numbers and an indexed data type `Vec A n` where `A` is a type and `n` is a natural number. The latter type comes with two constructors, `[]` for the vector of length zero and `_::_` for the infix cons operator that increases the length by one.

One way of writing a safe access operation first defines an indexed type that encodes the required less-than relation on natural numbers.

```
data _<_ : Nat -> Nat -> Set where
  z<s : {n : Nat} -> zero < suc n
  s<s : {m n : Nat} -> m < n -> suc m < suc n
```

Lines two and three of the definition encode named inference rules for the cases that $0 < n + 1$ (for all n) and that $m + 1 < n + 1$ if $m < n$ (for all m, n).

The access operation takes a vector of length `n`, an index `m`, and a proof of `m < n` (a derivation tree) to produce an element of the vector.

```
get : {A : Set} {n : Nat} -> Vec A n -> (m : Nat) -> m < n -> A
get []       _       ()      -- impossible case
get (x :: xs) zero    z<s     = x
get (x :: xs) (suc m) (s<s p) = get xs m p
```

This code cannot fail at run time because a caller has to construct the proof tree for `m < n` before invoking `get`. Thus, an "index out of bounds" error cannot happen. (In Agda, arguments in curly braces are *implicit arguments* that will be inferred if omitted in an application.)

2.2 Accelerate

Accelerate [3] is a *generative* data-parallel array language embedded into Haskell, which targets GPUs. Being generative, its data-parallel array operations are not executed directly. Instead, Accelerate constructs abstract syntax trees (AST) representing an entire data-parallel subcomputation. These *computation representations* are executed using a `run` operation that accepts such a representation (of type `Acc a`), compiles it to GPU kernels, uploads it to a device, executes it, and retrieves the results.[4]

```
CUDA.run :: Arrays a => Acc a -> a
```

[4] To distinguish Haskell code from Agda code, we display Haskell code in a blue box.

The type class constraint `Arrays a` restricts the result type to a single array or a tuple of arrays.

As computation representations of type `Acc a` are compiled at application run time, all `Acc` compilation errors are effectively *run-time errors* of the application. Hence, Accelerate uses a range of Haskell type system extensions to statically type Accelerate expressions, such that these run-time errors are avoided where possible. In particular, Accelerate uses GADTs [7], associated types [4], and type families [14].

As a simple example of an Accelerate program, consider a function implementing a dot product:

```
dotp :: Vector Float -> Vector Float -> Acc (Scalar Float)
dotp xs ys = let { xs' = use xs; ys' = use ys }
             in  fold (+) 0 (zipWith (*) xs' ys')
```

The types `Vector` and `Scalar` represent one- and zero-dimensional arrays. Plain arrays, such as `Vector Float` are conventional Haskell arrays, using an unboxed representation to improve performance. However, when they are wrapped into the constructor `Acc`, such as in `Acc (Scalar Float)`, they represent arrays of the embedded language and are allocated in GPU memory, which in current high-performance GPUs is physically separate from CPU memory.

The `use` operation makes a Haskell array available in the embedded language by wrapping it into the `Acc` constructor. It amounts to copying it to GPU memory.[5] The operations `fold` and `zipWith` represent collective operations on Accelerate arrays, effectively producing a representation of an array computation yielding a single float value (`Scalar Float`). The code relies on (type class) overloading: `0`, `(+)`, and `(*)` are overloaded to construct abstract syntax.

The types `Scalar` and `Vector` are type synonyms instantiating a shape-parameterised array type to the special case of zero and one dimensional arrays:

```
type Scalar e = Array DIM0 e
type Vector e = Array DIM1 e
```

In the general type for `use`, the class `Elt` characterizes all types that may be held in Accelerate arrays. These are currently primitive types and tuples.

```
use :: Elt e => Array sh e -> Acc (Array sh e)
```

Common dimensions, such as `DIM0`, `DIM1`, and so on, are predefined, but to enable shape polymorphic computations, along the lines pioneered in the Haskell array library Repa [8], shapes are inductively defined using type-level snoc lists built from the data types `Z` and `:.`. The use of snoc lists simplifies the type signatures of fold operations that reduce or abstract over the least significant dimensions.

[5] Accelerate employs caching to avoid the transfer of arrays that are already available in GPU memory.

```
data Z       = Z
data sh :. i = sh :. i
-- Types for often used dimensions
type DIM0 = Z
type DIM1 = DIM0 :. Int
-- and so on
```

3 Related Work

Peebles formalizes parts of the Repa API using Agda [11]. The formalisation relies on the same shape structure as Accelerate, but array computations are neither embedded nor can parallel high-performance code be generated.

Swierstra and Altenkirch investigated the use of dependent types for *distributed* array programming [15, 16]. Their notation for distributed arrays is inspired by the X10 language [13]. They focus on expressing locality awareness.

Dependent ML is an ML dialect with a restricted form of dependent types, which, among other applications, may be used to statically check array bounds [18]. However, only simple indexing and array updating are considered and not aggregate array operations, such as those provided by Accelerate.

Accelerator [17] enables embedded GPU computations in C# programs; it subsequently also added F# support. However, no attempt is made to track properties of array programs statically. Similarly, Copperhead [2] embeds an array language into Python, but does not attempt to track information statically.

4 Dependent Types for Accelerate

In this section, we investigate the potential uses of dependent typing in a language like Accelerate and point out how they may be implemented in Agda. First, we review some basics of the embedding.

4.1 Embedding of Haskell Types

Accelerate supports a wide range of numeric types, characterized by the type class `Elt`, as base types for array computations. Almost all of these types lack a suitable counterpart in Agda, which only supplies computationally expensive encodings for natural and rational numbers. For that reason, our embedding keeps the Haskell types abstract in Agda. To specify the types of functions that are polymorphic in such a Haskell type or depend on it in some way, we have reified the possible element types as an Agda type `Elt`:

```
data Elt : Set where
  Bool   : Elt
  Int    : Elt
```

```
  Float  : Elt
  Double : Elt
  Pair   : Elt -> Elt -> Elt
  -- and so on
```

Corresponding to Haskell type classes that are used in Accelerate, our embedding supplies predicates that characterize subsets. For example, the set of numeric types is defined by a predicate `Numeric`:[6]

```
Numeric : Elt -> Set
Numeric Int = ⊤
Numeric Float = ⊤
Numeric Double = ⊤
Numeric _ = ⊥
```

The embedding declares further subsets all in the same style.

4.2 Array Types

To demonstrate the Agda embedding in action, we translate the dot product example from Section 2.2 to Agda.[7]

```
dotp : forall {E : Elt} {{p : Numeric E}} {n : Nat}
     -> PreVector n E -> PreVector n E -> Scalar E
dotp{E} xs ys =
  let xs' = use xs
      ys' = use ys
  in  fold _+_ ("0" ::: E) (zipWith _*_ xs' ys')
```

Unlike the Accelerate code, this function is polymorphic with respect to the array element type, provided it is numeric. The length parameter n ensures that the two input vectors have the same size. The `PreVector` type of the arguments corresponds to the plain `Vector` type in Accelerate, whereas the result type `Scalar E` corresponds to `Acc (Scalar E)`—a piece of abstract syntax.

The `use` function works as before, but its type includes more information:

```
use : {sh : Shape}{E : Elt} -> PreArray sh E -> Array sh E
```

Like `E`, the index `sh` is now an element of an ordinary type instead of having to rely on type-level snoc lists:[8]

```
data Shape : Set where
  Z     : Shape
  _:<_> : Shape -> Nat -> Shape
```

[6] $\top$ is a one-element type, whereas $\bot$ is a type without elements. These types customarily represent truth and falsity.

[7] In Agda, arguments in double curly braces are *instance arguments* [5] that are aggressively inferred. We use them like type class constraints in Haskell.

[8] Recent work on Haskell's type system manages to avoid this issue [19].

Asking for arrays of equal shape, as in the signature of `use`, means that the arrays have to have the exact same layout. The `PreVector` and `Vector` types are just synonyms as in Haskell:

```
PreVector n E = PreArray (Z :< n >) E
Vector n E    = Array    (Z :< n >) E
```

The functions `fold`, `zipWith`, and `:::` are discussed in the subsequent subsections. The functions `_+_` and `_*_` both have the same type:

```
_+_ _*_ : {E : Elt} {{p : Numeric E}} -> Exp E -> Exp E -> Exp E
```

They are restricted to arguments of numeric type and construct abstract syntax for an addition or a multiplication by delegating to the corresponding Accelerate functions. The type `Exp E` denotes an AST of an expression of type `E`.

4.3 Exact Checking of Array Bounds

Accelerate's API features expressive type constraints that describe the shape of the array arguments and results. These constraints ensure that no shape mismatches occur (e.g., a 1D array cannot be considered 2D), but they do not ensure at compile time that the sizes of the dimensions match up. Such a mismatch results in a run-time error.

As an example, consider the function `reshape`. It takes a target shape `sh` and an array of source shape `sh'` and changes the layout of that array to `sh`.

```
reshape :: Exp sh -> Acc (Array sh' e) -> Acc (Array sh e)
```

For this reshaping to work correctly, the underlying number of elements must remain the same. For example, while it makes sense to reshape a two-dimensional 3×4-array to a vector of size 12 or to a three-dimensional $3 \times 2 \times 2$-array, an attempt to reshape to a 2×5-array should be rejected at compile time.

As `Shape` is an ordinary data type in Agda, we can define a `size` function that computes the number of elements stored in an array of a certain shape.

```
size : Shape -> Nat
size Z = 1
size (sh :< n >) = size sh * n
```

Now we can state an accurate type for `reshape` in Agda, which involves an extra argument with a proof that the source and target shapes have the same size.

```
reshape : {sh : Shape} {E : Elt}
       -> (sh' : Shape) -> Array sh E -> (size sh ≡ size sh')
       -> Array sh' E
```

There is a subtle difference to the original signature. In Accelerate, the first argument is an *expression* that produces a value of type `sh` at run time, whereas the Agda `reshape` requires a `Shape` as its first argument. Hence, Agda `reshape` computes the shape in the host language on the CPU, whereas the original signature admits to compute the new shape in the embedded language as part of a GPU computation. In other words, we slightly restrict expressiveness here to gain more static information, we will get back to that issue when discussing filtering.

In Agda, functions like `map` and `zipWith` obtain more precise types. The type of `map` tells us that the input shape is identical to the output shape:

```
map : {A B} {sh} -> (Exp A -> Exp B) -> Array sh A -> Array sh B
```

Similarly, the type of `zipWith` restricts its input arrays to identical shapes:

```
zipWith : {A B C} {sh} -> (Exp A -> Exp B -> Exp C)
        -> Array sh A -> Array sh B -> Array sh C
```

The latter type is more restrictive than the Accelerate implementation of `zipWith`. Instead of checking the sizes of the input arrays, it truncates them to the respective minima. We also developed an Agda type that directly corresponds to this implementation. It requires a binary function `isect` that computes the minimum of two shapes of the same rank, which we leave as an exercise to the reader.

```
zipWith' : {A B C} {shA shB} {p : rank shA ≡ rank shB}
         -> (A -> B -> C)
         -> Array shA A -> Array shB B -> Array (isect shA shB p) C
```

4.4 Associativity of Operations

Some parallel reduction operations require their base operation to be associative to return a predictable result. Here are two examples from Accelerate.

```
fold  :: (Shape ix, Elt a) =>
         (Exp a -> Exp a -> Exp a) -> Exp a ->
         Acc (Array (ix :. Int) a) -> Acc (Array ix a)
fold1 :: (Shape ix, Elt a) =>
         (Exp a -> Exp a -> Exp a) ->
         Acc (Array (ix :. Int) a) -> Acc (Array ix a)
```

In both cases, the text of the documentation says that "the first argument needs to be associative" and furthermore the `fold1` documentation "requires the reduced array to be non-empty". The second requirement can be enforced by asking for a suitable proof object on each call of `fold1`:

```
fold1 : ... -> Array (sh :< n >) E -> (size sh * n > 0)
          -> Array sh E
```

The first requirement can be rephrased to saying that the first two parameters of `fold` together form a monoid, which requires an associative operation with a unit element. The concept of a monoid can be formalized in Agda, which has indeed been done in the standard library. Unfortunately, the formalization from the library cannot be used because Accelerate deals with ASTs, not with values. So, a formalization is required that states that the meaning of an AST-encoded function is associative and the meaning of another AST-encoded constant is its unit element. Given that Accelerate encodes AST construction using higher-order abstract syntax, such a formalization is not straightforward. Moreover, even given expressions with a fixed meaning, associativity has to be proved on a case by case basis.

In any case, providing such information would be done by including an additional argument that holds a suitable proof object, as in

```
fold : forall {E}{sh}{n}
     -> (f : Exp E -> Exp E -> Exp E) -> (e : Exp E)
     -> Array (sh :< n >) E -> IsMonoid f e -> Array sh E
```

where

```
IsMonoid : forall {E} -> (Exp E -> Exp E -> Exp E) -> Exp E -> Set
IsMonoid f e = ( IsAssociative f , IsUnit f e)
```

Some readers may object that neither addition nor multiplication of floating point numbers is associative [6]. However, for advanced optimizations, the exploitation of algebraic laws is a necessity and the involved degradation of precision or change of result is accepted or accounted for in the error estimates. Moreover, there are other operations, like min or max, that are commonly used with `fold`-like operations, which are truly associative. Last, but not least, the associativity declarations serve as important documentation that passing an inherently non-associative function will produce unpredictable, implementation-dependent results.

4.5 Embedding of Constants

Accelerate relies on Haskell's built-in support for the type classes `Num` and `Fractional` to embed constants. The Haskell compiler reads each integer literal as a value of type `Integer`, which is a built-in type of arbitrary precision integers. To this value, Haskell applies the function `fromInteger` that converts to the type expected by the context. Similarly, floating point constants are read as values of type `Rational` (`Integer` fractions) and then converted using `fromRational`. Accelerate provides instances of these type classes that define `fromInteger` and `fromRational` to produce suitable AST fragments.

Because of Agda's lack of support for overloaded numeric literals, we embed numeric literals for integers and floating point numbers using a string with an explicit type annotation that determines the parsing of the string. Here are some example embeddings:

```
"3.1415926" ::: Float
"6.0221415E23" ::: Double
```

Recall that `Float` and `Double` are not types, but rather values of type `Elt`. The `:::` operation is the workhorse of the embedding:

```
_:::_ : (s : String) -> (E : Elt)
     -> {{nu : Numeric E}} -> {p : T (s parsesAs E)} -> Exp E
s ::: E = Ex (constantFromString (EltDict E) (ReadDict E) s)
```

The arguments `s` and `E` are explicit, but the remaining ones are inferred by Agda. As mentioned, the argument `nu` is an instance argument; it is automatically filled-in with a suitably typed value in scope [5]. As before, the predicate `Numeric` plays the role of a type class that characterizes the numeric types.

The function `parsesAs` dispatches on its "type" argument and parses the string to check whether it is an acceptable literal of the expected type. The function `constantFromString` is imported from Accelerate. It is an overloaded function that requires two type dictionaries, which are computed from `E` using the functions `EltDict` and `ReadDict`. This results in a flexible way of handling literals, which worked well in our examples.

5 Limitations

In a number of places, Accelerate's generativity limits the applicability of dependent typing. We already mentioned that the formalization of associativity or of the concept of a monoid cannot be verified in Agda because such properties have to be asserted for abstract syntax.

For a related problem, consider an implementation of the `filter` operation that takes a predicate and a source array and returns an array that only contains the elements of the source array fulfilling the predicate. First of all, filtering only makes sense for one-dimensional arrays, that is, for vectors. To see the second catch, let's try to write down a dependent type signature for `filter`.

```
filter : forall {n m : Nat}{E : Elt}
       -> Vector n E -> (Exp E -> Exp Bool) -> Vector m E
```

The problem is that the size of the result cannot be determined statically — that is, we cannot simplify define a type-level function that determines the length of a vector. Why? `Vector m E` is not a representation of a vector. Instead, it is a representation of a computation that, *once run,* produces a vector.

Similarly, we cannot define a function that uses the predicate passed to `filter` to count the number of elements that will appear in `filter`'s result. Such a function would need access to the elements of the `filter`ed vector, but, as discussed, we cannot even get its length. Moreover, such a counting function would need to evaluate the predicate. We cannot do that as the predicate of type `Exp E -> Exp Bool` maps abstract syntax to abstract syntax; it *does not* directly implement a Boolean predicate. We might consider to include an evaluator for abstract syntax to lift these restrictions. However, that evaluator would not be the code actually executed on the GPU, and hence, it doesn't seem to be any more valuable than simply asserting an axiom concerning the size of `filter`'s result. That is the price we pay for a generative approach, where at program runtime, we dynamically generate the code to be executed on the GPU.

We encounter similar restrictions if we try to, at least, establish that `m` must be less than or equal to `n` for `filter`. We cannot prove this constraint as the GPU code of `filter` is not available to us — it is generated by the underlying Haskell library. Even if we had access to that code, any statements about its properties would need to be based on the semantics of CUDA (i.e., NVIDIA's C dialect for GPU programming).

We might contemplate employing an existential type like

```
exists Nat (\ m -> m <= n -> Vector m E)
```

but it is not possible to build such an existential package because the evidence `m` is not available when the existential package has to be constructed.

However, we may use an alternative encoding of arrays that is compatible with filtering. The idea is to keep all elements but mark those which are no longer present because they have been filtered out. There are several ways of implementing this idea. The simplest approach is to pair up each element with a boolean flag that indicates its presence, which we call *predicated arrays*:[9]

```
FVector : Nat -> Elt -> Set
FVector n E = Vector n (Pair Bool E)
```

In this encoding, filtering is quite simple because the length of the `FVector` does not change. Furthermore, filtering could be extended to multi-dimensional arrays, although the result might require careful interpretation.

```
filterF : forall {n : Nat}{E : Elt}
        -> (Exp E -> Exp Bool) -> FVector n E -> FVector n E
filterF {n}{E} pred vec = map g vec
  where g : Exp (Pair Bool E) -> Exp (Pair Bool E)
        g bx = pair ((fst bx) && p (snd bx)) x
```

Mapping, which applies a function to each element of an array, becomes more complicated as it either has to materialize a dummy result for each absent element in the argument vector or apply the function to absent elements, too. This makes `filter` reminiscent of the `where` statement of the SIMD language C* [12].

```
mapF : forall {n : Nat}{E F : Elt}
     -> Exp F -> (Exp E -> Exp F) -> FVector n E -> FVector n F
mapF {n}{E}{F} defaultF f vec = map g vec
  where g : Exp (Pair Bool E) -> Exp (Pair Bool F)
        g bx = if (fst bx) then (pair (fst bx) (f (snd bx)))
                           else (pair (fst bx) defaultF)
```

Some operations can get rid of absent elements. A fold operation which reduces a filtered vector with a monoid returns a single value. In Accelerate, such a value has type `Scalar`, which is a synonym for an array of dimension 0.

[9] Accelerate currently does not support `Maybe` types as array elements.

```
foldF : forall {n : Nat}{E : Elt}
      -> (Exp E -> Exp E -> Exp E) -> Exp E
      -> FVector n E -> Scalar E
foldF f e vec =
  fold f e (map (\ bx -> if (fst bx) then (snd bx) else e) vec)
```

Operations like `fold1` and the scan operations extend to this representation, but they cannot revert to a non-filtered representation.

In the end, such a representation may not even lead to reduced efficiency on a GPU. As long as all computations take the same path, all processing elements work in unison. As soon as there are different paths in the same computation step, then some elements will be idle for part of the computation step. So it would be most advantageous to organize work as uniformly as possible by reorganizing the array so that the present and the absent elements are grouped together. A segmented array might be a suitable representation.

6 Implementation

Ordinarily, Agda is an interactive tool for constructing proofs and verified programs. Programs may be run, which amounts to normalizing Agda expressions, but this process is not very efficient.

Alternatively, an interactively developed program may be compiled to Haskell using the Alonzo compiler. It supports a Haskell foreign function interface (FFI), for Agda programs to invoke Haskell functions. Using this interface amounts to declaring a typed identifier in Agda and then binding the identifier to a suitably typed Haskell function. As an example, consider the import of the `use` function.

```
postulate
  useHs : {E : Set}
      -> HsEltDict E -> HsArray HsDIM1 E -> Acc (AccArray HsDIM1 E)
  {-# COMPILED useHs        (\ _ -> Accel.use) #-}
```

The first three lines introduce the typed identifier `useHs` and the last line is a pragma for the Alonzo compiler that binds the Agda identifier `useHs` to the Haskell expression on the right. But wait, this type looks very unpleasant and quite different to the one mentioned in Section 4.2. This difference arises as the type translation of Alonzo is unable to cope with the index type `Shape`. Hence, the interface uses a simplified array type and adapter functions are required, in the worst case, both on the Agda side and on the Haskell side of the interface.

At the foreign function interface level, all arrays are considered as one-dimensional arrays. Additional arguments are passed to encode the shape information as far as it is needed. The Agda adapter provides the encoding of this structure and the Haskell adapter decodes it again.

We believe that these adaptations only have a minor performance impact because (1) most functions just manipulate abstract syntax, so that only AST construction is affected, and (2) internally, Accelerate considers all arrays as one-dimensional so that operations like `reshape` are no-ops at run time.

Here is the Agda adapter for `use`:

```
use : forall {sh : Shape}{E : Elt} -> PreArray sh E -> Array sh E
use {sh}{E} (PA y) = Ar (useHs (EltDict E) y)
```

It makes use of two wrapper types. `PreArray` wraps a one-dimensional Haskell array using the constant `HsDIM1` (the `DIM1` type shown in Section 2.2 imported from Haskell via FFI) and the function `EltType` (not shown), which interprets a value of type `Elt` as a Haskell type. The latter types are also imported via FFI.

```
data PreArray (E : Elt) : Shape -> Set where
  PA : {sh : Shape} -> HsArray HsDIM1 (EltType E) -> PreArray sh E
```

The `Array` type wraps an AST reference for an Accelerate array, where `Acc` and `AccArray` are types imported from Haskell.

```
data Array (E : Elt) : Shape -> Set where
  Ar : {sh : Shape} -> Acc (AccArray HsDIM1 (EltType E)) -> Array sh E
```

The `EltDict` function translates a value `(E : Elt)` into a Haskell expression that evaluates to a dictionary for the Haskell type of `E` for the Haskell type class `Elt`. Such a dictionary is passed, whenever the corresponding Haskell function has type class constraints.

```
EltDict : (E : Elt) -> HsEltDict (EltType E)
```

The Haskell side of the adapter has several purposes. First, it materializes the type class dictionaries from the encoding that we just discussed. Second, it reconstructs sufficient information about the array shape so that the intended operation can execute. Here is the code for `Accel.use`, where the module name `A` is a shorthand for `Data.Array.Accelerate`.

```
use :: EltDict e -> Array A.DIM1 e -> A.Acc (A.Array A.DIM1 e)
use EltDict (ARRAY ar) = (A.use ar)
```

It does not have to reconstruct any information except the type class constraint. This constraint is materialized using the type `EltDict` below.

```
data EltDict e where
  EltDict :: (A.Elt e) => EltDict e
```

This datatype is built such that each value captures the `Elt` dictionary of type `e`. It remains to build such values for all types that we want to transport across the FFI. These are the values used by the (Agda) `EltDict` function. Here are two examples.

```
eltDictBool :: EltDict Bool
eltDictBool = EltDict

eltDictInt :: EltDict Int
eltDictInt = EltDict
```

As an example for a function that requires more work on both sides, consider the `fold` operation.

```
fold : forall {E}{sh}{n}
     -> (Exp E -> Exp E -> Exp E)
     -> Exp E
     -> Array (sh :< n >) E
     -> Array sh E
fold {E}{sh}{n} f (Ex e) (Ar a) =
  Ar (foldHs (EltDict E) (toHsInt (size sh)) (toHsInt n)
             (unwrap2 f) e a)
```

As values of type `Exp` also need a wrapper type in Agda (it is not possible to import type constructors via the FFI), there is some unwrapping going on for the `e` and `f` arguments. The implementation of `fold` just calls the `foldHs` function and encodes the information about the shape in two integer arguments. Here, `size sh` is the size of the result and `n` is the size of the dimension that is folded. As these values are initially available as Agda natural numbers, they need to be converted to Haskell numbers using the function `toHsInt`.

The `foldHs` function is defined via the FFI.

```
postulate
  foldHs : {A : Set}
         -> HsEltDict A
         -> HsInt
         -> HsInt
         -> (AccExp A -> AccExp A -> AccExp A)
         -> AccExp A
         -> Acc (AccArray HsDIM1 A)
         -> Acc (AccArray HsDIM1 A)
  {-# COMPILED foldHs      (\_ -> Accel.fold) #-}
```

The Haskell adapter reconstructs the `Elt` dictionary as before, but it also needs to reshape the one-dimensional array representation into a two-dimensional one for executing the fold operation. The two size arguments are required for exactly this reshape operation. With that insight, the code is straightforward.

```
fold :: EltDict a
     -> Int -> Int
     -> (A.Exp a -> A.Exp a -> A.Exp a)
     -> A.Exp a
     -> A.Acc (A.Array A.DIM1 a)
     -> A.Acc (A.Array A.DIM1 a)
fold EltDict size2 size1 f e a =
     (A.reshape (A.lift (A.Z A.:. size2))
      (A.fold f e
       (A.reshape (A.lift (A.Z A.:. size2 A.:. size1)) a)))
```

Fortunately, the `fold` example is about as complicated as the adapter code gets. There are also many cases where at least one side of the adapter code is trivial. However, each case must be considered separately.

7 Conclusion

We have built an experimental Agda frontend for the Accelerate language. The goal of this experiment was to explore potential uses of dependently-typed programming for data-parallel languages.

At the moment, the outcome of the experiment is mixed. It is successful, because we have been able to construct Agda functions for a representative sample of Accelerate's functionality. However, there was less scope for encoding extra information in the dependent types than we had hoped for. Exact matching of array bounds works, but results in restrictions (like the problems with `zipWith` and filtering) that were not anticipated.

Exploiting algebraic properties did not work out in the intended way, mainly because it boils down to asserting that some AST denotes an associative function. However, these assertions cannot be proven: the proof would have to apply the semantics to the AST, but the AST is an abstract type in our implementation. An AST representation in Agda might give us a better handle at this problem.

In some places, the Agda frontend is less dynamic than Accelerate. In a number of places, Accelerate accepts a run-time value of type `Exp sh` for a shape argument, where the Agda frontend requires a value of type `Shape`. To address this problem, we would have to include a `Shape`-indexed encoding of the `Shape` type in the `Elt` type so that we can describe the type of an expression whose value has a certain shape.

Finally, the type translation of Agda's FFI has a number of shortcomings that cause problems when transporting information between Agda and Haskell. One part of the problem is, unfortunately, the rich type structure of Accelerate's API which already encodes many useful constraints. An alternative, untyped (or less-typed) interface to Accelerate would make the adaptation to an Agda frontend simpler.

References

1. A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda - a functional language with dependent types. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78, Munich, Germany, 2009. Springer.
2. B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. Technical Report UCB/EECS-2010-124, University of California, Berkeley, 2010.
3. M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In M. Carro and J. H. Reppy, editors, *Workshop on Declarative Aspects of Multicore Programming, DAMP 2011*, pages 3–14, Austin, TX, USA, Jan. 2011. ACM.

4. M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In B. C. Pierce, editor, *Proceedings International Conference on Functional Programming 2005*, pages 241–253, Tallinn, Estonia, Sept. 2005. ACM Press, New York.
5. D. Devriese and F. Piessens. On the bright side of type classes: Instance arguments in Agda. In O. Danvy, editor, *Proceedings International Conference on Functional Programming 2011*, pages 143–155, Tokyo, Japan, Sept. 2011. ACM Press, New York.
6. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.
7. S. L. P. Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In J. Lawall, editor, *ICFP*, pages 50–61, Portland, Oregon, USA, Sept. 2006. ACM Press, New York.
8. G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *ICFP '10: Proc. of the 15th ACM SIGPLAN Intl. Conf. on Functional Programming*. ACM, 2010.
9. P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.
10. U. Norell. Dependently typed programming in Agda. In P. W. M. Koopman, R. Plasmeijer, and S. D. Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266, Heijen, The Netherlands, 2008. Springer.
11. D. Peebles. A dependently typed model of the Repa library in Agda. https://github.com/copumpkin/derpa, 2011.
12. J. R. Rose et al. C*: An extended c language for data parallel programming. In *Proceedings of the Second International Conference on Supercomputing*, pages 2–16, 1987.
13. V. Saraswat. Report on the programming language X10. http://dist.codehaus.org/x10/documentation/languagespec/x10-200.pdf, Oct. 2009. Version 2.0.
14. T. Schrijvers, S. L. Peyton Jones, M. M. T. Chakravarty, and M. Sulzmann. Type checking with open type functions. In P. Thiemann, editor, *Proceedings International Conference on Functional Programming 2008*, pages 51–62, Victoria, BC, Canada, Oct. 2008. ACM Press, New York.
15. W. Swierstra. More dependent types for distributed arrays. *Higher-Order and Symbolic Computation*, pages 1–18, 2010.
16. W. Swierstra and T. Altenkirch. Dependent types for distributed arrays. In *Trends in Functional Programming*, volume 9, 2008.
17. D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *ASPLOS-XII: Proc. of the 12th Intl. Conf. on Architectural Support for Programming Lang. and Operating Systems*, pages 325–335. ACM, 2006.
18. H. Xi. Dependent ML: An approach to practical programming with dependent types. *Journal of Functional Programming*, 12(2), Mar. 2007.
19. B. A. Yorgey, S. Weirich, J. Cretin, S. L. P. Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In B. C. Pierce, editor, *Proceedings of TLDI 2012*, pages 53–66, Philadelphia, PA, USA, Jan. 2012. ACM.