This is a peer-reviewed, accepted author manuscript of the following conference paper: de Muijnck-Hughes, J & Noble, J 2024, 'Colouring flags with Dafny & Idris', Paper presented at Dafny 2024, London, United Kingdom, 14/01/24.

Colouring Flags with Dafny & Idris

Jan de Muijnck-Hughes
University of Strathclyde
Glasgow, United Kingdom
jan.de-Muijnck-Hughes@strathclyde.ac.uk

James Noble
Creative Research & Programming
Wellington, New Zealand
kjx@programming.ac.nz

Abstract

Dafny and Idris are two *verification-aware* programming languages that support two different styles of fine-grained reasoning about programs. Dafny is an imperative *design-by-contract* language that provides a clear *separation* between specifications and code, while Idris is a dependently-typed functional language in which specifications *are* code. Each of these approaches support different styles of verification — Hoare Logic in Dafny versus Dependent Type Theory in Idris. In this paper, we will examine how Dafny and Idris express *The Problem of the Dutch National Flag* from Dijkstra's Discipline of Programming, and note the differences and similarities between both approaches.

Keywords: Idris, Dafny, Dependent Types, Verification

1 Introduction

Verification-aware programming languages support direct reasoning about programs within the structure of the language itself. Such languages distinguish themselves from *interactive theorem provers* in two ways: 1) the languages are geared towards practical programming just as much as they are towards verification, efficiency is just as important as correctness; and 2) not everything has to be mechanically verified — the 'verification dial' can be turned down and up as required.

Dafny and Idris are two verification-aware programming languages that support not only developing programs, but also mathematically verifying that those programs are correct. In spite of having essentially the same goals, the two languages are very different.

Dafny [26] is primarily imperative and object-oriented, based on languages like C♯, Eiffel, and Algol, and software engineering techniques such as *Vienna Development Methodology* (VDM) [9]. Dafny embodies *design-by-contract* [16, 17] — a Hoare logic modularised via function pre- and post-conditions, and relying on *demesnes* [31, 32] (aka *dynamic frames* [10, 11]) to manage aliasing in a mutable, garbage-collected heap. Dafny enforces a clear *distinction* between (mathematical) specifications and (imperative) programs.

Idris [5], in contrast, is a dependently-typed pure functional language in the lineage of Haskell and mathematical proof tools such as Coq [25], Agda [24], PVS [28], and NuPRL [27]. In Idris, specifications *are* code, and side effects such as mutation must be mediated by techniques like monads and effect handlers — very much unlike Dafny. Idris2 (the version discussed in this paper) incorporates *Quantitative Type Theory* (QTT) [3] to support type erasure and linear resources.

In this paper, we will examine how Dafny and Idris address a classical sorting problem, *The Problem of the Dutch National Flag*, attributed by E.W. Dijkstra to W.H.J. Feijen [8] and note the differences and similarities between both approaches. The next section presents the problem of the Dutch national flag, and Dijkstra's solution; the following section translates this solution into Dafny. We then present a more general Idris2 solution — a merge sort — as purely functional Idris is not well suited to Dijkstra's rather imperative framing of both problem and solution. We then compare the two languages and conclude by considering possible future work.

2 The Problem of the Dutch National Flag

The problem of the Dutch National Flag requires the design of an algorithm to sort a randomly allocated array of colours taken from the Dutch flag (Red, White, and Blue) in order according to the Flag itself (top-to-bottom; Red, White, then Blue). Dijkstra presents the problem as:

> There is a row of buckets numbered from *1* through *N*. It is given that
>
> *P1*: each bucket contains one pebble
>
> *P2*: each pebble is either red, white, or blue.
>
> A mini-computer is placed in front of this row of buckets and has to be programmed in such a way that it will rearrange (if necessary) the pebbles in the order of the Dutch national flag, i.e. in order from low to high bucket number first the red, then the white, and finally the blue pebbles. In order to be able to do so, the mini-computer has been equipped with one output command that enables it to interfere with pebble positions, viz.
>
> "buck:swap(i, j)" for $1 \leq i \leq N$ and $1 \leq j \leq N$:
>
> for $i = j$: the pebbles are left as they are
>
> for $i \neq j$: two computer-controlled hands pick up the pebbles from buckets *nrs. i* and *j* respectively and then drop them in each other's bucket respectively. (This operation leaves relations *P1* and *P2* invariantly true.)
>
> and one input command that can inspect the colour of a pebble, viz.
>
> "buck(i)" for $1 \leq i \leq N$: when the computer program prescribes the evaluation of this function of type "colour", a movable "eye" is directed upon bucket *nr. i*, and delivers to the mini-computer as the value of the function the colour (i.e. red, white, or blue) of the pebble currently lying in the bucket, the contents of which is inspected by the "eye".

where N is a "global constant from the context". He goes on to add three systemic requirements:

1. *degenerate inputs* — fewer than three colours, perhaps none if $N = 0$.
2. *storage* — only scalar variables, no arrays.

3. *efficiency* — “the program may direct the ’eye’ at most once upon each pebble”.

Dijkstra also offers the following solution to the problem, along with a rigorous informal English argument that the program correctly solves the problem:

```
begin glovar buck; glocon N; privar r, w, b;
  r vir int, w vir int, b vir int := 1, N, N;
  do w ≥ r →
  begin glovar buck, r, w, b; pricon col;
    col vir colour := buck(w);
    if col = red → buck:swap(r, w); r:= r + 1
     ▯ col = white → w:= w − 1
     ▯ col = blue → buck:swap(w, b); w, b := w − 1, b − 1
    fi
  end
  od
end
```

The key to Dijkstra’s solution is to divide the array into four regions: one for red pebbles; one for white; one region for pebbles we’ve yet to inspect; and finally one for blue pebbles — delineated by index variables *r*, *w*, and *b*. The whole array starts in the uninspected region (between the *w* and *b* indices), then each trip around the loop inspects the contents of the lowest uninspected bucket (at *w*), and then adjusts the index variables and swaps that pebble if necessary to place that pebble into the correct region. A key point of Dijkstra’s algorithm is that not only does it examine each pebble only once, it also performs at most one swap for each element of the array. *Program Proofs* [12] includes the following diagram of these regions:

3 Dafny Colours the Flag

Leino’s *Program Proofs* [12, § 15.0] includes a good description of a Dafny implementation of the algorithm, shown here in Figure 1, and the rationale for its proof.

The Dafny code is very similar to Dijkstra’s post-Algol / guarded commands code, with three main differences: the more traditional `while` and `match` statements rather than **do** and multi-way **if** statements; the `ensures` clauses giving the postconditions and `modifies` clause giving the frame; and the `invariant` clauses giving loop invariants.

The Dafny `DutchFlag` method has two postconditions. The first postcondition requires that the elements of the array must be ordered — the underlying ordering of the Color type ensures the red-white-blue order of the Dutch flag. The second condition requires that the output array is a permutation of the input array. Earlier versions of Dafny would also have required a precondition to state that the input array `a`

```
method DutchFlag(a: array<Color>)
  modifies a
  ensures forall i,j :: 0 <= i < j < a.Length
    ==> Ordered(a[i], a[j])
  ensures multiset(a[..]) == old(multiset(a[..]))
{
  var r, w, b := 0, 0, a.Length;
  while w != b
    invariant 0 <= r <= w <= b <= a.Length
    invariant forall i :: 0 <= i < r ==> a[i] == Red
    invariant forall i :: r <= i < w ==> a[i] == White
    invariant forall i :: b <= i < a.Length ==> a[i] == Blue
    invariant multiset(a[..]) == old(multiset(a[..]))
  {  match a[w]
       case Red =>
         a[r], a[w] := a[w], a[r];
         r, w := r + 1, w + 1;
       case White =>
         w := w + 1;
       case Blue =>
         b := b - 1;
         a[w], a[b] := a[b], a[w];
  }
}
```

Figure 1. Dafny version of Dijkstra’s solution to the Problem of the Dutch National Flag.

was actually an array, rather than a null pointer — recent versions of Dafny express that precondition implicitly in the types, because Dafny’s types are now non-null by default.

The heavy lifting — as is often the case — is done by the loop invariants. The first invariant describes the regions making up the core of the algorithm: the next three invariants characterise the colours of each of the output regions. The last invariant echoes the last postcondition, saying that the array processed so far is a permutation of the corresponding entries of the input array.

While Dijkstra discusses both postconditions and invariants in his presentation of the problem, he does not make either explicit *as integral parts of the program code*. Dafny obviously must make postconditions and invariants explicit — and also preconditions, although they are not required in this example. Dafny specifications are syntactically distinguished as special clauses on method definitions and loop statements respectively.

4 Idris’ Turn to Colour the Flag

Idris2 is a general purpose functional programming language with support for full-spectrum dependent types and QTT.

Dependent type theory is an expressive setting in which types can depend on values, supporting fine-grained type-level reasoning about our programs. Dependently typed languages, such as Idris [6], Agda [24], Lean [19], etc., epitomise the *Curry-Howard* correspondence of *propositions are types; proofs are programs.* Idris and Agda, however, present this

correspondence as part of a more typical programming setting than seen in more traditional theorem provers that support dependent types such as Lean and Coq [25]. With this approach, our programs *can become* intrinsically verified because the code we *can* write *is* the proof that the code is correct: *Correctness-by-Construction*.

QTT provides reasoning not only about how many times terms can be used (linearity) but also when terms can be used. We can use quantities to distinguish between code that we only need at compile time (to reason about sorting) from code we need to run (the sorting code itself), but that also the compile-time only code will influence the runtime code used.

As with many functional languages, in-place update algorithms (such as Quicksort or Dijkstra's algorithm for the Dutch national flag) are not always the most idiomatic ways to sort lists. Thus, when Colouring the flag in Idris, we will not attempt to adhere strictly to Dijkstra's requirements and we will produce a correct-by-construction *MergeSort* implementation adapted from *Type-driven development with Idris* [6, Chp. 10].

Note. Even though we are reasoning about MergeSort we could potentially use Views (Figure 2) to reason about in-place sorting [1], but that approach would require more consideration and thought.

4.1 Engineering with Proofs

Within the dependently-typed setting, datatypes are the specifications we want to reason about, and we construct programs that embody the proofs that these specifications are correct.

An (un)fortunate side effect of the dependently typed approach, however, is that our specification, code, and proofs are all one and the same. Such mixing means that, if we are not careful, code that is easier to reason about is not necessarily performant. We thus use two idiomatic approaches, that when combined with quantities, enables efficient reasoning about our programs: *Views*—for detailing how to traverse over our input lists more elegantly and aid in reasoning about our functions' totality (Section 4.1.1); and *Thinnings*—for detailing how we rearrange our input lists so that we know the output is a permutation of the input (Section 4.1.2).

4.1.1 Views. *Views* are a programming idiom first observed by Philip Wadler for Haskell in 1987 [29], in which datatypes (the views) are used to observe data and their relations to support more informed reasoning about the observed data. When combined with dependent-types, specifically dependent pattern matching [13], we have a more expressive programming idiom for reasoning about data in more interesting ways. For instance, views can not only better describe how our functions should operate, but they can also give sufficient evidence over the computational structure of our programs to reason about their totality.

Consider the example 'view & covering function' in Figure 2. Here we have defined a 'View' that compares the shape of two lists and if they are balanced (are equal in length) or not. `Balance` is an inductive datatype, and each constructor determines the valid comparisons. First are the base cases: both lists are empty; the left list has more elements; and the right list has more elements. Second, and final, case is the inductive step: examine the head of both lists, and then examine the tail. The 'covering function', `balance`, computes the view given two lists.

```
data Balance : (xs, ys : List a) -> Type where
  BalEmpty : Balance Nil Nil
  LeftLean : Balance (x::xs) Nil
  RyetLean : Balance Nil     (y::ys)
  Balanced : Lazy (Balance xs ys)
          -> Balance (x::xs) (y::ys)
```

(a) View

```
balance : (xs, ys : List a) -> Balance xs ys
balance [] [] = BalEmpty
balance [] (x :: xs) = RyetLean
balance (x :: xs) [] = LeftLean
balance (x :: xs) (y :: ys) = Balanced (balance xs ys)
```

(b) Cover

Figure 2. Example of 'View & Covering'.

4.1.2 Thinnings. *Thinnings*, derived from *Order Preserving Embeddings*, is another programming idiom to reason about data. Specifically, how data moves between lists such that we retain the relation between inputs and outputs [14, 7]. Moreover, thinnings establish relations between lists of different sizes, such as filtering lists and subset relations.

Consider, for example, Figure 3 that shows a standard thinning specifying how a given input list (`xs`) is distributed between two output lists that capture if a given predicate (`f`) holds. Much like our `Balance` datatype, `Thinning` is inductive, and the base case captures the empty list. The remaining cases capture how, as we traverse over the list, the elements of `xs` are kept and which are dropped depending on if the given predicate holds i.e. can be instantiated or not. We construct the thinning, as described in its specification, by traversing over the input, and using the provided decision procedure (the predicate must be decidable) to move copies of the element to the correct output lists. Further, a dependent pair promotes values (our evidence of which elements were kept and dropped) to the type level.

4.2 Type-Driven Merging of Lists

Using our idioms, we can now start to specify, and thus reason about, our merge sort algorithm by construction.

We begin with how to merge two lists. Figure 4 details a thinning and its implementation to specify how two lists are merged by comparison, an implementation of `mergeBy`. We make our specifications parametric by parameterising

```
data Thinning : (0          f : a -> Type)
             -> (  xs, ys, zs : List a)
                            -> Type
  where
    TEmpty : Thinning f Nil Nil Nil
    TKeep : {0 f : a -> Type}
         -> (prf : f x)
         -> (ltr : Thinning f     xs  ys     zs)
                -> Thinning f (x::xs) ys (x::zs)
    TSkip : {0   f : a -> Type}
         -> (prf : f y -> Void)
         -> (ltr : Thinning f     xs      ys  zs)
                -> Thinning f (x::xs) (x::ys) zs
```

(a) Thinning

```
thin : (f : (x : a) -> Dec (p x))
    -> (xs : List a)
          -> (dropped ** kept ** Thinning p xs dropped kept)
thin f []
  = ([] ** ([] ** TEmpty))
thin f (x :: xs)
  = case f x of
      (Yes pH)
        => case thin f xs of
             (ds ** ks ** pT)
               => (ds ** (x::ks) ** (TKeep pH pT))
      (No contra)
        => case thin f xs of
             (ds ** ks ** pT)
               => (x::ds ** ks ** TSkip contra pT)
```

(b) Code

Figure 3. Thinnings to Reason about Lists.

```
data Merge : (0         f : a -> a -> Type)
          -> (  xs,ys,zs : List a)
          -> Type
  where
    EmptyR : Merge f Nil ys  ys
    EmptyL : Merge f xs  Nil xs

    GoLT : {0 f : a -> a -> Type}
        -> (prf  : f x y)
        -> (rest : Merge f     xs  (y::ys)     zs)
                -> Merge f (x::xs) (y::ys) (x::zs)

    GoGTE : {0 f  : a -> a -> Type}
         -> (prf  : f x y -> Void)
         -> (rest : Merge f (x::xs)     ys      zs)
                 -> Merge f (x::xs) (y::ys) (y::zs)
```

(a) Specification

```
merge : (cmp : (x,y : a) -> Dec (f x y))
     -> (xs  : List a)
     -> (ys  : List a)
            -> Subset (List a)
                      (Merge f xs ys)
merge cmp xs ys with (balance xs ys)
  merge cmp [] [] | BalEmpty = (Element [] EmptyR)
  merge cmp (x :: xs) [] | LeftLean
    = (Element (x :: xs) EmptyL)
  merge cmp [] (y :: ys) | RyetLean
    = (Element (y :: ys) EmptyR)

  merge cmp (x :: xs) (y :: ys) | (Balanced b)
    = case cmp x y of
        (Yes pH)
          => case merge cmp xs (y::ys) of
               (Element zs pT) => (Element (x::zs)
                                           (GoLT pH pT))
        (No contra)
          => case merge cmp (x::xs) ys of
              (Element zs pT) => (Element (y::zs)
                                          (GoGTE contra pT))
```

(b) Code

Figure 4. Type-Driven Merging of Lists

our type with the predicate (another datatype) we will use to represent the comparison; the two lists we wish to merge; and the resulting merged list. Our specification recurses on the left operand, much as we do with list append, and the base cases capture when either input list is empty. In the recursive cases, we examine the heads of each input and append the left element if the comparison is true, and the right element if the comparison is false. Note that we erase the type representing the comparison by giving it the zero quantity.

Realising our merge program is not as straightforward as one could hope, we must battle the totality checker, and think about what our program is returning. Regardless, our program's interface is actually straightforward, we provide a comparison function, the two input lists, and the result is an instance of the specification. Our program requires the need for a dependent pair such that we can produce the witness (i.e. the result of merging) required for our `Merge` type to become inhabited. If a standard dependent pair was used, however, it would mean that the evidence (the instance of `Merge`) remains available at run time. Sometimes this is what you want to do, and other times you do not require the evidence once the program has been verified. For this example we shall use `Subset` which quantifies the evidence as being compile time only (runtime irrelevant). Furthermore, Idris' totality checker becomes confused in both recursive cases, as the size of an input list in each recursive call appears to grow. We can reuse our `Balance` view (Section 4.1.1 to capture the dually recursive structure of the merge so that Idris does see it as being total.

4.3 Type-Driven MergeSort

Merge sort operates by recursively splitting the input list in two, and merging the results in order. Section 4.2 tells us how to merge, we now need to split n' merge. Fortunately, we can adapt and take inspiration from an existing view (`SplitRec`) to describe our merge sort specification. Within Idris2's standard library, the `SplitRec` view demonstrates how to efficiently, and recursively, split a list in two. We can adapt the definition of this view to detail how we can then

merge the result of the split. Figure 5 illustrates the resulting specification: `MergeSort`

The base cases capture the empty and singleton list, and the recursive case splits the list in two and uses our `Merge` specification to perform the merge. Note that we must parameterise the type of `MergeSort` with a description of the comparison predicate, and that we have a single input and output list.

The interface for the `MergeSort` implementation follows that of `Merge`, we provide the comparison function as input together with the input list and that the output is the sorted list but with an erasable proof. When implementing `MergeSort` we can reuse the covering function for `splitRec` to perform the recursive split and use our `merge` function to merge the outputs.

Our implementation can then be used to sort *any* list, provided that we provide a suitable decidable decision procedure to compare elements.

5 Dafny & Idris

We know that our language's can colour flags, but how do they compare?

5.1 Verbosity of Verification

An immediate difference is the *verbosity* of the verification. With Dafny, the two languages (verification and programming) are not subsets of each other. Dafny provides extrinsic verification of our programs, and as a result the verification language has purposefully limited expressiveness when compared to the programming language. The limited expressiveness results in more succinct reasoning about our program's invariants, and there is a strict separation between the specification and code. The complete solution results in a single figure. That being said, the proof of correctness is not self-evident from the result. Dafny will tell us the code is correct (against the specification) but neither the code nor the IDE directly explains why.

Idris, on the other hand, does not distinguish between code and specification, and when reasoning about programs it is important to decide upon whether the approach will be extrinsic, intrinsic, or a mixture of the two. With our implementation of MergeSort it is natural to have an extrinsic proof, albeit one that is correct-by-construction, that our MergeSort result is correct.

Regardless of where the proofs are, the correct-by-construction approach results in a more verbose specification and implementation, where the implementation *is* the proof of correctness. More so, many of the specifications invariants, such as that the output must be a permutation of the input, are implicitly given in the Idris specification. Such implicit invariants can lead to incorrect specifications (Section 5.3). It is important to note, however, that with Idris we also see the detailed reasons why our code is correct and not *just* that it is.

5.2 Efficiency

The efficiency over QuickSort and MergeSort aside, it is interesting to note the efficiency of the resulting code against how it has been verified.

Given Dafny's strict separation between specification and code, the specifications are easily erased for runtime, and the specifications have no impact on the code's performance. When using Idris, however, the lack of separation means that our specifications *will* affect our code's runtime performance. That is, unless explicitly erased (i.e. given quantity `0`) or is an unbound implicit variable (which by default have quantity `0`) the code will be runtime relevant. For `mergesort`, we erased the evidence, but when using views such as `balance` there is nothing to erase. We must, thus, be careful when structuring our code, and specifying the erasure properties.

Although, views seem like an efficiency anti-pattern they can in fact help with efficient data operations. Brady [6, Chp. 10], from which we adapted the MergeSort implementation, details that views `splitRec` and `snocList` (for reversing a list) can be performant, $O(n \log n)$ and linearly. Views *are* performant.

5.3 Specification Correctness

Although we can write verified software using Idris or Dafny, a question remains: Did we verify the correct thing? Validation is just as important as verification, after all.

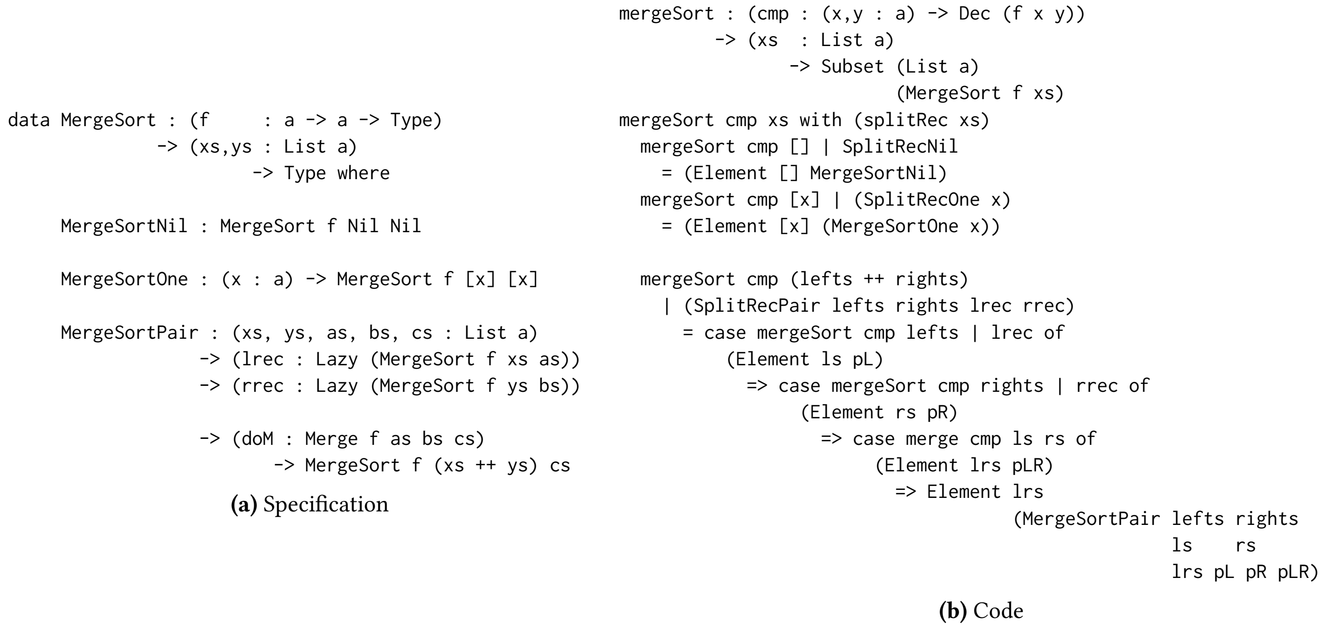
An issue with both Idris and Dafny is that we do not reason about our specifications' correctness *per se* only that they are sound. Which for Dafny means we need to ensure Boolean satisfaction through the SAT solver, and for Idris that we can write a program that generates an instance of the specification. For Idris, however, there are two ways in which we can reason about the soundness and correctness of our programs.

If the proposition is decidable, we can reason about the completeness through generation of a contradiction that the proposition will fail. This was evidenced by our use of `Dec` when comparing elements in a list. Provision of such contradictory evidence can, unfortunately, take some engineering skill.

Another approach is to provide a proof of equality that the result of running a program (in this case `mergesort`) upon completion *will* produce an expected output (in this case a sorted list). That is, providing evidence (a value, a proof) for the following type:

```
test : mergeSort [3,2,1] = [1,2,3]
test = ?hole
```

Regardless, for both languages, care must be taken that we have written suitable specifications.



```
data MergeSort : (f     : a -> a -> Type)
              -> (xs,ys : List a)
                       -> Type where

     MergeSortNil : MergeSort f Nil Nil

     MergeSortOne : (x : a) -> MergeSort f [x] [x]

     MergeSortPair : (xs, ys, as, bs, cs : List a)
                  -> (lrec : Lazy (MergeSort f xs as))
                  -> (rrec : Lazy (MergeSort f ys bs))

                  -> (doM : Merge f as bs cs)
                         -> MergeSort f (xs ++ ys) cs
```

(a) Specification

```
mergeSort : (cmp : (x,y : a) -> Dec (f x y))
         -> (xs  : List a)
                -> Subset (List a)
                          (MergeSort f xs)
mergeSort cmp xs with (splitRec xs)
  mergeSort cmp [] | SplitRecNil
    = (Element [] MergeSortNil)
  mergeSort cmp [x] | (SplitRecOne x)
    = (Element [x] (MergeSortOne x))

  mergeSort cmp (lefts ++ rights)
    | (SplitRecPair lefts rights lrec rrec)
      = case mergeSort cmp lefts | lrec of
          (Element ls pL)
            => case mergeSort cmp rights | rrec of
                 (Element rs pR)
                   => case merge cmp ls rs of
                        (Element lrs pLR)
                          => Element lrs
                                     (MergeSortPair lefts rights
                                                    ls    rs
                                                    lrs pL pR pLR)
```

(b) Code

Figure 5. Type-Driven Merge Sorting of Lists

5.4 How correct is our 'correctness'?

Just as we must think about the correctness of our specifications, we must also think about the correctness of the languages themselves.

Dafny checks the correctness of implementations (against specifications) using external tooling — principally using SAT solvers, addressed via the Boogie intermediate language [4]. Dafny and Boogie are written in C♯, and the most-used solver, Z3, in C++ [18]. We must rely on the correctness of this toolchain, and results interpretation, as performed by the Dafny developers and their testing abilities.

Idris is developed on top of a small and verifiable core language (TT) and high-level programs are elaborated down to this language. More so, Idris is self-hosting to ensure that a certain class of bugs (relating to naming and substitution) are removed. This self-hosting, however, is there to demonstrate Idris' suitability as a general purpose programming language; the semantics of TT, and higher level languages, have yet to be reasoned about within Idris.

5.5 One more Thing… Interactive Verification

Another area that we have not touched upon is the process through which our verified programs were written. Programming has long been an interactive experience, with many languages now supporting the LSP protocol; a protocol through which programmers, compilers, and editors can interact with each other. Both Dafny and Idris support such interacting editing. Dafny through the LSP protocol, and Idris using a bespoke IDE-Protocol — although there is nascent support for LSP in Idris2. What makes Idris more interesting is the support for typed holes [2] in which we can specify partial programs with, and use program synthesis to attempt to fill the hole. Using typed holes we can use our types to drive the development of our verified program's making the verification an interactive experience, something that Dafny cannot (yet) do.

6 Conclusion

We have used two verification-aware programming languages to verify that we can colour the Dutch flag correctly, and looked at how both languages help us do so. Given Dafny's simpler specification language, it is easy to see why it has gained traction in comparison to Idris' arguably more verbose presentation. Idris is, however, not just a verification-aware language, it is a language that supports dependent types and the power of pi [21, 15] and offers users a rich environment in which we can reason deeper about our programs and the languages we use to write those programs [30, 22, 23, 20].

We are just beginning our comparison between the two languages and the challenge will be: Can Dafny and Idris colour the same flags or are there limits to what each language can do, and how they do it?

References

[1] Guillaume Allais. 2023. Builtin types viewed as inductive families. In *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings* (Lecture Notes in Computer Science). Thomas Wies, editor. Vol. 13990. Springer, 113–139. DOI: 10.1007/978-3-031-30044-8_5.

[2] Luís Eduardo de Souza Amorim, Sebastian Erdweg, Guido Wachsmuth and Eelco Visser. 2016. Principled syntactic code completion using placeholders. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering* (SLE 2016). Association for Computing Machinery, Amsterdam, Netherlands, 163–175. ISBN: 9781450344470. DOI: 10.1145/2997364.2997374.

[3] Robert Atkey. 2018. Syntax and semantics of quantitative type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. Anuj Dawar and Erich Grädel, editors. ACM, 56–65. DOI: 10.1145/3209108.3209189.

[4] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs and K. Rustan M. Leino. 2005. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO* (Lecture Notes in Computer Science). Vol. 4111. Springer, 364–387.

[5] Edwin C. Brady. 2021. Idris 2: quantitative type theory in practice. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)* (LIPIcs). Anders Möller and Manu Sridharan, editors. Vol. 194. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:26. ISBN: 978-3-95977-190-0. DOI: 10.4230/LIPIcs.ECOOP.2021.9.

[6] Edwin C. Brady. 2017. *Type-driven development with Idris*. English. Manning Publications Co., (Mar. 2017). ISBN: 9781617293023.

[7] James Chapman. 2009. *Type checking and normalisation*. PhD thesis. University of Nottingham, UK. http://eprints.nottingham.ac.uk/10824/.

[8] Edsger Wybe Dijkstra. 1976. The problem of the Dutch national flag. In *A Discipline of Programming*. Prentice-Hall. Chap. 14.

[9] Cliff B. Jones. 1990. *Systematic Software Development Using VDM*. Prentice-Hall.

[10] Ioannis T. Kassios. 2006. Dynamic frames: support for framing, dependencies and sharing without restrictions. In *FM*, 268–283.

[11] Ioannis T. Kassios. 2011. The dynamic frames theory. *Formal Aspects Comput.*, 23, 3, 267–288.

[12] K. Rustan M. Leino. 2023. *Program Proofs*. MIT Press.

[13] Conor McBride and James McKinna. 2004. The view from the left. *J. Funct. Program.*, 14, 1, 69–111. DOI: 10.1017/S0956796803004829.

[14] Conor Thomas McBride. 2014. How to keep your neighbours in order. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. Johan Jeuring and Manuel M. T. Chakravarty, editors. ACM, 297–309. DOI: 10.1145/2628136.2628163.

[15] James McKinna. 2006. Why dependent types matter. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. J. Gregory Morrisett and Simon L. Peyton Jones, editors. ACM, 1. DOI: 10.1145/1111037.1111038.

[16] Bertrand Meyer. 1992. Applying "design by contract". *Computer*, 25, 10, 40–51.

[17] Bertrand Meyer. 1988. *Object-oriented Software Construction*. Prentice Hall.

[18] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *TACAS*, 337–340.

[19] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 theorem prover and programming language. In *CADE*, 625–635.

[20] Jan de Muijnck-Hughes, Guillaume Allais and Edwin C. Brady. 2023. Type theory as a language workbench. In *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands* (OASIcs). Ralf Lämmel, Peter D. Mosses and Friedrich Steimann, editors. Vol. 109. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:13. ISBN: 978-3-95977-267-9. DOI: 10.4230/OASIcs.EVCS.2023.9.

[21] Nicolas Oury and Wouter Swierstra. 2008. The power of Pi. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. ACM, 39–50. DOI: 10.1145/1411204.1411213.

[22] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers and Eelco Visser. 2018. Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.*, 2, POPL, 16:1–16:34. DOI: 10.1145/3158104.

[23] Cas van der Rest, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser and Peter D. Mosses. 2022. Intrinsically-typed definitional interpreters à la carte. *Proc. ACM Program. Lang.*, 6, OOPSLA2, 1903–1932. DOI: 10.1145/3563355.

[24] Agda Development Team. 2023. Agda. (2023). https://github.com/agda/agda.

[25] The Coq Proof Assistant Development Team. 2023. The Coq poof assistant. (2023). https://coq.inria.fr/.

[26] The Dafny Development Team. 2023. The Dafny programming and verification language. (2023). https://dafny.org/.

[27] The NuPRL Development Team. 2023. The PRL project. (2023). https://nuprl.org/.

[28] The PVS Development Team. 2023. PVS. (2023). https://pvs.csl.sri.com/.

[29] P. Wadler. 1987. Views: a way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (POPL '87). Association for Computing Machinery, Munich, West Germany, 307–313. ISBN: 0897912152. DOI: 10.1145/41625.41653.

[30] Philip Wadler, Wen Kokke and Jeremy G. Siek. 2022. *Programming Language Foundations in Agda*. (Aug. 2022). https://plfa.inf.ed.ac.uk/22.08/.

[31] Alan Wills. 1991. Capsules and types in Fresco. In *ECOOP*. (July 1991).

[32] Alan Cameron Wills. 1992. *Formal Methods applied to Object-Oriented Programming*. PhD thesis. University of Manchester.