Chapter 29

Lightweight Invariants with Full Dependent Types

Edwin Brady[1], Christoph Herrmann[1], Kevin Hammond[1]
Category: Position Paper

Abstract: Dependent types allow a programmer to express invariant properties of functions, such as the relationship between the input and output lengths of a list. Several "lightweight" approaches to dependent types have been proposed for existing systems, such as Haskell's Generalised Algebraic Data Types or Type Families. Such approaches are lightweight in the sense that they require minimal modifications to existing systems. However, while these extensions are apparently simple, we find that we often run into limitations fairly quickly. In this paper we will explore these limitations, and show that a full dependent type system allows more straightforward implementation of simple invariants without restricting expressivity.

29.1 INTRODUCTION

Dependent types, which allow types to be predicated on *values*, allow us to specify in advance both precisely what a program is intended to do and the invariants that it is expected to maintain. Dependent types have become a very active area of research, and various "lightweight" approaches have been proposed that either extend existing functional programming languages, e.g. GADTs in Haskell [16, 10] or Ωmega [19], interact with a theorem prover like Concoqtion [9], or that exploit existing language features in creative ways, e.g. [12, 13].

While these approaches are indeed lightweight in the sense that they require little or no change to existing production programming languages, as the complexity of functions increases, so it becomes more difficult to explain the required program invariants using these lightweight systems. Even an apparently simple

[1]School of Computer Science, University of St Andrews, St Andrews, Scotland; Phone: +44 1334-461629; Email: {eb,ch,kh}@cs.st-andrews.ac.uk

example, partitioning a list for sorting, in a way akin to the quicksort divide-and-conquer schema, can be difficult to encode since it requires reasoning about the behaviour of addition and defining a strong link between the representation of the list and the partitions. While these lightweight approaches are certainly powerful, they require a deep understanding of the underlying type system on the part of the programmer in order to explain the necessary reasoning for such programs.

In this paper we will argue that *lightweight* should mean that the programmer has to do *as little work as possible* to demonstrate that a program satisfies the required invariants. It is our contention that a lightweight approach should allow the *programmer* to decide on the level of safety that is required, and be sufficiently flexible that if additional invariants are required, they can be quickly and easily encoded.

29.2 THE IDRIS LANGUAGE

We introduce IDRIS, a new experimental functional language with full dependent types[2]. IDRIS is similar to EPIGRAM [14] or AGDA [15], in that it supports dependent pattern matching. It is built on top of the IVOR [2] theorem proving library. It is a pure functional language with a syntax similar to Haskell with GADTs. The purpose of the language is to provide a platform for practical programming with dependent types. We have used our own implementation, rather than an existing tool, as this gives complete freedom to experiment with abstractions and language features beyond the type system, such as I/O and concurrency. Additionally, although unrelated to the work we present in this paper, the core language of IDRIS is intended as an important step towards a fully dependently typed implementation of Hume [11].

29.3 THE BASIC QUICKSORT ALGORITHM

We will take a functional variant of the standard *quicksort* sorting algorithm as a running example to illustrate our argument. For this example, several different invariants may be desirable, depending on context:

1. sorting must return a list of the same element type as the input[3];
2. sorting must be size preserving;
3. sorting must return a permutation of the original list; and
4. sorting must return an ordered permutation of the original list.

In this paper, we will explore which of these invariants may be specified and guaranteed both in Haskell, with various extensions, and in IDRIS. We will use the same functional variant of *quicksort*, named `qsort` throughout the paper. It

[2] http://www.cs.st-and.ac.uk/%7Eeb/Idris/

[3] This is, of course, easily enforced using conventional type-checking.

first partitions the argument list around the pivot into two sets, values that are less than the pivot, and values that are greater than or equal to the pivot; then sorts each set; and finally appends the two sorted partitions to form the result of the sort. We can implement this in Haskell as follows:

```
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = let part = partition x xs
               in qsort (fst part) ++ (x : qsort (snd part))

partition :: Ord a => a -> [a] -> ([a], [a])
partition pivot []     = ([], [])
partition pivot (x:xs) = let (ys,zs) = partition pivot xs
                         in if (x<pivot) then (x:ys, zs)
                                         else (ys, x:zs)
```

A similar implementation is possible in IDRIS, although the current implementation lacks type classes or syntactic sugar for lists. Lists can be declared using Haskell-style notation:

```
data List a = Nil | Cons a (List a);
```

`qsort` and `partition` can then be defined in the same way as in Haskell:

```
qsort :: (a->a->Bool) -> List a -> List a;
qsort lt Nil = Nil;
qsort lt (Cons x xs)
 = let part = partition lt x xs
   in append (qsort lt (fst part))
             (Cons x (qsort lt (snd part)));

partition :: (a->a->Bool) -> a -> List a -> (List a, List a);
partition lt pivot Nil = (Nil, Nil);
partition lt pivot (Cons x xs)
 = let prec = partition lt pivot xs
   in if (lt x pivot) then (Cons x (fst prec), snd prec)
                      else (fst prec, Cons x (snd prec));
```

In principle, any valid Haskell 98 program can be implemented in IDRIS, except that IDRIS requires top level type declarations on all but nullary functions. This is because type inference for dependent types is, in general, undecidable.

29.4 SIZE PRESERVATION

An important abstract program property concerns the sizes of data structures. Apart from the obvious contribution to program verification this property could also be exploited, for example, by a compiler to improve program performance by using static instead of dynamic memory allocation. Sorting functions deliver

a data structure which has the same size as the input. The programmer could express this by choosing a particular data structure, e.g., an array. However, array operations in Haskell can be expensive unless monadic arrays are used. Moreover, if possible, we want the programmer to be able to choose appropriate data structures for the problem domain rather than just for implementation reasons. Since our sorting function does not use explicit indexing, we have already excluded a typical source of error, when the wrong index is used. The additional size information allows us to also prevent many other simple common errors, for example a typographical error in code causing the wrong list to be used will often be caught since the lists will be different sizes. In this section, we will attempt to implement a size-preserving list partition function, as would be used to implement a functional variant of quicksort, named `qsort`, using GADTs and type families in Haskell and a full dependently-typed approach in IDRIS.

The GADT approach

GADTs [16, 10, 19] are a generalisation of Haskell algebraic data types which allow fairly flexible parameterisation of data constructor types. The following example defines two type constructors `Z` (zero) and `S` (successor) to represent natural numbers in the type language; plus a GADT to define a corresponding type of natural numbers `Nat` which reflects the value of data objects in the type and uses the data constructors `Zero` and `Succ`:

```
data Z
data S n

data Nat :: * -> * where
   Zero :: Nat Z
   Succ :: Nat n -> Nat (S n)
```

Now it is possible to enforce compile-time restrictions on natural numbers when they are used as function arguments and to automatically perform compile-time calculations on such numbers, for example.

In order to define addition by induction on the type, we use a type class so that there is a single name (`plus`) which works for all natural numbers. Note that 0 (`Zero`) and 1 (`Succ Zero`) now have different types.

```
class Plus m n s | m n -> s where
   plus :: Nat m -> Nat n -> Nat s
```

The extended type class definition generalises the type of the function with the parameters `m`, `n` and `s` and contains a functional dependency `m n -> s` telling the compiler that `s` depends on `m` and `n`.

We express the definition of `Plus` inductively with two type class instances, depending on whether the first argument is `Z` or `S m`. In the second instance definition the context `Plus a b c => Plus (S a) b (S c)` tells the compiler about the logic of recursion within the type itself, i.e., it exposes that induction is performed on the first argument. Note that type class instances have to be

stated as predicates, so the third type argument of `Plus` here corresponds to the result of the function `plus`.

```
instance Plus Z b b where
   plus Zero y = y

instance Plus a b c => Plus (S a) b (S c) where
   plus (Succ x) y = Succ (plus x y)
```

Instead of lists we now use vectors which carry their length as a type parameter. We simply state that the length increases by one with every new element.

```
data Vect :: * -> * -> * where
   VNil  :: Vect a Z
   VCons :: a -> Vect a k -> Vect a (S k)
```

In the `qsort` example we have to maintain an invariant stating that the sum of the length of both partitions and the pivot element equals the length of the original vector. Since the decision about which part an element will be assigned to, and thus the absolute size of each part, is not known at compile time, in order to preserve the information about the sum of the sizes, we have to calculate both partitions in a single function. We do this by defining a new GADT for partitions which carries this information as a type context (`Plus l r x`).

```
data Partition :: * -> * -> * where
   MkPart :: Plus l r x =>
             Vect a l -> Vect a r -> Partition a x
```

During partitioning, we have to insert an element either into the left part of the partition or into the right part. Insertion into the left part of the partition is simple:

```
mkPartL :: Plus l r x =>
           a -> Vect a l -> Vect a r -> Partition a (S x)
mkPartL x left right = MkPart (VCons x left) right
```

This typechecks because the instance of the `Plus` type class is defined by analysis of its first argument, and we insert the item into the first argument of the partition. However, insertion into the right part causes a problem:

```
mkPartR :: Plus l r x =>
           a -> Vect a l -> Vect a r -> Partition a (S x)
mkPartR x left right = MkPart left (VCons x right)
```

The obvious way to resolve this is to add another instance of the type class:

```
instance Plus a b c => Plus a (S b) (S c) where
   plus x (Succ y) = Succ (plus x y)
```

While this is clearly a reasonable statement, it yields overlapping instances of the type class. Although compiler flags exist which circumvent the problem, this

dirty hack would eventually lead to a run-time error — the type classes must be resolved at some point, and the run-time system does not have enough information to do this. This is not static safety!

What we need is a way to teach the type checker a weak form of commutativity for `plus`. However, even if we only want to *pretend* that we are always increasing the first argument of `plus` (even though in the definition of `mkPartR`, we actually increase the second argument), we are unable do this here. It turns out that syntactic equality on type parameters is not enough to allow us to write such functions.

Type families

Type families are an experimental feature in Haskell which allow the definition of functions over types [17]. This goes beyond the application of type constructors, as with GADTs. In particular, these type functions can be used by the Haskell type checker to normalise type expressions to a semantically equivalent syntactic structure. Using type families, we can define a new version of the quicksort algorithm. The difference with the previous GADT-based example is that addition can now be defined in the type language itself, and we can define our own addition symbol, `:+`, for use within type expressions. The `type family` syntax allows us to define the `:+` symbol and its arity, and the `type instance` syntax allows us to specify the corresponding normalisation rules, i.e., how the type checker should proceed with eliminating the `:+` symbol.

```
type family n :+ m
type instance Z     :+ m = m
type instance (S n) :+ m = S (n :+ m)
```

We can now use the type operator `:+` to specify the length of the partition:

```
data Partition :: * -> * -> * where
   MkPart :: Vect a l -> Vect a r -> Partition a (l :+ r)
```

As in our previous attempt, appending two vectors and extending the left part is straightforward, since the `mkPartL` function recurses on its first argument.

```
app :: Vect a m -> Vect a n -> Vect a (m :+ n)
app VNil ys = ys
app (VCons x xs) ys = VCons x (app xs ys)

mkPartL :: a -> Vect a l -> Vect a r ->
           Partition a (S (l :+ r))
mkPartL x left right = MkPart (VCons x left) right
```

Before defining `mkPartR` and the partitioning function, we need to define a set of lemmas that express semantic equalities for types. The `rewrite` function converts the type of its first argument using the appropriate equality proof term for the equality as its second argument. We omit the definitions for space reasons, but give the types below:

```
data EQ2 :: * -> * -> * -> * -> * where
   EQ2 :: EQ2 x y a a

plus_Z   :: EQ2 n m (n :+ Z)   (n)
plus_nSm :: EQ2 n m (n :+ S m) (S n :+ m)
rewrite  :: c a -> EQ2 x y a a' -> c a'
```

Insertion into the right part of the partition requires a rewrite on the type expression so that we can normalise the first argument of `:+`.

```
mkPartR :: forall a l r. a -> Vect a l -> Vect a r ->
           Partition a (S (l :+ r))
mkPartR x left right =
  MkPart left (VCons x right)
  `rewrite` (plus_nSm :: EQ2 l r (l :+ S r) (S l :+ r))
```

Function `partInsert` inserts one element, `insertN` a vector of elements and `partition` initialises the `Partition` data type.

```
partInsert :: Ord a => a -> a -> Partition a n ->
              Partition a (S n)
partInsert pivot val (MkPart left right)
  | val < pivot = mkPartL val left right
  | otherwise   = mkPartR val left right

insertN :: Ord a => Vect a m -> a -> Partition a n ->
           Partition a (m :+ n)
insertN VNil _ part = part
insertN (VCons x xs) pivot part
         = let part' = insertN xs pivot part
           in partInsert pivot x part'

partition :: Ord a => Vect a m -> a -> Partition a m
partition vec pivot
      = (insertN vec pivot (MkPart VNil VNil))
        `rewrite` (plus_Z :: EQ2 m n (m :+ Z) m)
```

Putting it all together, the `qsort` function is defined as follows:

```
qsort :: Ord a => Vect a m -> Vect a m
qsort VNil = VNil
qsort (VCons x xs) =
 case partition xs x of
  MkPart (l::Vect a l1) (r::Vect a r1)
       -> (app (qsort l) (VCons x (qsort r)))
             `rewrite` (plus_nSm
               :: EQ2 l1 r1 (l1 :+ S r1) (S l1 :+ r1))
```

Full dependent types approach

We have already seen some simple types in IDRIS declared using the Haskell style syntax, e.g. unary natural numbers:

```
data Nat = Z | S Nat;
```

Dependent types are declared using GADT-style syntax, using # as the type of types[4]. The indices are not restricted to being types, however — unlike with GADTs, we can use *data* constructors such `Z` and `S`. e.g. Vectors:

```
data Vect : # -> Nat -> # where
   nil  : Vect a Z
 | cons : a -> (Vect a k) -> (Vect a (S k));
```

For the type of `Partition`, we can use an ordinary function `plus` to give the length of the pair of vectors. It is convenient to be able to lift `plus` directly into the type, so giving a strong and machine checkable link between partitions, vectors and their sizes:

```
data Partition : # -> Nat -> # where
   mkPartition : (left:Vect a l) ->
                 (right:Vect a r) ->
                 (Partition a (plus l r));
```

In adding a value to a partition, we need to decide whether the value goes in the left or right of the partition, depending on a pivot value. We encounter the same difficulties as with the GADT and type family approaches in that the type of the resulting partition will also vary according to whether we insert into the left or right part of the partition:

- Insertion on the left,
  `mkPartition (cons x xs) ys : Partition (plus (S l) r)`
- Insertion on the right,
  `mkPartition xs (cons x ys) : Partition (plus l (S r))`

To make it easy to write a well-typed partition function, it is therefore necessary to be able to explain to the typechecker that these are really the same length. We achieve this by using a type rewriting function and a lemma to show that adding a successor to the second argument is equivalent to adding a successor to the first. These are provided by the standard library:

```
plus_nSm : ((plus n (S m)) = (plus (S n) m)));
rewrite  : {A:B->#} -> (A m) -> (m = n) -> (A n);
```

We can then write a function which inserts a value into the right of a partition, but which rewrites the type so that it is the same as if it were inserted into the left:

```
mkPartitionR : a -> (Vect a l) -> (Vect a r) ->
         (Partition a (plus (S l) r));
mkPartitionR x left right
    = rewrite (mkPartition left (cons x right))
              plus_nSm;
```

[4]The reason for choosing # instead of the more conventional $\star$ is to avoid syntactic conflict with the multiplication operator!

Now that insertion into the left and right have the same type, it is simple to write an insertion function based on a pivot and a comparison function:

```
partInsert: (lt:a->a->Bool) -> (pivot:a) -> (val:a) ->
            (p:Partition a n) -> (Partition a (S n));
partInsert lt pivot val (mkPartition left right)
   = if lt val pivot
         then mkPartition (VCons val left) right
         else (mkPartitionR val left right);
```

Length-preserving partitioning then simply iterates over a vector, inserting each element into the partition.

```
partition : (lt:a->a->Bool)->(pivot:a)->
            (xs:Vect a n)->(Partition a n);
partition lt pivot VNil = mkPartition VNil VNil;
partition lt pivot (VCons x xs)
    = partInsert lt pivot x (partition lt pivot xs);
```

To complete a `qsort` program, we will need to glue two partitions back together, along with their pivot.

```
qsort : (lt:a->a->Bool)->(Vect a n)->(Vect a n);
qsort lt VNil = VNil;
qsort lt (VCons x xs) = glue lt x (partition lt x xs);
```

The `glue` function takes a partition and the pivot, and reassembles them into a list. Again, because we insert the pivot at the start of the right list, we will need to rewrite the type, since `plus` is defined by recursion on its first argument:

```
glue : (lt:a->a->Bool)->
       a -> (Partition a n) -> (Vect a (S n));
glue lt val (mkPartition left right)
  = let lsort = qsort lt left,
        rsort = qsort lt right in
    rewrite (append lsort (VCons val rsort)) plus_nSm;
```

The complete IDRIS program we have outlined here, including test cases, is available online[5]. The approach we have taken here is similar to that for a conventional functional `qsort`. We need to define an intermediate type, `Partition`, to ensure the total length of the partition is the same as the length of the vector being partitioned, but otherwise we can program in the usual functional style.

Discussion

We initially attempted to use GADTs, a small extension to Haskell, to represent the relatively simple invariant that sorting a list preserves size. However, this

[5] http://www-fp.cs.st-and.ac.uk/%7Eeb/darcs/Idris/samples/partition.idr

approach ran into difficulties since there is no obvious way to do verifiable equational reasoning with GADTs. A further extension to Haskell, type families, is sufficient to verify this invariant. The full dependent type version in IDRIS is defined in a similar way to the type family version, indexing lists by their length and rewriting the type to insert into the right of a partition. The key conceptual difference is that with full dependent types, data may appear in a type or a value, whereas the Haskell version with type families maintains a strict separation between types and values.

Size, represented by natural numbers, is a common invariant on many data structures and it is therefore important to be able to represent and manipulate sizes effectively. Using sizes we can, for example, represent the depth of a tree structure or guarantee that a tree is balanced. However, we would sometimes like to represent more sophisticated invariants. In the case of `qsort`, a fully guaranteed version would not only maintain the length invariant, but also that the output is ordered and is a permutation of the input.

29.5 PERMUTATIONS

To show the flexibility of our fully dependently typed approach, let us consider how we can represent list permutations. Informally, an empty list is a permutation of an empty list. A non-empty list is a permutation of another list, if its tail is a permutation of that other list with its head removed. We revert to standard polymorphic lists, without the size index, since lists which are permutations of each other are naturally the same size:

```
data List a = Nil | Cons a (List a);
```

We need a predicate to represent list membership:

```
data Elem : A -> (List A) -> # where
   now   : {x:A} -> {xs:List A} -> (Elem x (Cons x xs))
 | later : {x:A} -> {ys:List A} -> {y:A} ->
           (Elem x ys) -> (Elem x (Cons y ys));
```

We can use this predicate to remove an element from a list safely. The predicate is effectively an index into the list, with `now` indicating the element is at index zero, and `later x` indicating the element is at index $x+1$. Removing an element is then by induction over the index. We also match on the (implicit) argument `xs`:

```
remove : {x:A} -> {xs:List A} -> (Elem x xs) -> (List A);
remove {xs=(Cons x ys)} now = ys;
remove {xs=(Cons x (Cons y ys))} (later p)
       = Cons x (remove p);
```

We can then represent permutations as a predicate on two lists, so making our informal description above precise:

```
data Perm : (List A) -> (List A) -> # where
   PNil  : Perm {A} Nil Nil
 | PCons : {x:A} -> {xs:List A} -> {xs':List A} ->
           (e:Elem x xs') -> (Perm xs (remove e)) ->
           (Perm (Cons x xs) xs');
```

Full dependent types allow us to write such predicates in terms of the actual data structures, and index them in terms of arbitrary functions such as `remove`.

In our previous definition, partitions were indexed over their total length. We now need to ensure that in building the partition we maintain knowledge about how the permutations are built. In order to achieve this, we carry the left and right lists in the partition *in the type*, and require a proof that inserting an element into the partition maintains the permutation.

```
data Partition : (l:List A) -> (r:List A) ->
                 (xs:List A) -> # where
   nilPart : Partition Nil Nil Nil
 | lCons   : {x:A} -> {xs,ys,zs:List A} ->
             (Partition xs ys zs) ->
             (Perm (Cons x (app xs ys)) (Cons x zs)) ->
             (Partition (Cons x xs) ys (Cons x zs))
 | rCons   : {x:A} -> {xs,ys,zs:List A} ->
             (Partition xs ys zs) ->
             (Perm (app xs (Cons x ys)) (Cons x zs)) ->
             (Partition xs (Cons x ys) (Cons x zs));
```

This is indexed over both sublists and the result, which allows us to maintain all the information we need throughout the structure. However, we cannot know in advance what the sublists will be given the original list, so we wrap this in a less informative type indexed only over the original list:

```
data Part : (xs:List A) -> # where
   mkPart : {ls,rs,xs:List A} -> (Partition ls rs xs) ->
            (Part xs);
```

Then our partition function has the following type, expressing that the partition arises from the input list. The types of `Part` and `Permutation` ensure that we will be able to extract a proof that the resulting list is a permutation of the original.

```
partition : (lt:a->a->Bool) -> (pivot:a) ->
            (xs:List a) -> (Part xs);
```

We can always extract a permutation proof from a partition:

```
getPartPerm : {xs,ys,zs:List A} ->
    (Partition xs ys zs) -> (Perm (app xs ys) zs);
getPartPerm nilPart = PNil;
getPartPerm (lCons part perm) = perm;
getPartPerm (rCons part perm) = perm;
```

Building a partition requires an explanation of what to do when inserting into the left and right parts of the partition, as before. We write a helper function `rConsP` so that insertion into the left and right parts of a partition have the same type:

```
rConsP : {xs,ys,zs:List A} ->
         (Perm (app xs ys) zs) ->
         (Perm (app xs (Cons x ys)) (Cons x zs));
rConsP {xs=Nil} p = PCons now p;
rConsP {xs=Cons w ws} {ys} {zs=Cons z' zs} (PCons e p)
      = PCons (later e) (consRPerm p);
```

We use `rConsP` to show that adding an element to the right list and then appending maintains a permutation with adding an element to the left of the whole list. This allows us to write `partInsert`:

```
partInsert : {xs:List a} ->
             (lt:a->a->Bool) -> (pivot:a) -> (x:a) ->
             (p:Part xs) -> (Part (Cons x xs));
partInsert lt pivot val (mkPart p)
   = if lt val pivot
        then (mkPart (lCons p (PCons now (getPartPerm p))))
else (mkPart (rCons p (rConsP (getPartPerm p))));
```

The partitioning function itself has a very similar definition to the size preserving version (only the `Nil` case and the type differ):

```
partition : (lt:a->a->Bool)->(pivot:a)->
            (xs:List a)->(Part xs);
partition lt pivot Nil = mkPart nilPart;
partition lt pivot (Cons x xs)
   = partInsert lt pivot x (partition lt pivot xs);
```

This definition (and the corresponding definition of `glue` which we omit here) requires a certain amount of reasoning about the construction of permutations. This is to be expected — the more static information we want to express, the more we have to explain to the type checker.

Comparison with GADTs and Type Families

The above definition gives even more static guarantees about our sort function — not only is it size preserving, but it also guarantees that the output is a permutation of the input. We could, potentially, go further and guarantee that the output is ordered, if we needed such a guarantee and were prepared to do the required reasoning. Although it is conceivable that such proofs could also be done using a combination of GADTs, type families and type classes, some technical problems would need to be overcome first:

- Lists are polymorphic, and we are using lists both as data *and* as an index to partitions. When we restricted the invariant to list length, the length was

treated independently of the data. Permutations, however, are directly linked to the real, polymorphic data in a fundamental way. It is not clear how to represent this conveniently if a strict separation between types and values is to be maintained.

- Although we have used it in the index to a type, the `remove` function operates on data. It may be possible to write this function with type families, but not generically for use on types and data.

While we do not make the claim that a permutation preserving quicksort is *impossible* in (extended) Haskell, we do believe that once invariants become sufficiently complex, they also become too difficult to express easily. In particular, it is not obvious how to express types which themselves depend on dependent types. There are fairly simple examples of such types, e.g. in the implementation of an interpreter, an environments of well-typed, well-scoped values could depend on a sized list of types as described in [3].

29.6 RELATED WORK

In this paper we have used a new experimental dependently typed language, IDRIS, which is closely related to EPIGRAM [14, 7] and AGDA [15], but which differs in that it is intended not primarily as a theorem prover, but as a platform for practical programming with dependent types. In IDRIS, we have the full flexibility of dependent types, but also allow conventional polymorphic functional programming.

An advantage of dependent types is that, in linking a datatype's representation directly with its meaning, we can provide strong static guarantees about the underlying data. In some of our own previous work [4, 6] we have used this to show properties of complex functions in a theorem proving environment [2]. We do not always want or need total correctness, however, and checking of "lightweight" invariants in a partial language may be sufficient in many cases. A number of recent approaches to lightweight static invariant checking have been based on small extensions to existing languages [16, 10] or use of existing features [13, 12]. As we have seen, such approaches are, however, usually limited in their expressivity. Recent work by Schrijvers et al. [17] describes an extension of Haskell with restricted type-level functions which improves the level of expressivity, but still requires separate implementations at the type and value levels. While we can express the partition example with this extension, we anticipate difficulties encoding properties of parametrised types such as Lists.

The languages Ωmega [18, 19] and Concoqtion [9] can to some degree handle type-based certificates of list lengths in programs. In Ωmega normalisation of type-level functions is carried out by left-to-right rewrite rules, but in contrast to Haskell this is done in an unrestricted manner, leaving termination issues the responsibility of the programmer. In Concoqtion, checking dependent types relies on an application of the proof checker Coq [8] to type level constraints which are integrated in the MetaOCaml language. Both Ωmega and Concoqtion have

independent languages of program expressions and type terms which forces the programmer to repeat part of what he or she had expressed at the expression level at the type level. As types become more complex, and especially when types depend themselves on parametrised or dependent types (such as our list permutation example), this becomes increasingly difficult to express.

29.7 DISCUSSION

The direction being taken in the development of GHC is encouraging, especially the recent experimental addition of type families. The introduction and development of GADTs, and more recently type families, allows increasing expressivity in the type system, which, in turn, allows several invariants to be verified by the typechecker. In the short term, this is an effective way of introducing some of the benefits of dependent types to functional programmers, by incorporating them into a familiar system with familiar (and extensive) libraries.

However, as demonstrated by the example of list partitioning, this general methodology has clear limits. Using only GADTs, we run into difficulties when manipulating even simple equations. Type families improve the situation, but as the invariants we wish to maintain become more complex (as, for example, with list permutations) even these present some difficulties which do not arise in a full dependently typed system. However lightweight or small the invariant we wish to keep may seem at first, there is always a possibility that we may run into some limitation which requires a more expressive type system, or a *property* which can be represented more easily in a more expressive type system. For this reason, we believe that in the longer term, new languages with full dependent types offer a more promising approach to writing programs which satisfy machine checkable invariants.

29.8 CONCLUSION

We have taken a relatively simple and well-known program, *quicksort*, and considered how to represent some invariants; firstly that the length of the input is preserved in the output, and secondly that the output is a permutation of the input. We consider these to be lightweight invariants in the sense that they do not *guarantee* total correctness of the implementation, but they do *verify* some important aspects. With full dependent types, we can protect against several potential programming errors in this way, and the overhead for the programmer is relatively small. There are two places in the size-preserving example where a `rewrite` must be made to show the typechecker that length is preserved. If the programmer has sufficient understanding of the function, such rewrites are easy to define.

In contrast, implementing the algorithm with GADTs has proved difficult, and to finish the implementation in Haskell required an additional Haskell extension, type families. The implementation required not only understanding of the algorithm to be implemented, but also details of an increasingly complex type system.

We believe that if dependent types are to be taken seriously in practical programming, and if programmers are to be encouraged to increase confidence in program correctness through the type system, there *must* be more support at the type level for reasoning about invariants. While there are obvious short-term advantages of adding less powerful type dependency to existing systems (namely, the existing code base and extensive library support Haskell offers), we believe that in the long term full dependent types will allow the programmer the greatest flexibility and expressivity. Full dependent types give us the flexibility to write the program with minimal invariant checking (i.e., the output merely has the same element type as the input), lightweight checking (i.e., the output shares some property with the input such as length or permutation) or, potentially, a full specification (the output must also be ordered). With each additional invariant we need to consider how to represent the invariant in the intermediate structure, and additional reasoning is required. Nevertheless, the computational content remains the same, as static properties such as types and proofs can be erased at run-time [1].

29.9 FURTHER WORK

There are several theoretical details we have not yet explored or only briefly touched on in our comparison of the full dependent type approach with the various Haskell extensions. For example, we have not mentioned decidability of type checking, termination analysis or the presence of side effects in types. If we allow arbitrary expressions in types, it is possible that type checking will not terminate. While this is already a possibility in Haskell with certain extensions, it is still important to consider. One approach may be to allow only structurally recursive functions on strictly-positive datatypes to reduce at compile-time.

Such theoretical considerations will, ultimately, be important in an industrial-strength dependently-typed programming language. Our goal with IDRIS, however, is to explore the practical possibilities of programming with full dependent types. We are already beginning to explore domain-specific language implementation for concurrency [5], and we are integrating the type system of IDRIS into Hume [11] in order to investigate functional correctness of embedded systems.

ACKNOWLEDGEMENTS

We would like to thank James Caldwell for several fruitful discussions, and the anonymous reviewers for their valuable comments. This work is generously supported by EPSRC grant EP/C001346/1.

REFERENCES

[1] Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.

[2] Edwin Brady. Ivor, a proof engine. In *Proc. Implementation of Functional Languages (IFL 2006)*, volume 4449 of *LNCS*. Springer, 2007.

[3] Edwin Brady and Kevin Hammond. A Verified Staged Interpreter is a Verified Compiler. In *Proc. ACM Conf. on Generative Prog. and Component Engineering (GPCE '06), Portland, Oregon*, 2006.

[4] Edwin Brady and Kevin Hammond. A dependently typed framework for static analysis of program execution costs. In *Implementation of Functional Languages 2005*. Springer, 2006.

[5] Edwin Brady and Kevin Hammond. Correct by construction concurrency, 2008. In Preparation.

[6] Edwin Brady, James McKinna, and Kevin Hammond. Constructing Correct Circuits: Verification of Functional Aspects of Hardware Specifications with Dependent Types. In *Trends in Functional Programming*. Intellect, 2007.

[7] James Chapman, Thorsten Altenkirch, and Conor McBride. Epigram reloaded: a standalone typechecker for ETT. In *TFP*, 2005.

[8] Coq Development Team. The Coq proof assistant — reference manual. `http://coq.inria.fr/`, 2001.

[9] Seth Fogarty, Emir Pasalic, Jeremy Siek, and Walid Taha. Concoqtion: Indexed types now! In *ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation*, 2007.

[10] Jeremy Gibbons, Meng Wang, and Bruno C. d. S. Oliveira. Generic and index programming. In *Draft proceedings of TFP 2007*, 2007.

[11] Kevin Hammond and Greg Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. Conf. Generative Programming and Component Engineering (GPCE '03)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[12] Oleg Kiselyov and Chung-Chieh Shan. Lightweight static capabilities. In *PLPV 2006*, 2006.

[13] Conor McBride. Faking it – simulating dependent types in Haskell. *Journal of Functional Programming*, 12(4+5):375–392, 2002.

[14] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[15] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[16] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple Unification-Based Type Inference for GADTs. In *Proc. ICFP '06: 2006 International Conf. on Functional Programming*, 2006.

[17] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *ICFP 2008*, 2008. To appear.

[18] Tim Sheard. Languages of the Future. In *Object Orientated Programming Systems, Languages and Applications*. ACM, 2004.

[19] Tim Sheard. Type-level computation using narrowing in Ωmega. In *Programming Languages meets Program Verification (PLPV 2006)*, Electronic Notes in Theoretical Computer Science, URL:`www.elsevier.com/locate/entcs`, 2006. Elsevier.