Case of (Quite) Painless Dependently Typed Programming: Fully Certified Merge Sort in Agda

Ernesto Copello, Álvaro Tasistro and Bruno Bianchi

Universidad ORT Uruguay
{copello, tasistro}@ort.edu.uy, bgbianchi@gmail.com,
WWW home page: http://docentes.ort.edu.uy/perfil.jsp?docenteId=2264

Abstract. We present a version of merge sort, fully certified, in the language Agda. It features: syntactic warrant of termination (i.e. no need of explicit termination proof), no proof cost to ensure that the output is sorted, and an inexpensive proof that the output is a permutation of the input.

1 Introduction

Programming is an activity which consists in:

1. Taking on a problem (technically named the *specification*) and
2. writing code to solve it.

Therefore, the delivery of every program implies an assertion, namely that the code meets the specification. For most programmers it is desirable to have access to technology that helps disclosing as many inconsistencies in such assertions as possible, as early as possible in the course of program development. Any such technology will demand to declare at least some aspect of the programmer's intention, i.e. of the specification, so that it be able to check the code against such declaration.

Type systems constitute one very successful technology of this kind, in which (aspects of) specifications are declared as types. *Dependent* type systems, in particular, make it possible to declare functional specifications in full detail, so that type checking entails actual logical correctness of the code. In other words, such languages can be said to feature an appealing catchword: *If it compiles, it works.* Now, the fact is that what actually compiles in these cases is not just the executable code, but this together with additional *mathematical* code, i.e. *formal proof* that the executable code matches the specification. This is due to the fact that it is impossible in general to automatically prove that some given executable code satisfies an arbitrary specification. Therefore it can be said that these systems turn, at least in principle, Programming into *fully formalized* Mathematics. Actually, the languages arisen from this trend are functional programming languages, for example Agda [Nor07] or Idris [Bra13], derived in

general from (Constructive) Type Theory, a system of logic intended for the full formalisation of (constructive) Mathematics.

For some practitioners the latter is indeed a very welcome feature, i.e. one allowing to enhance program construction with the composition of explicit foundation which is, in addition, automatically checked for correctness. But for most programmers the cost is rather too expensive, both in terms of instructional and actual work load. One specific difficulty with dependently typed programming concerns the need to ensure the termination of every computation, which is a requisite if the language is to feature decidable type checking and consistency as a language wherein to do Mathematics as suggested above. This requirement necessarily places some restriction as to the forms of recursion allowed in the languages, which affects the naturalness of programming or increases the cost of production, due to the necessity of often burdensome termination proofs.

As a consequence of the former circumstance, the largest part of the functional programming community has, for most of the time, not paid serious consideration to the alleged importance of dependently typed programming languages, being content with the service provided by various extensions of Hindley-Milner's type system. Now, as it happens, these extensions have, however, eventually begun to move towards incorporating dependently typed features. One first example is the *generalized algebraic data types* and, more recently, the use of Haskell's *type classes* as predicates and relations on types, giving rise to a kind of logic programming at the type level. As a consequence, some interest has awakened in the Haskell community concerning the power of the very much extended Haskell's type system for carrying out dependently typed programming.

A quite representative example of the mentioned investigations is [LM13], where the system of classes and kinds of Haskell is used to program a partially certified *merge sort* program. The result is quite satisfactory as the correctness properties considered are ensured in an almost totally *silent* way, i.e. without need to mention proof. But, at the same time, the certification is partial, only providing for the ordering of the output list.

Our investigation in this paper concerns the development of the merge sort example in Agda, a language *designed* to be dependently typed. We consider the problem of full certification, i.e. we show that the list resulting from merge sort is sorted and a permutation of the input and that the program terminates on every input. As a result, we are able to assert that:

1. Ensuring the sorted character of the output is achieved at the cost of minute proof obligations that are in all cases automatically discharged.
2. Ensuring that the output list is a permutation of the input requires a very low proof cost, due to a carefully chosen formalisation of the required condition.
3. Termination can be ensured syntactically by the use of Agda's *sized types*. This is quite significant, as recursion in this algorithm is general well-founded, i.e. not structural, which makes it a priori not checkable by a purely syntactic mechanism.

The rest of the paper begins by the latter feature above, i.e. by explaining the *sized types* of Agda and showing how to use them for transforming

a general well-founded form of recursion into a structural one. Next to that we turn to discussing the specification of the sorting problem and the certification of merge sort with respect to it. We end up in section 4 with conclusions. As already said, the code shown is Agda and can be fully accessed at `http://docentes.ort.edu.uy/perfil.jsp?docenteId=2264`.

2 Sized Types

Introduction. Consider the following versions of subtraction and quotient on natural numbers:

```
minus : ℕ → ℕ → ℕ
minus 0 y = 0
minus x 0 = x
minus (suc x) (suc y) = minus x y


div : ℕ → ℕ → ℕ
div 0 y = 0
div (suc x) y = suc (div (minus x y) y)
```

The latter uses a form of non-structural recursion. It actually works under some preconditions, namely $\mathsf{div}\,x\,y$ yields the expected result if $y \neq 0$ and either $x \neq 0$ or $x \geqslant y$. Specifically, if $y = 0$ the computation is terminating with result x —which is rather arbitrary, to say the least. It might therefore be deemed a very peculiar version of the quotient operation, but it will anyhow serve our purpose of explaining how the mechanism of sized types can be used to achieve syntactic control of the well-foundedness of some recursion forms.

The reason why the recursion in div is well-founded is that the first argument decreases strictly when passing on to the recursive call. Indeed, $\mathsf{minus}\,x\,y < \mathsf{suc}\,x$ (even if $y = 0$). We can make this recursion structural by redesigning the *type* of the first argument to div so that it carries appropriate information about its value. If we could for instance express that the parameter x in the recursive equation has i as an upper bound, then we would get that:

1. $\mathsf{suc}\,x$ has upper bound $\mathsf{suc}\,i$.
2. $\mathsf{minus}\,x\,y$ has upper bound i.

The type of x would in such case be something like $\mathsf{NatLt}\,i$ (in words: *Natural numbers less than* i) and the type of div ought to be:
$(i : \mathbb{N}) \to (\mathsf{NatLt}\,i) \to \mathbb{N} \to \mathbb{N}$.
The important point is the appearance of i as a further parameter: Indeed, calling div will require to explicitly pass i, i.e. the upper bound to the subsequent parameter. Then the recursive equation that we are looking at will, when adequately rewritten, expect an upper bound $\mathsf{suc}\,i$ in the pattern on the left and effect the recursive call on just i on the right, i.e. a *structural* decrease of the parameter. Let us now write the details down:

We start by introducing the natural numbers *with an upper bound*:

```
data NatLt : ℕ → Set where
  zero : (ι : ℕ) → NatLt (suc ι)
  succ : (ι : ℕ) → NatLt ι → NatLt (suc ι)
```

Now, to begin with, minus is rewritten as follows:

```
minus : (i : ℕ) → NatLt i → ℕ → NatLt i
minus _       x          0 = x
minus .(suc i) (zero i)   _ = zero i
minus .(suc i) (succ i x) (suc y) = upcast i (minus i x y)
```

The first equation considers the case in which the subtrahend is 0. Otherwise, we proceed by pattern matching on the minuend, i.e. the second parameter. Now, it is a general phenomenon that pattern matching in the presence of dependent types induces patterns on parameters other than the one being considered. For instance, in the two cases of patterns of the second parameter of minus, the first parameter —i.e. the upper bound— cannot but be suc i. This *must* be specified when writing the equations: What would otherwise be a non-linear pattern becomes in Agda a so-called *dotted* pattern, i.e. one used exclusively for the purpose of well-formation checking.

The last equation makes use of a *casting* function upcast: Indeed, the recursive call to minus yields a result of type NatLt i, but what we need to return is one of type NatLt (suc i). The div function becomes:

```
div : (i : ℕ) → NatLt i → ℕ → ℕ
div .(suc i) (zero i) _ = zero
div .(suc i) (succ i x) y = suc (div i (minus i x y) y)
```

and, as announced above, as we proceed on the second argument the recursive call is *structurally* decreasing on the first (i.e. the upper bound).

Now, as it happens, the information about the bound of the arguments of type NatLt can actually be inferred from the corresponding constructors. This inference facility is what constitutes the *sized types* feature of Agda: There is to begin a type Size, which is essentially ℕ with a constructor ↑ for "successor". Then one can declare e.g. the *sized* natural numbers as follows:

```
data SNat : {ι : Size} → Set where
zero : {ι : Size} → SNat {↑ ι}
succ : {ι : Size} → SNat {ι} → SNat {↑ ι}
```

In Agda, curly braces around parameters indicate that these are optional or *implicit*, i.e. that they can be omitted and shall then be inferred by the type-checker in function calls. Therefore the programmer may omit size information whenever this is unimportant. Besides, sized types admit *subtyping*, i.e. every expression of a type of size i —say $\alpha\{i\}$— is also of type $\alpha\{\uparrow i\}$ and, transitively,

of every other instance of α of greater size. Using sized naturals the final code for minus and div is the same as the one we began with but, unlike this, it passes Agda's termination check:

```
minus : {ι : Size} → SNat {ι} → SNat → SNat {ι}
minus zero y          = zero
minus x    zero       = x
minus (succ x) (succ y) = minus x y


div : {ι : Size} → SNat {ι} → SNat → SNat {ι}
div (zero) y = zero
div (succ x) y = succ (div (minus x y) y)
```

Merge sort. We shall now use sized lists to write a version of merge sort whose termination is certified by Agda in a purely syntactic manner, i.e. without having to produce a termination *proof*. Merge sort works in two parts: It first splits the given list into two –which we shall call deal– and then merges the recursively sorted permutations of the latter into the final sorted list –which we shall call merge. Direct encoding of this algorithm using ordinary lists will not pass Agda's termination check, as the recursive calls take on the components of the result of the deal function, which are lists with no structural relation with the original list. Lists are defined in Agda in the following way:

```
data List : Set where
[] : List
_::_ : (x : A) (xs : List) → List
```

The type A of the members of the list is a parameter to the whole module or theory. It comes equipped with a *total* relation $\leqslant$ which is what is required to perform list sorting. In the code below we add size parameters to the previous definition. As can be seen, in Agda it is possible to (re)use the former, ordinary list constructors:

```
data ListN : {ι : Size} → Set where
[] : {ι : Size} → ListN {↑ ι}
_::_ : {ι : Size} → A → ListN {ι} → ListN {↑ ι}
```

In general, it is convenient to program functions converting back and forth between a type and its sized version, which is readily done. We shall in the following make use of the forgetN function going from sized to ordinary lists. Next we introduce the code of the deal function. Notice that the size annotations certify that the function does not increase the size of the input in any of the resulting lists:

```
deal : {ι : Size} (xs : ListN {ι}) → ListN {ι} × ListN {ι}
deal [] = ([], [])
```

```
deal (x :: []) = (x :: [], [])
deal (x :: y :: xs) with deal xs
... | (ys, zs) = (x :: ys, y :: zs)
```

We go immediately on to present the merge sort, so that it can be seen that the resulting code is indeed simple and quite natural:

```
mergeSort : {ι : Size} → ListN {↑ ι} → ListN
mergeSort [] = []
mergeSort (x :: []) = x :: []
mergeSort (x :: y :: xs) with deal xs
... | (ys, zs) = merge (mergeSort (x :: ys)) (mergeSort (y :: zs))
```

As in the div example, the recursion has become structural on the implicit size parameter. Indeed, the pattern of the recursive equation is of size at most $\uparrow(\uparrow \iota)$ whereas the parameters to the recursive calls can be calculated of size at most $\uparrow \iota$. It could be objected that a more natural code would apply deal in the third case directly to the whole list, making later use of the fact that in such case the resulting lists are both of lesser length than the original one. But this fact cannot be captured syntactically, specifically by means of the sized types, for these cannot tell variations in the behaviour of the function at different inputs.

The merge function also needs to specify size information in order to automatically work out the termination, in spite of only performing structural recursive calls. The reason is that those calls are on alternate parameter places.

```
merge : {ι ι′ : Size} (xs : ListN {ι}) (ys : ListN {ι′}) → ListN
merge [] l = l
merge l [] = l
merge (x :: xs) (y :: ys)
with tot⩽ x y
... | .1 x⩽y = x :: merge xs (y :: ys)
... | .2 y⩽x = y :: merge (x :: xs) ys
```

The function tot⩽ is the one deciding the total relation $\leqslant$.

It can be concluded that certification of termination of this algorithm can be achieved at an indeed low cost, namely just the use of size annotations and a not at all unnatural encoding of the recursion.

3 Sorting

Ordered lists. In order to get a fully certified sorting program we must ascertain that the output list is an ordered permutation of the input. We start with the ordering issue. One method to accomplish the certification of a program with respect to a specification is simply to prove that the output satisfies the post-condition in every case in which the input satisfies the precondition. In our case, and considering only the ordering issue, this would amount to writing a function

$$\mathsf{lemma\text{-}sorted} \;:\; (\mathsf{xs} \;:\; \mathsf{ListN}) \to \mathsf{Sorted}\;(\mathsf{forgetN}\;(\mathsf{mergeSort}\;\mathsf{xs}))$$

for an adequately defined Sorted predicate on ordinary lists. A standard definition of the latter, relative to the $\leqslant$ relation assumed in the preceding section, is the following:

$$\frac{}{\mathsf{Sorted}\;[]}\;[\textsc{nils}] \qquad\qquad \frac{x \in A}{\mathsf{Sorted}\;[\mathsf{x}]}\;[\textsc{singls}]$$

$$\frac{x \leqslant y \qquad \mathsf{Sorted}\;(\mathsf{y} :: \mathsf{l})}{\mathsf{Sorted}\;(\mathsf{x} :: \mathsf{y} :: \mathsf{l})}\;[\textsc{conss}]$$

The corresponding Agda code is:

```
data Sorted : List → Set where
nils : Sorted []
  --
singls : (x : A)
    -- ---------
    → Sorted [x]
  --
conss : (x y : A) (ys : List) → x ⩽ y → Sorted (y :: ys)
    -- ------------------------------------------------
    → Sorted (x :: y :: ys)
```

A very similar method of certification would be to use what can be called *conditioned* types. If the problem is that of transforming input of type α satisfying a precondition P into output of type β standing in the relation Q to the input, then the function to be programmed is one of type

$(x : \alpha)(P(x) \to (\exists y : \beta)\, Q(x, y))$.

I.e. in our case:

$(xs : List) \to (\exists y : \beta)\, \mathsf{Sorted}\, y$.

Elements of type $(\exists x : \beta)\gamma$ are *pairs* formed by an object of type β and a proof that it satisfies γ.

The difference between proving lemma-sorted and programming a function of the latter type is minor. In the second case we get a function in which executable code is interspersed with proof code whereas the first corresponds to program verification. In either case we have to develop proof code that follows closely the (recursive) structure of the program or executable code.

Using what we are calling *conditioned* types is but *one* way of encoding the specification into the output type. That approach takes at least two inductive definitions, namely the one of the output data (i.e. the type β above) and that of the correctness property (predicate or relation) that must be satisfied. Now, as it happens, the encoding in question can often be accomplished in a more direct way by giving just *one* inductive definition that represents the data of type β that satisfy the required condition. Such definition will be of what would in conventional Mathematics be called a *part* of the βs, which in Type Theory

means βs exhibiting certain further feature or evidence. We call them for this reason *refined* data types or data structures.

Examples are of course the natural numbers NatLt with an upper bound or, generally, the sized types, both given in the preceding section. Notice for example that the type NatLt could be used to give a finer specification of the subtraction, i.e. one requiring the minuend to be not lesser than the subtrahend:

$(-) : (m : \mathbb{N}) \to \mathsf{NatLt}(\mathsf{suc}\, m) \to \mathbb{N}$.

One example of refined data structures is the *ordered* lists introduced in [AMM05] and used again in [LM13]. It is the type of ordered lists whose elements are bounded above and below by given values. We give here a definition tailored to our needs, which on the one hand does not make use of an upper bound[1] and, on the other, is sized:

```
data OList : {ι : Size} → Bound A → Set where
onil : {ι : Size} {l : Bound A}
  -- ------------------------
  → OList {↑ ι} l

:< : {ι : Size} {l : Bound A} (x : A) {l⩽x : LeB l (val x)} → OList {ι} (val x)
  -- ------------------------------------------------------------------------
  → OList {↑ ι} l
```

Observe the "cons" constructor :<. It requires first implicit arguments corresponding to the size and the lower bound to the members of the list. Next comes the head and, next to that, an implicit argument which is a proof that the lower bound given is indeed lesser than the head. Finally the tail must have the head as a lower bound, thus ensuring the sorted character of the entire list. In the implicit argument just mentioned, which stands between the head and the tail, we use the names LeB and val. The reason is connected to the use of the class Bound of types for the members of the lists. This is defined as follows:

```
data Bound (A : Set) : Set where
bot : Bound A
val : A → Bound A
```

i.e. it wraps up a type A together with an absolute lower bound bot. This is convenient in order to produce Olists without having to care for providing a precise lower bound thereof. That bot is indeed a minimum of Bound A in ensured by redefining the "order" relation:

```
data LeB : Bound A → Bound A → Set where
lebx : {b : Bound A} → LeB bot b
lexy : {a b : A} → a ⩽ b → LeB (val a) (val b)
```

[1] That would be necessary if we should make use of the append operation on the ordered lists. This is not either used in [LM13], but the authors do not bother to simplify the definition.

Using the forgetO function from ordered lists to plain lists we can prove the following, which may provide some additional reassurance to the definition of Olists.

```
lemma-sort : {l : Bound A} (xs : OList l) → Sorted (forgetO xs)
lemma-sort onil = nils
lemma-sort (:< x onil) = singls x
lemma-sort (:< x (:< y {lexy x⩽y} xs))
 = conss x y (forgetO xs) x⩽y (lemma-sort (:< y {lexy x⩽y} xs))
```

We next show the code of merge sort acting on sized Olists. The boxed parts constitute the additional code corresponding to proof obligations of the conditions related to the ordered character of the lists. The corresponding parameter places have been put into curly braces in an attempt to dispense with them as much as possible but, as it happens, they are actually required by Agda. In the case of merge sort the extra code is minimal, viz. the instantiation of a parameter standing for a proof of inequality. The gray highlighting indicates that Agda is able to supply the required proof object automatically. Therefore, we observe that programming the new version of merge sort entails no cost due to proof.

It should also be observed that deal is without change, as it was to be expected. This is due to the simple fact that it is able to continue acting on just sized lists, not necessarily ordered.

```
mergeSort : {ι : Size} → ListN {↑ ι} → OList bot
mergeSort [] = onil
mergeSort (x :: []) = :< x {l⩽x = l⩽x} onil
mergeSort (x :: y :: xs) with deal xs
... | (ys, zs) = merge (mergeSort (x :: ys)) (mergeSort (y :: zs))
```

As to merge, we get the following new code, with additions of proof obligations boxed and grayed:

```
merge : {ι ι' : Size} {l : Bound A} → OList {ι} l → OList {ι'} l → OList l
merge onil l = l
merge l onil = l
merge (:< x {l⩽x =l⩽x} xs)
  (:< y {l⩽x =l⩽y} ys)
  with tot⩽ x y
... | ._1 x⩽y = (:< x {l⩽x = l⩽x} (merge xs (:< y {l⩽x = lexy x⩽y} ys)))
... | ._2 y⩽x = (:< y {l⩽x = l⩽y} (merge (:< x {l⩽x = lexy y⩽x} xs) ys))
```

The two first occur in patterns and are therefore not proof obligations, but just parameters that have to be made explicit. The other four are automatically

solved by Agda. Therefore, there is no cost associated to proof construction. Finally, we apply the previosly presented lemma to the present merge sort algorithm in order to return an ordered list:

```
lemma-mergeSort-sorted' : (xs : ListN) → Sorted (forgetO (mergeSort' xs))
lemma-mergeSort-sorted' = lemma-sort ∘ mergeSort'
```

We observe that the certification of the sorted character of the output list is, if not totally silent as in [LM13], nearly so and, in any case, not more expensive, since the minute proofs are solved automatically by Agda.

The permutation relation. We now turn to the problem of ensuring that the output list is a permutation of the input. The point in this case is the choice of an appropriate formalisation of the specification. We begin by inductively defining a ternary relation whose instances will be written $xs/x \mapsto xs'$ and will hold when xs' is the result of removing an element x from an arbitrary position of the list xs. The following rules define this relation, establishing that a list element can be removed either from the head or from the tail of a list.

$$\frac{}{(x :: l)/x \mapsto l}$$

$$\frac{l/y \mapsto l'}{(x :: l)/y \mapsto (x :: l')}$$

We next encode this relation in Agda:

```
data _/_↦_ : List A → A → List A → Set where
removeFromHead : {x : A} {xs : List A}
  ---------------------------------------
  → (x :: xs) / x ↦ xs
removeFromTail : {x y : A} {xs ys : List A} → xs / y ↦ ys
  ---------------------------------------------------------
  → (x :: xs) / y ↦ (x :: ys)
```

The preceding relation is used in the following standard definition of the permutation relation $\sim$, which does not need a decidable equality:

$$\frac{}{[] \sim []}\,[\sim []]$$

$$\frac{l_1/x \mapsto l_3 \qquad l_2/x \mapsto l_4 \qquad l_3 \sim l_4}{l_1 \sim l_2}\,[\sim x]$$

The corresponding Agda code is:

```
data _∼_ : List A → List A → Set where
∼ [] : [] ∼ []
```

```
∼x : {x : A} {xs ys xs' ys' : List A}
  → (xs / x ↦ xs') → (ys / x ↦ ys') → xs' ∼ ys'
    -- ------------------------------------------
    → xs ∼ ys
```

Now we face a situation similar to the one in the preceding paragraph, i.e., in order to ensure that the merge sort returns a permutation of the input list, we could think of employing an appropriate refined type. The existence of the latter, however, seems unlikely: we would need to define inductively the lists that are permutations of a given list, and there exists a large variety of transformations between the two related lists that do not seem able to be captured by simple constructors. We therefore choose to employ a specification with conditioned types, i.e. produce an output consisting of both a list and a proof that it is a permutation of the input list. We know that the proof will follow the structure of the merge sort algorithm. Let us the look at this: First, deal returns the pair (ys, zs) of lists, which must be a partition of the original list xs. We might therefore think of proving that the concatenation of ys and zs is a permutation of xs, but this will introduce properties of the concatenation operation in our proof. We try to avoid such a detour by defining when a list is a permutation of a pair of lists, as follows:

```
data _∼p_ : List A → List A × List A → Set where
∼ [] r : (xs : List A) → xs ∼p ([], xs)
  --
∼ [] l : (xs : List A) → xs ∼p (xs, [])
  --
∼xr : {x : A} {xs ys xs' zs zs' : List A}
  → (xs / x ↦ xs') → (zs / x ↦ zs') → xs' ∼p (ys, zs')
    -- ------------------------------------------------
    → xs ∼p (ys, zs)
  --
∼xl : {x : A} {xs ys xs' ys' zs : List A}
  → (xs / x ↦ xs') → (ys / x ↦ ys') → xs' ∼p (ys', zs)
    -- ------------------------------------------------
    → xs ∼p (ys, zs)
```

The first two cases are trivial. The remaining two consider removing an element from the first list: Then the same element has to be removed from (exactly) one of the other two lists and the lists resulting of the removals must still be in the permutation relation. We can indeed prove that a list is a permutation of a pair of lists according to this definition if and only if the list is a permutation of the concatenation of the pair of lists in question.

Now, for the purpose of certifying the merge sort algorithm we do not need so much: We can do with a coarser relation, only allowing to remove elements from the head of the lists. The relation is clearly a *sound*, although not *complete* version of the permutation relation, but this is all we need for ensuring that

merge sort returns a permutation of the given input. The observation gives rise to the following definition:

```
data _∼p'_ : List A → List A × List A → Set where
∼[]r : (xs : List A) → xs ∼p' ([], xs)
∼[]l : (xs : List A) → xs ∼p' (xs, [])
∼xr : {x : A} {xs ys zs : List A} → xs ∼p' (ys, zs)
  -- ------------------------------
  → (x :: xs) ∼p' (ys, x :: zs)
∼xl : {x : A} {xs ys zs : List A} → xs ∼p' (ys, zs)
  -- ------------------------------
  → (x :: xs) ∼p' (x :: ys, zs)
```

The next lemma states that this restricted relation is indeed a sound version of the permutation relation, i.e. that it implies that the first list is a permutation of the concatenation of each pair to which it is related:

```
lemma∼p'∼ : {xs ys zs : List} → xs ∼p' (ys, zs) → xs ∼ (ys ++ zs)
```

It is used for proving the following, which is necessary later to establish the correctness of the algorithm:

```
lemma∼p' : {xs ys zs ws ys' zs' : List A} → xs ∼p' (ys, zs) →
  ys ∼ ys' → zs ∼ zs' → ws ∼p' (ys', zs') →
  xs ∼ ws
```

Now we go on to certify that the pair of lists returned by deal is related to the latter's input list by the ∼p' relation. We use the forgetN and forgetNp functions to erase size information. We choose to encode the specification fully in the output type of the function just for the sake of showing in parallel the composition of the algorithm and the proof. We could as well have proven the desired condition as a property of a purely executable code.

```
deal : {ι : Size} (xs : ListN {ι}) →
  Σ (ListN {ι} × ListN {ι}) (λ p → forgetN xs ∼p' forgetNp p)
deal [] = ([], []),
                ∼[]r []
deal (x :: []) = (x :: [], []),
                ∼[]l (x :: [])
deal (x :: y :: xs) with deal xs
... | (ys, zs), xs∼ys, zs = (x :: ys, y :: zs),
  ∼xl (∼xr xs∼ys, zs)
```

Again, because we have carefully selected the specification, we obtain an almost free proof. Proofs marked with grayed boxes were automatically solved, and only one proof must be given, corresponding to the case of a list with at least

two members. What this requires is to prove that $x :: y :: xs \sim$p' $(x :: ys, y :: zs)$ if $xs \sim$p' (ys, zs), which is easily seen to follow using the rules $\sim$xl and $\sim$xr of the $\sim$p' relation, as is indeed exhibited in the code's last line.

Now we encode the merge function, whose type specifies that given two ordered lists it returns another ordered list together with a proof that it is a permutation of (the pair of) the two originally given ones. Again, the proofs obligations marked in grey were automatically solved by Agda. We use pair projections π.

```
merge : {ι ι' : Size} {l : Bound A} (xs : OList {ι} l) (ys : OList {ι'} l) →
  Σ (OList l) (λ zs → forgetO zs ∼p' (forgetO xs, forgetO ys))
merge onil l = l,
               [∼[]r (forgetO l)]
merge l onil = l,
               [∼[]l (forgetO l)]
merge (:< x [{l⩽x = l⩽x}] xs)
  (:< y [{l⩽x = l⩽y}] ys)
with tot⩽ x y
... | ₁ x⩽y = (:< x [{l⩽x = l⩽x}] zs), ∼xl hi
  where zs = Π₁ (merge xs (:< y [{l⩽x = lexy x⩽y}] ys))
        hi = Π₂ (merge xs (:< y [{l⩽x = lexy x⩽y}] ys))
... | ₂ y⩽x = (:< y [{l⩽x = l⩽y}] ws), ∼xr hi
  where ws = Π₁ (merge (:< x [{l⩽x = lexy y⩽x}] xs) ys)
        hi = Π₂ (merge (:< x [{l⩽x = lexy y⩽x}] xs) ys)
```

As can be seen, only two proofs had to be provided, corresponding to the base cases. We end up giving the fully certified merge sort algorithm. Its type is a specification in the form of a conditioned Olist, requiring that it is a permutation of the input. The only proof that could not be automatically solved is a direct application of lemma presented above to the induction hypotheses, named $xs \sim ys, zs$ in the where clause, and the permutation results of merge and deal functions. The code could be more neat if we had chosen just to verify the algorithm instead of developing it together with its proof.

```
mergeSort : {ι : Size} (xs : ListN {↑ ι}) →
  Σ (OList bot) (λ ys → forgetN xs ∼ forgetO ys)
mergeSort [] = onil, [∼[]]
mergeSort (x :: [])
```

```
    = :< {l = bot} x {l⩽x = lebx} onil,
      ∼x removeFromHead removeFromHead ∼[]
mergeSort (x :: (y :: xs)) with deal xs
... | (ys, zs), xs∼ys,zs
  with mergeSort (x :: ys) | mergeSort (y :: zs)
... | ys', x::ys∼ys | zs', y::zs∼zs
  with merge ys' zs'
... | ws, ws∼ys,zs
    = ws,
      lemma∼p' (∼xl (∼xr (xs∼ys, zs)))
        x::ys∼ys'
        y::zs∼zs'
        ws∼ys', zs'
```

4 Conclusions

Dependently typed programming provides a framework for certified software development, i.e. one in which what compiles works. The price to pay for its use is increased cost in code production, since not only executable but also mathematical code (i.e. formal proofs) has to be produced. One specific difficulty arises in connection to program termination and the forms of recursion allowed: Totality is required in order to feature decidable type checking and, therefore, recursion has to be restricted. Normally the restriction is to structural recursion and, therefore, explicit proofs of termination have to be given for the general forms of recursion that are generally required. In this paper we have shown how all of these problems can be eased by the use of appropriate techniques and features of the languages involved. Specifically, we have considered *merge sort* to show how:

1. Agda's *sized types* can ensure termination of algorithms by converting some forms of general recursion into structural in a silent manner.
2. The use of *refined* types that incorporate certain semantic information in the inductive definition of appropriate (sub)classes of data can lead to almost silent certifications.
3. The cost of proofs can significantly be reduced by a *careful choice of formalisation* of the specification in question.

One purpose of ours was to compare programming in a language designed with dependent types with experiences that have recently been put forward using the by now very elaborated type system of Haskell. Specifically with respect to [LM13], we observe that:

1. They obtain a silent certification of merge sort with respect to the sorted character of the output list. We, on the other hand, obtain a not fully, but

nearly, silent version that is nevertheless without cost due to proof construction.
2. We are able to certify the termination of our version also without cost of proof, whereas the issue is not addressed in [LM13]. The price to pay in this case reduces to encoding the algorithm in a way that might not be the first choice for every programmer but that it is nevertheless clear and quite natural.
3. We are able to provide an inexpensive certification that the output is a permutation of the input, whereas the issue is not either addressed in [LM13].

It has to be said that it looks highly unlikely that the kind of dependently typed programming developed in the mentioned work is somehow able to cope with the last two issues just mentioned.

We also believe that our code is generally more elegant and comprehensible that the one obtained by pursuing the kind of logic programming given rise to by the hacking on the workings of Haskell's type classes constraint solver, even if, as we have done in this case, programs and proofs are composed as part of the same code. Actually we are ready to sustain that dependent type systems are much easier to learn and use, and indeed more natural, than the present type system of Haskell. Therefore we also prefer to pursue the expansion of the knowledge on genuine (i.e. by design) dependently typed programming rather than in intricate encodings thereof.

Possibly the main difficulty with dependently typed programming remains the restriction as to the allowable forms of recursion imposed by the needs of decidable type checking and of consistency of the internal logic via the isomorphism of propositions-as-types. Sized types have added to the alleviation of this matter, as we expect to have shown, but, of course, the problem is not fully solvable and there will always remain the necessity of developing termination proofs. The investigation of techniques to adequately formulate problems in ways that further facilitate the development of easily proven terminating programs is a matter worth pursuing. It is nevertheless important to mention that the consistency of the language as logic and the decidability of type checking are features derived from the restriction that are not enjoyed by the present expansion and form of use of Haskell's system of kinds and classes.

References

[AMM05] Thorsten Altenkirch, Conor McBride, and James McKinna, *Why dependent types matter*, http://www.e-pig.org/downloads/ydtm.pdf, 2005.

[Bra13] Edwin Brady, *Idris, a general-purpose dependently typed programming language: Design and implementation*, Journal of Functional Programming **23** (2013), 552–593.

[LM13] Sam Lindley and Conor McBride, *Hasochism: The pleasure and pain of dependently typed haskell programming*, SIGPLAN Not. **48** (2013), no. 12, 81–92.

[Nor07] Ulf Norell, *Towards a practical programming language based on dependent type theory*, Ph.D. thesis, 2007.