Restricted Types: Help in Documenting Client Code Under a Verified Software Paradigm

Jason Kirschenbaum and Bruce W. Weide

The Ohio State University, Columbus OH 43210, USA
{kirschen,weide}@cse.ohio-state.edu
http://www.cse.ohio-state.edu/rsrg

Abstract. A novel programming language construct, restricted types, provides a mechanism to document abstract invariants of program variables and also may simplify program correctness proofs. Examples illustrating the use and utility of restricted types are presented.

1 Introduction

It has long been claimed in some circles that software professionals cannot be expected to write mathematically rigorous descriptions of their code such as formal specifications and loop invariants [1]. This contention arguably underestimates the capabilities of software professionals—after all, most of them have not been *taught* either why or how to write such annotations, so it is not surprising they are currently unequipped to do so. Nonetheless, the perception has led to exploration of some promising mitigating techniques that might be useful under a verified software paradigm. One approach involves inferring invariants (*e.g.,* loop invariants) either by dynamic or static analysis of code [2–6]. A complementary approach involves minimizing what needs to be written in mathematical language by providing special syntax for certain situations: syntax that looks more familiar and code-like to software developers. For instance, rather than demanding that the post-condition of an operation include a clause like `x' = x`, `x = old(x)` or `x = #x` to specify that the value of `x` does not change, most specification languages have tailor-made syntax for documenting this. JML [7] uses a `modifies` clause to list operation parameters whose values might be changed during the operation body. RESOLVE [8] offers (among others) a `restores` parameter "mode" to state that an operation parameter, while it might change temporarily during the body of the operation, has the same value at the end of the operation body as it had at the beginning.

The net effect of such mechanisms incrementally reduces the mathematical annotation burden for the software professional. It is not yet clear how effective the invariant-inference approach will be under a verified software paradigm; when automated verification does not succeed, it will be critical for a human to understand these invariants in order to repair the code or the annotations or both. This means that inferred invariants should be not only technically correct but also comprehensible to the software professional, who will ultimately

be responsible for at least reading and likely for modifying formal mathematical descriptions of software behavior. Some human input into writing invariants and other assertions therefore seems unavoidable.

This paper describes an advance down the special-syntax path: providing language constructs to reduce the annotation burden. It focuses on relationships between abstract invariant properties of individual variables that hold during an entire code segment, and loop invariants within that segment. We observe that two kinds of properties must be included in a loop invariant to verify software. The first kind arise from the desire to treat a loop as a single statement in straight-line code for verification and reasoning purposes. These properties document the behavior of the loop (by stating what it does not change); they are intimately tied to the loop and local to it. The second kind arise from the need to maintain continuity of abstract invariants on variable values. These properties are often incidental to a particular loop yet are critical pieces of the loop invariant. For example, when using memoization to avoid re-computation of a function with a Java `Map`, one abstract invariant on the `Map`'s value is that if a key is defined then the value associated with that key is the function applied to the key. This information must be written in the loop invariant for any loop involving the `Map`, because this property is true before (perhaps well before) the loop is encountered, is maintained by the loop, and might be intended to persist after (perhaps well after) the loop has terminated. This restricted set of `Map` values is known *a priori* by the software developer independently of any loops, and it can and should be documented. If the documentation is formal, its connection to the code can be verified. In other words, this documentation not only records the software developer's reasoning but—in a verified software paradigm—also can be used to check that the reasoning is correct.

The contribution of this paper is a programming language construct, restricted types, that can be used to document *abstract invariant properties of individual variables over segments of code*. The potential impact on automated verification arises partly because this construct implicitly provides guidance to the verifier by "factoring" potentially difficult verification conditions (VCs) into conceptually simpler VCs. The overabundance of assumptions in VCs has been reported [9, 10] as a problem for back-end provers.

Restricted types are presented in the context of the RESOLVE programming language, a research language designed for verifiability. Specifically, RESOLVE has clean semantics, and provides syntactic slots for contracts and mathematical annotations of various kinds. However, the restricted type construct should be adaptable to other programming languages with little change.

The paper is structured as follows. Section 2 presents a simple motivating example (in C++ rather than RESOLVE). Section 3 includes a summary of the features and syntax of RESOLVE needed to explain restricted types. An introduction to restricted types in Section 4 features an in-depth example using sorting. Related work is discussed in Section 5, with conclusions in Section 6.

2 Motivating Example

Consider code that computes x^p where x is a double and p is a positive integer; see Figure 1(a). It computes x^p by first computing x^{2^k} where k is the largest natural number that satisfies $2^k \leq p$ and then making a recursive call to finish the job. In this particular implementation, `q` always equals $2^{k'}$ where k' is some non-negative integer, and this property holds both as a loop invariant and, more generally, as an invariant on `q` throughout the code. We argue that this invariant can and should be documented.

```
double Power (double x, int p)
{
    double result = x;
    int q = 1;

    while(q <= p/2)
    /*!
        updates result, q
        maintains
            result = x ^ q and
            q <= p and
            there exists k : integer
                (q = 2 ^ k)
        decreases
            p - q
    !*/
    {
        q *= 2;
        result *= result;
    }
    if (p - q > 0)
    {
        result *= Power(x, p-q);
    }
    return result;
}
```

(a) Original version

```
double Power (double x, int p)
{
    double result = x;
    int q = 1;
    /*! restrict q to PowerOfTwo !*/
    while(q <= p/2)
    /*!
        updates result, q
        maintains
            result = x ^ q and
            q <= p


        decreases
            p - q
    !*/
    {
        q *= 2;
        result *= result;
    }
    if (p - q > 0)
    {
        result *= Power(x, p-q);
    }
    return result;
}
```

(b) Documented with a restricted type

Fig. 1. Code to compute x^p

One method a software professional can use to document the invariant on `q` is to add extra assertions in the code. At every line where the invariant holds, she `asserts` that the mathematical formula is true in the code. Frame properties allow one to limit the number of such statements needed, by using them only after a modification to a variable under consideration. This documents the invariant on `q`, but it is rather clumsy and the annotation burden is high. Restricted types (Section 4) are a construct to document the claims for this code more clearly and to reduce the annotation burden. Figure 1(b) shows what the code might look like in this situation. The loop invariant is simplified and the invariant on `q` is explicit.

We now turn our attention to presenting this idea more formally in the context of RESOLVE.

3 RESOLVE Overview

As mentioned in the introduction, RESOLVE is an imperative and component-based research language designed for verifiability [8]. The language has value semantics; reference-like behavior, when needed, is provided by a component rather than pervading all software written in the language. While RESOLVE is designed for verifiability, the design goals also include performance and understandability. Each component has a mathematical model of its behavior described in a contract, as illustrated in Figure 2; a `Queue` contract is shown. The mathematical model is explicit in the type declaration. Each operation has a formal description of behavior in terms of the mathematical model via standard `requires` and `ensures` clauses. The `control` return type is used within `if`/`while` conditions. One more restriction simplifies verifiability: all program function operations must behave as mathematical functions and must restore their arguments.

```
contract QueueTemplate (type Item)

    uses UnboundedIntegerFacility

    math subtype QUEUE_MODEL is string of Item

    type Queue is modeled by QUEUE_MODEL
        exemplar q
        initialization ensures
            q = empty_string

    procedure Enqueue (updates q: Queue, clears x: Item)
        ensures
            q = #q * <#x>

    procedure Dequeue (updates q: Queue, replaces x: Item)
        requires
            q /= empty_string
        ensures
            #q = <x> * q

    function IsEmpty (restores q: Queue): control
        ensures
            IsEmpty = (q = empty_string)

end QueueTemplate

contract Concatenate enhances QueueTemplate

    procedure Concatenate (updates p: Queue, clears q: Queue)
        ensures
            p = #p * #q

end Concatenate
```

Fig. 2. `QueueTemplate` contract and `Concatenate` extension

Behavioral extensions to abstract components, such as the `Concatenate` extension in Figure 2, are specified via contracts and implemented in terms of

other components' contracts. Another extension of a `Queue` type is the operation `Sort`. Sorting has been studied by the computer science community since the field's inception. In the past few years, there has been significant work on inferring loop invariants [2–6] among other work on verification of sorting algorithms. For the purposes of demonstrating the utility of restricted types, sorting therefore serves as an appropriate standard benchmark that naturally involves variables with abstract invariants beyond any of the generic Abstract DataType (ADT) invariants of its variables.

3.1 Sort Specification

Since the `QueueTemplate` component is generic, *i.e.,* parameterized by a type `Item`, the contract of a `Sort` operation should also be generic. Figure 3 shows the requisite restriction on the ordering relation `ARE_IN_ORDER` to be used in sorting, namely that `ARE_IN_ORDER` is a total pre-order.

Figure 3 shows the mathematical definitions used to specify sorting, given `ARE_IN_ORDER`. `OCCURS_COUNT` is a mathematical function that returns the number of times a given `Item` appears in a string; it is used to construct the other mathematical definitions. This allows the contract to be specific about the value of the outgoing `Queue`: not only are the values of items in the outgoing `Queue` the same as in the incoming queue, but the number of times each appears is the same. `IS_PRECEDING` is a binary predicate and evaluates to true if and only if every item in the first string is related by `ARE_IN_ORDER` to every item in the second string; intuitively, every item in the first string is "no larger" than every item in the second. `IS_NON_DECREASING` is a predicate on a string and evaluates to true if and only if every consecutive pair of `Item`s in the string are related by `ARE_IN_ORDER`. Finally, `IS_PERMUTATION` is a binary predicate on strings that is true if and only if the number of occurrences of every `Item` is the same in the first string and in the second.

The contract for the specification of a `Sort` operation is given in Figure 3. The `Sort` operation takes a `Queue` and returns with the property that the outgoing string `q` is a permutation of the incoming string and the outgoing string is non-decreasing.

3.2 Quicksort Implementation

We present an implementation of the `Sort` operation using quicksort in Figure 4. Our implementation partitions a non-empty incoming queue into two queues (`q` and `qBig`) and a partitioning element (`partitionElement`) with the property that every `Item` in `q` is in order with `partitionElement` and `partitionElement` is in order with every item in `qBig`. Each of the smaller queues is sorted recursively and `q`, `partitionElement`, and `qBig` are all concatenated to obtain the final, sorted queue. Besides the loop invariant and other mathematical annotations, this code is similar to code in most other languages. The loop invariant documents the insight of the algorithm, namely the ordering relationships among the variables `partitionElement`, `q` and `qBig`, as expressed formally via

```
contract Sort (
      definition ARE_IN_ORDER (x: Item, y: Item): boolean
             satisfies restriction
                for all z: Item
                     ((ARE_IN_ORDER (x, y)  or  ARE_IN_ORDER (y, x))  and
                      (if (ARE_IN_ORDER (x, y)  and  ARE_IN_ORDER (y, z))
                         then ARE_IN_ORDER (x, z)))
              )
    enhances QueueTemplate

    definition OCCURS_COUNT (
        s: string of Item,
        i: Item
    ) : integer
       satisfies
           if s = empty_string
           then OCCURS_COUNT (s, i) = 0
           else
               there exists x: Item, r: string of Item ((s = <x> * r)
               and
               (if x = i
               then OCCURS_COUNT (s, i) = OCCURS_COUNT (r, i) + 1
               else OCCURS_COUNT (s, i) = OCCURS_COUNT (r, i)))

     definition IS_PRECEDING (
         s1: string of Item,
         s2: string of Item
     ) : boolean
        is for all i, j: Item where (OCCURS_COUNT (s1, i) > 0 and
                                     OCCURS_COUNT (s2, j) > 0)
                 (ARE_IN_ORDER (i, j))

     definition IS_NON_DECREASING (
         s: string of Item
     ) : boolean
        is for all a, b: string of Item where (s = a * b)
                 (IS_PRECEDING (a, b))

    definition IS_PERMUTATION (
        s1: string of Item,
        s2: string of Item
    ) : boolean
      is for all i: Item
               (OCCURS_COUNT (s1, i) = OCCURS_COUNT (s2, i))

    procedure Sort (updates q: Queue)
        ensures
            IS_PERMUTATION (q, #q)  and
            IS_NON_DECREASING (q)

end Sort
```

Fig. 3. `Sort` extension to `QueueTemplate`

`IS_PRECEDING` and `IS_PERMUTATION`. The programmer's reasoning about why the operation terminates is given by the `decreases` clause (*i.e.,* progress metric).

A local operation `Partition` is used to split a queue according to the quicksort algorithm. The `:=:` operator is the "swap " operator [11, 12]. This exchanges the values of its two arguments, and is a key aspect of avoiding aliasing while preserving efficiency.

```
realization QuickSort (
        function AreInOrder (restores i: Item, restores j: Item): control
            ensures
                AreInOrder = ARE_IN_ORDER (i, j)
    ) implements Sort for QueueTemplate

    uses Concatenate for QueueTemplate

    local procedure Partition (updates qSmall: Queue,
                               replaces qBig: Queue,
                               restores p: Item)
        ensures
            IS_PERMUTATION (qSmall * qBig, #qSmall) and
            IS_PRECEDING(qSmall, <p>) and IS_PRECEDING(<p>, qBig)

        variable tmp: Queue
        Clear (qBig)
        loop
            updates qSmall, qBig, tmp
            maintains
                IS_PERMUTATION (qSmall * qBig * tmp,
                                #qSmall * #qBig * #tmp) and
                IS_PRECEDING(tmp, <p>) and
                IS_PRECEDING(<p>, qBig)
            decreases |qSmall|
        while not IsEmpty (qSmall) do
            variable x: Item
            Dequeue (qSmall, x)
            if AreInOrder (x, p) then
                Enqueue (tmp, x)
            else
                Enqueue (qBig, x)
            end if
         end loop
         qSmall :=: tmp
    end Partition

    procedure Sort (updates q: Queue)
        decreases |q|

        if not IsEmpty (q) then
            variable partitionElement: Item
            variable qBig: Queue

            Dequeue (q, partitionElement)
            Partition (q, qBig, partitionElement)
            Sort(q)
            Sort(qBig)

            Enqueue (q, partitionElement)
            Concatenate (q, qBig)
        end if
    end Sort
end QuickSort
```

Fig. 4. Quicksort implementation of `Sort` extension to `QueueTemplate`

The next section rewrites this sorting contract and implementation to use restricted types.

4 Introduction to the Syntax and Semantics of Restricted Types: Sorting Example

First we examine the issues involved in definitions of restricted types, and then the issues in the usage of restricted types in client code.

4.1 Restricted Types

We create three restricted types for this example, one for each different abstract invariant maintained by specific uses of `Queue`s in quicksort. The first invariant is that a `Queue` is sorted, *i.e.,* it is an `OrderedQueue`. The other invariants relate the `Item`s in a `Queue` to another `Item`. These invariants arise during the `Partition` implementation and simply relate `qSmall` to `p` and `qBig` to `p` by `ARE_IN_ORDER`; more specifically every `Item` in `qSmall` is in order with `p` and `p` is in order with every `Item` in `qBig`. For each operation that is called on any `Queue` that satisfies one of these properties, the programmer reasons that the abstract invariant is not broken by the operation call. The proof boils down to the question: if the operation is executed, does the new value of the variable still satisfy the restriction? With this intuition in mind, we show the contracts for the restricted types corresponding to these ideas in Figure 5.

A restricted type is declared using an existing programmatic type, in this case `Queue`. The new type's restriction is given by a predicate using an exemplar (which is a name for a prototypical variable of the restricted type). Operations on the previously created programmatic type that are affected by this restricted type are written in the `operations` section. Each of these operations can have additional `requires` and `ensures` clauses defined by `also requires` and `also ensures`, respectively. Since functions may not "break" the invariant—they cannot change the abstract value of any argument—they are always available to be used on any restricted type. `SmallValueQueue` and `LargeValueQueue` are constructed in the same manner.

Conceptually, the `also requires` clauses are conjoined with the original `requires` clauses. These are used by the programmer to ensure both that the restriction is maintained by the operation, and to document conditions under which it is safe to call the operation while still maintaining the invariant. The `also ensures` clauses strengthen the previous postconditions. In the `OrderedQueue` contract, `Dequeue`'s `also ensures` clause gives information about how the dequeued item relates to items that remain in the `OrderedQueue`.

Since programmers may need some help in making sure that their reasoning process is correct, the compiler should generate VCs corresponding to the correctness of the restricted type contract. The contract's correctness condition is that if an operation is invoked in a state satisfying the variable restrictions

```
restricted type OrderedQueue is Queue
   exemplar q
   restriction
       IS_NON_DECREASING(q)
   operations
       Enqueue(updates q: OrderedQueue,
               clears x : Item)
           also requires IS_PRECEDING(q, <x>)
       Dequeue(updates q: OrderedQueue,
               replaces x : Item)
           also ensures IS_PRECEDING(<x>, q)
end OrderedQueue

restricted type SmallValueQueue(max : Item) is Queue
    exemplar q
       restriction IS_PRECEDING(q,<max>)
    operations
       Enqueue(updates q : SmallValueQueue,
               clears x : Item)
            also requires ARE_IN_ORDER(x, max)
       Dequeue(updates q : SmallValueQueue,
               replaces x : Item)
            also ensures ARE_IN_ORDER(x, max)
end SmallValueQueue

restricted type LargeValueQueue(min : Item) is Queue
    exemplar q
       restriction IS_PRECEDING(<min>, q)
    operations
       Enqueue(updates q : LargeValueQueue,
               clears x : Item)
           also requires ARE_IN_ORDER(min, x)
       Dequeue(updates q : LargeValueQueue,
               replaces x : Item)
           also ensures ARE_IN_ORDER(min, x)
end LargeValueQueue
```

Fig. 5. `OrderedQueue`, `SmallValueQueue` and `LargeValueQueue` restricted types

and the `requires` clause, and the operation completes successfully, then the restriction is still satisfied by the updated variables; any `also ensures` clauses must also be satisfied. More concretely, each operation's invocation can be assumed to occur in a state in which the restriction, the original `requires` clause, and the `also requires` clause hold. By a process similar to datatype induction, these VCs are generated just once for the contract. (Notice that this construction leaves the initialization of restricted types to a client-side activity and is discussed in Section 4.2.) The general form of the generated VCs, where s is a variable of the mathematical model of the restricted type, $args$ is the list of arguments to the operation, and $'$ indicates a fresh variable, is given by:

$$\begin{aligned} & restriction(s') \wedge requires_{original}(s', args') \wedge \\ & requires_{also}(s', args') \wedge ensures_{original}(s', args', s, args) \\ \rightarrow & restriction(s) \wedge ensures_{also}(s', args', s, args) \end{aligned}$$

4.2 Client Usage of Restricted Types

The updated `Sort` contract, shown in Figure 6, is almost the same as the original contract. The difference is that the `Queue` formal parameter `q` is restricted to be of type `OrderedQueue` when the operation returns. When this operation is called by a client, the formal parameter `q` must be of type `Queue` 9checked as part of normal static type-checking; when `Sort` returns, `q` conforms to the restricted type `OrderedQueue` (checked by verification). We can, therefore, omit the `IS_NON_DECREASING(q)` from the `ensures` clause, since it is subsumed by the restricted type.

```
contract Sort (
....

    procedure Sort (updates q: Queue restricts to OrderedQueue)
        ensures
            IS_PERMUTATION (q, #q)

end Sort
```

Fig. 6. `Sort` extension to `QueueTemplate` using restricted types

This reduces the annotation burden on the programmer. The `restricts to` annotation in the formal parameters is not checked by the type system. It is equivalent to having the type restriction in the `ensures` clause for that variable. We examine this issue in more depth in the discussion of the `Sort` operation.

Figure 7 shows an additional operation `Concatenate` defined on `Queue`s that is used by `OrderedQueue`s. These operations can be added in an `additional operations` section to specify the operations on the underlying type that should be added to the restricted type.

```
restricted type OrderedQueue
   additional operations
       Concatenate(updates q1: OrderedQueue,
                   clears q2: OrderedQueue)
           also requires IS_PRECEDING(q1, q2)
end OrderedQueue
```

Fig. 7. Additional operations needed on `OrderedQueue`

The `Partition` operation uses the `SmallValueQueue` and `LargeValueQueue` restricted types. The code for performing the partition operation is given in Figure 8. In the contract of `Partition`, the `ensures` clause is simplified by the use of the `restricts to` annotation. Otherwise, the code is similar to the original version.

```
realization QuickSort (
....

    local procedure Partition (updates qSmall: Queue
                                  restricts to SmallValueQueue(p),
                               replaces qBig: Queue
                                  restricts to LargeValueQueue(p),
                               restores p: Item)
        ensures
            IS_PERMUTATION (qSmall * qBig, #qSmall)

        variable tmp: SmallValueQueue(p)
        Clear (qBig)
        restrict qBig to LargeValueQueue(p)
        loop
            updates qSmall, qBig, tmp
            maintains
                IS_PERMUTATION (qSmall * qBig * tmp,
                                #qSmall * #qBig * #tmp)
            decreases |qSmall|
        while not IsEmpty (qSmall) do
            variable x: Item
            Dequeue (qSmall, x)
            if AreInOrder (x, p) then
                Enqueue (tmp, x)
            else
                Enqueue (qBig, x)
            end if
         end loop
         restrict qSmall to SmallValueQueue
         qSmall :=: tmp
    end Partition

    procedure Sort (updates q: Queue restricts to OrderedQueue)
        decreases |q|

        variable qtmp: Queue
        qtmp :=: q
        restrict q to OrderedQueue
        if not IsEmpty (qtmp) then
            variable partitionElement: Item
            variable qBig: Queue

            Dequeue (qtmp, partitionElement)
            Partition (qtmp, qBig, partitionElement)
            Sort(qtmp)
            Sort(qBig)

            Enqueue (qtmp, partitionElement)
            Concatenate (qtmp, qBig)
            q :=: qtmp
        end if
    end Sort
end QuickSort
```

Fig. 8. Quicksort implementation of `Sort` using restricted types

Recall that in the contracts of restricted types, there were no VCs generated for initialization; that piece is left to the clients or users of the restricted type. So, when a variable of a particular type, say `Queue`, is restricted to, say, a `LargeValueQueue`, a VC is generated to make sure that that variable is a valid value of that restricted type. The VC to be generated is exactly what

one would expect—the variable satisfies the restricted type restriction. For example, `restrict qBig to LargeValueQueue(p)` generates a VC whose goal is `IS_PRECEDING(<p>, qBig)` and whose assumptions are those facts known at that point in the code, *e.g.,* resulting from path conditions, loop invariants, and contracts of other operations called. As syntactic sugar for declaring a variable of the underlying type and then restricting it immediately, a declaration of a variable of a restricted type generates a VC that any initial value of its underlying type satisfies the restriction. We note that not only is the specification of `Partition` simpler, but the loop invariant has been significantly simplified as well.

Figure 8 also shows the `Sort` implementation using restricted types and the modified `Partition` local operation. Except for the `restricts to` annotation, the code is exactly the same as the original version.

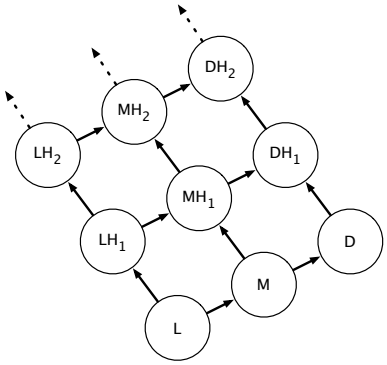
Finally, to finish an earlier discussion about the implementation of the `restricts to` annotation in an operation parameter, one can implement the annotation by automatically translating it into an `ensures` clause in the operation contract. On every client use of the operation, the verification system would add a `restrict to` annotation after the call to reassert the restricted type, generating one additional (simple) VC. This process would be invisible to the user, but would simplify the type information needed for restricted types by avoiding carrying that information across operation boundaries.

4.3 Evaluation

The use of restricted types may also help simplify proofs of VCs. We examine the impact of restricted types on the difficulty of VCs as defined by [13]. In that work, VCs are categorized according to the number of hypotheses (H_0, H_1,...) and whether only logical rules (L), theory-specific knowledge (M) or local mathematical definitions (D) are needed to prove a VC. VCs that use fewer assumptions or require less mathematical knowledge are considered less difficult. The metrics are summarized in Figure 9.

Label	**What is needed in the proof**
L	Rules of mathematical logic
H_n	At most n hypotheses from the VC needed ($n > 0$)
M	Knowledge of mathematical theories used in the specifications
D	Knowledge of programmer-supplied definitions based on mathematical theories above

(a) VC classification



(b) Lattice of the VC classification

Fig. 9. VC classification and diagram of category relationships (adapted from [13])

The code presented in Section 3 without restricted types was compared to the code in this section with restricted types. The original quicksort implementation's most difficult VC was categorized as MH_6, while the restricted types version has VCs of difficulty at most MH_3. The MH_6 VC is particularly difficult; it arises from proving the second conjunct in the `ensures` clause of `Sort`:

$$
\begin{array}{ll}
1: & \quad is\_initial(partitionElement_2) \\
2: & \wedge\ IS\_PERMUTATION(q_5 * qBig_5, q_4) \\
3: & \wedge\ IS\_PRECEDING(q_5, \langle partitionElement_4 \rangle) \\
4: & \wedge\ IS\_PRECEDING(\langle partitionElement_4 \rangle, qBig_5) \\
5: & \wedge\ IS\_PERMUTATION(q_6, q_5) \\
6: & \wedge\ IS\_NON\_DECREASING(q_6) \\
7: & \wedge\ IS\_PERMUTATION(qBig_7, qBig_5) \\
8: & \wedge\ IS\_NON\_DECREASING(qBig_7) \\
9: & \wedge\ is\_initial(partitionElement_8) \\
10: & \wedge\ \langle partitionElement_4 \rangle * q_4 \neq \Lambda \\
\hline
 & \quad \rightarrow IS\_NON\_DECREASING(q_6 * \langle partitionElement_4 \rangle * qBig_7)
\end{array}
$$

The proof requires hypotheses 3 through 8, and is fairly involved; mathematical lemmas are needed, for instance, to conclude that hypotheses 3, 5 and 6 imply $IS\_PRECEDING(q_6, \langle partitionElement_4 \rangle)$. The corresponding VCs from restricted types are easier. The direct analog of the above VC, in particular, is in category LH_1, *i.e.,* the goal is one of the hypotheses. The proof of a VC arising from the call to `Concatenate` in the body of `Sort` is the most difficult: it is in the category MH_3.

Another VC in MH_3 arises from the `also requires` clause for `Concatenate`:

$$
\begin{array}{ll}
1: & \quad IS\_PRECEDING(q1\_o, q2\_o) \\
2: & \wedge\ IS\_NON\_DECREASING(q1\_o) \\
3: & \wedge\ IS\_NON\_DECREASING(q2\_o) \\
\hline
 & \quad \rightarrow IS\_NON\_DECREASING(q1\_o * q2\_o)
\end{array}
$$

The one-time, reusable proof of this VC is also in MH_3. However, it is a relatively easy proof to discharge; it is an algebraic lemma of string theory. This is the essence of the proof of the original VC. Proving these VCs with Isabelle [14] using a version RESOLVE's string theory in an automatic mode [10] confirms that the MH_6 VC is hard to prove—Isabelle does not prove it automatically. The two MH_3 VCs are proved automatically. For this example, restricted types are able to simplify the code annotations and reduce the maximum difficulty of VCs generated from the resulting code.

5 Related Work

The idea of restricted types is similar to predicate subtyping, particularly as manifest in PVS [15]. Predicate subtyping is a mathematical notion of using a predicate over a type to define a subtype. In the implementation of PVS, the new type is defined and Type Correctness Conditions (TCCs) are generated when converting from a type to a predicate subtype. Those TCCs serve a similar purpose to the VCs presented here, *e.g.,* a TCC to ensure the existence of an object that satisfies the given predicate along with TCCs to ensure that the predicate holds when a subtype is returned. Restricted types essentially moves this idea from the mathematical domain into the programmatic domain.

The Jahob system [9] uses annotated Java source code as its source language. The annotation language has support for a proof language, with essentially a full first-order prover functionality. There are the usual first order proof commands, such as applying modus ponens, along with commands to perform local proofs. Invariants can be expressed as well. While Jahob's proof language is powerful, the proof commands are not as natural for a software professional. Rather than learning a proof system, software professionals using restricted types think in terms of contracts and component invariants, concepts that are used in the normal course of programming.

Behavioral subtyping [16] uses a set of rules to ensure that a subtype can always be used in place of a supertype without violating a behavioral property of the client program. Contractually, the preconditions of any subtype operation may not be strengthened, postconditions may not be weakened, and invariants must be preserved. Restricted types impose different requirements; in particular, preconditions may be strengthened. The goal of restricted types is not to allow for substitution, but rather to indicate that during specific code segments (i.e., not necessarily for the entire lifetimes of variables) stronger abstract invariants hold for specific variables.

6 Conclusion

We have presented a programming language construct, restricted types, that addresses a limitation in current verification languages, namely the clumsiness of formally documenting client code. This construct, when applied to code similar to that shown in Section 4, provides a mechanism to separate out two uses of loop invariants, namely an abstraction of the behavior of a loop and a mechanism to maintain abstract invariants on variables. This approach not only can simplify VCs generated in client code, but also can result in reasoning reuse. This reuse happens both when restricted types are reused across clients, and even when there are multiple calls to a single restricted-type operation by a particular client.

7 Acknowledgments

The authors are grateful for the constructive feedback from Bruce Adcock, Derek Bronish, Paolo Bucci, Harvey M. Friedman, Wayne Heym, Bill Ogden, Aditi Tagore, and Diego Zaccai. This material is based upon work supported by the National Science Foundation under Grants No. DMS-0701260 and CCF-0811737. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL '02, New York, ACM (2002) 191–202
2. Seghir, M.N., Podelski, A., Wies, T.: Abstraction refinement for quantified array assertions. In: SAS '09, Berlin, Heidelberg, Springer-Verlag (2009) 3–18
3. Perrell, V., Halbwachs, N.: An analysis of permutations in arrays. In: VMCAI: 2010, Berlin, Heidelberg, Springer-Verlag (2010) 279–294
4. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: PLDI '08, New York, ACM (2008) 339–348
5. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. SIGPLAN Not. **43**(1) (2008) 235–246
6. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. SIGPLAN Not. **44**(6) (2009) 223–234
7. Leavens, G.T., Leino, K.R.M., Poll, E., Ruby, C., Jacobs, B.: JML: notations and tools supporting detailed design in Java. In: OOPSLA 2000 Companion, Minneapolis, Minnesota. (2000) 105–106
8. Ogden, W.F., Sitaraman, M., Weide, B.W., Zweben, S.H.: Part I: the RESOLVE framework and discipline: a research synopsis. SIGSOFT Softw. Eng. Notes **19**(4) (1994) 23–28
9. Zee, K., Kuncak, V., Rinard, M.C.: An integrated proof language for imperative programs. In: PLDI '09, New York, NY, USA, ACM (2009) 338–351
10. Kirschenbaum, J., Adcock, B.M., Bronish, D., Bucci, P., Weide, B.W.: Adapting isabelle theories to help verify code that uses abstract data types. In: SAVCBS. (November 2008) 67–74
11. Harms, D., Weide, B.: Copying and swapping: Influences on the design of reusable software components. IEEE Transactions on Software Engineering **17**(5) (May 1991) 424–435
12. Pike, S.M., Heym, W.D., Adcock, B., Bronish, D., Kirschenbaum, J., Weide, B.W.: Traditional assignment considered harmful. In: OOPSLA '09, New York, ACM (2009) 909–916
13. Kirschenbaum, J., Adcock, B.M., Bronish, D., Smith, H., Harton, H.K., Sitaraman, M., Weide, B.W.: Verifying component-based software: Deep mathematics or simple bookkeeping? In: ICSR. (2009) 31–40
14. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
15. Rushby, J., Owre, S., Shankar, N.: Subtypes for specifications: Predicate subtyping in PVS. IEEE Transactions on Software Engineering **24** (1998) 709–720
16. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6) (1994) 1811–1841