Liquid Types for Array Invariant Synthesis [⋆]

Manuel Montenegro, Susana Nieva, Ricardo Peña, and Clara Segura
montenegro@fdi.ucm.es,nieva@ucm.es,ricardo@sip.ucm.es,csegura@sip.ucm.es

Universidad Complutense de Madrid, Spain

Abstract. Liquid types qualify ordinary Hindley-Milner types by predicates expressing properties. The system infers the types of all the variables and checks that the verification conditions proving correctness hold. These predicates are currently expressed in a quantifier free decidable logic.
Here, we extend Liquid types with quantified predicates of a decidable logic for arrays, propose a concept of an array refinement type, and present an inference algorithm for this extension, that we have implemented. By applying our tool to several imperative algorithms dealing with arrays, we have been able to infer complex invariants.

1 Introduction

Liquid types [13, 10, 17–19] are a variant of dependent types which have been successfully used for automatically verifying a number of non trivial properties of programs. Recently they have also been used as a guide for synthesizing correct programs [12]. They have been mainly applied to functional languages. A liquid type is a refinement of an ordinary type, defined by restricting the set of possible values to those satisfying a predicate. This predicate may have as free variables some variables in scope. In this way, the type depends on the values computed by the program.

The original idea [13] has been extended to recursive data structures [10], and it is possible for instance to define a list whose tail values depend on the value at the head, or a tree whose children values depend on the value at the root. This captures in a natural way invariants of sorted lists, binary search trees, binary heaps and many other interesting data structures. Once the programmer has written the invariant, the system assists the programmer in verifying that the functions manipulating the data structure actually preserve the invariant. This saves most of the verification effort that would be needed by doing it manually.

The underlying machinery is a type inference algorithm which tries to prove a set of logical implications, which in essence are the verification conditions that a human programmer would try to prove manually. The system does it automatically with the aid of a SMT solver. In order that the solver never fails to prove a formula, the logic of the predicates used in liquid types must be

[⋆] Work partially supported by the Spanish MINECO project CAVI-ART (TIN2013-44742-C4-3-R), Madrid regional project N-GREENS Software-CM (S2013/ICE-2731) and UCM grant GR3/14-910502

decidable. In its current state, the logic, and hence the liquid types one can infer, does not include quantifiers. It supports however linear integer arithmetic, equality, algebraic types and uninterpreted functions (this logic is known as QF-EUFLIA).

Our contribution here is extending the liquid types to properties on arrays, which very frequently need predicates universally quantified over the array indices. Nevertheless, we still remain in the safe side by only allowing formulas in a decidable theory of arrays, which is a fragment of Bradley and colleagues' [2].

Additionally, we extend the type inference algorithm to quantified formulas, and also use SMTs to automatically discharge them. This extension includes two novelties: (a) new liquid variables are created dynamically in order to split a quantified formula over an array segment into two or more formulas over smaller segments; and (b) these variables occur in negative positions of the formula. Even though, our domain is still a finite one, and our abstract interpretation is monotonic in this domain. This ensures that the inference algorithm always terminates without the need of a widening operator.

Another contribution is that we apply the liquid type technology to imperative programming languages dealing with arrays such as C++ and Java. This is possible thanks to our verification platform [11] that transforms programs into an *intermediate representation* (IR) common to all these languages. In essence, this IR is a desugared functional language, where state updating has been replaced by dynamic creation of variables, and iteration has been transformed into recursion.

Our inference algorithm has been integrated in that platform. With this new tool, we have inferred complex invariants on arrays, as for instance those occurring in the imperative sorting algorithms. We think that this opens the door to the use of liquid types in verifying non trivial properties of programs written in conventional imperative languages.

The plan of the paper is as follows: after this introduction, Sec. 2 explains some fundamentals about liquid types and their inference algorithm; then, Sec. 3 reviews the decidable theories about arrays; inspired in those theories, Sec. 4 contains our proposal for an array refinement, whose aim is to capture as many properties about arrays as possible out of those arising in imperative algorithms; Sec. 5 presents our type inference algorithm, and Sec. 6 shows a number of meaningful examples to which the algorithm has been applied. Sec. 7 relates our approach to other works in the literature, and Sec. 8 draws some conclusions.

2 Liquid Types

The Liquid type system [13] extends the polymorphic Hindley-Milner type system by decorating types with *refinement predicates* constraining the values represented by them. A refined type has the form $\{\nu : \tau \mid e\}$, where τ is a Hindley-Milner type and e is a boolean expression which may name the ν variable and other program variables. This type represents the values b of type τ such that $e[b/\nu]$ evaluates to *true*. For instance, the type $\{\nu : \mathit{int} \mid \nu \geq 0\}$ represents the type of nonnegative integers. Another example is the following declaration, which specifies the type of a function `get` for array indexing,

$$\texttt{get} :: \forall\alpha.(a : \mathit{array}\ \alpha) \to i : \{\nu : \mathit{int} \mid 0 \leq \nu \wedge \nu < \mathit{len}\ a\} \to \{\nu : \alpha \mid \nu = a[i]\} \qquad (1)$$

where *len* a represents the length of the array a. The type *array* α abbreviates the refined type $\{\nu : \mathit{array}\ \alpha \mid \mathit{true}\}$.

In their most general form, type checking and type inference of refined types is undecidable. However, in the Liquid type system inference becomes decidable by restricting the boolean expressions to the logic of linear arithmetic, equality and uninterpreted functions (QF-EUFLIA), and by bounding the search space of refinements with the help of logical qualifiers.

A *logical qualifier* q is a predicate which depends on ν and a placeholder variable denoted by $\star$. The set $\mathbb{Q}$ of qualifiers to be used is given by the programmer. The larger this set, the more complex refinements can be specified, but the larger the search space becomes. An *instance* of a logical qualifier q is another qualifier obtained by replacing each placeholder in q by a program variable. We denote by $\mathbb{Q}^*$ the set of qualifiers that are instances of $\mathbb{Q}$. Since $\mathbb{Q}$ is finite, so is $\mathbb{Q}^*$. A *liquid type* is a refined type in which the refinement predicates are conjunctions of elements in $\mathbb{Q}^*$. For instance, if $\mathbb{Q} = \{\nu \geq 0, \nu < \mathit{len}\ \star, \nu = \star[\star]\}$, the type (1) is a liquid type.

The inference algorithm, which will be detailed in Sec. 2.1 transforms subtyping relations between liquid types into boolean formulas which are subsequently sent to a SMT solver. The variables occurring in these formulas are assumed to be universally quantified at the outermost level. However, in some cases we need nested quantification: assume a function that initializes all the positions of an array with a given element x. The type of the resulting array have the refinement $\forall i.0 \leq i < n \to \nu[i] = x$. As another example, a function that sorts an array would have the refinement $\forall i.\forall j.0 \leq i \leq j < \mathit{len}\ \nu \to \nu[i] \leq \nu[j]$ in the type of the output. These types are not liquid types, since their refinements are not conjunctions of qualifiers, but universally quantified formulas.

The original work of [13] summarized above only manages quantifier-free refinements in order to make inference decidable. In further work [10, 17], the authors extend the Liquid type system in order to allow parametricity on the refinement predicates. This is achieved by including *refinement predicate variables*. For instance, if p denotes a predicate variable, the type $\mathit{int}\ \langle p\rangle$ stands for the set of integers x such that $p\ x$ holds. This type can be instantiated, for instance, to $\mathit{int}\langle\lambda x.x\ \mathit{mod}\ 2 = 0\rangle$, which denotes the set of even integers. This idea is also applied to arrays by including two refinement predicate variables into the *array* data type. The first one (*dom*) constraints the set of valid indexes, whereas the second one (*rng*) specifies the property that must hold for each element stored in the array. This property may also, in turn, depend on the element index. Therefore we would have the type $\mathit{array}\ \alpha\langle \mathit{dom}, \mathit{rng}\rangle$ with the following signatures for accessing and modifying arrays:

$$\begin{aligned}
&\texttt{get} :: \forall\alpha.\forall(\mathit{dom} :: \mathit{int} \to \mathit{bool}).\forall(\mathit{rng} :: \mathit{int} \to \alpha \to \mathit{bool}).\\
&\qquad\qquad (i : \mathit{int}\langle \mathit{dom}\rangle) \to \mathit{array}\ \alpha\langle \mathit{dom}, \mathit{rng}\rangle \to \alpha\langle \mathit{rng}\ i\rangle\\
&\texttt{set} :: \forall\alpha.\forall(\mathit{dom} :: \mathit{int} \to \mathit{bool}).\forall(\mathit{rng} :: \mathit{int} \to \alpha \to \mathit{bool}).\\
&\qquad\qquad (i : \mathit{int}\langle \mathit{dom}\rangle) \to \mathit{array}\ \alpha\langle \mathit{dom}', \mathit{rng}\rangle \to \alpha\langle \mathit{rng}\ i\rangle \to \mathit{array}\ \alpha\langle \mathit{dom}, \mathit{rng}\rangle
\end{aligned}$$

where dom' abbreviates $\lambda k.\mathit{dom}\ k \wedge k \neq i$. These parametric arrays allow one to express properties on the elements of an array while still avoiding quantified formulas in the refinements. However, this approach has some drawbacks. In principle we would be tempted to think that the type $\mathit{array}\ \alpha\langle \mathit{dom}, \mathit{rng}\rangle$ is semantically equivalent to the refined type $\{\nu : \mathit{array}\ \alpha \mid \forall i.\mathit{dom}\ i \to \mathit{rng}\ i\ \nu[i]\}$. However, there is a difference: in presence of refinement variables, the subtyping relation is defined covariantly. That is, in order to prove that $\mathit{array}\ \alpha\langle \mathit{dom}, \mathit{rng}\rangle$ is a subtype of $\mathit{array}\ \alpha\langle \mathit{dom}', \mathit{rng}'\rangle$ the condition $\forall i.\forall z.(\mathit{dom}\ i \wedge \mathit{rng}\ i\ z \Rightarrow \mathit{dom}'\ i \wedge \mathit{rng}'\ i\ z)$ is sent to an SMT solver. We cannot justify covariance in the dom variable, as this implies, for instance, that an array whose indices are in $[0..3]$ is a subtype of an array whose indices range over the interval $[0..5]$. On the other hand, if we allow quantifiers in refinement types, proving that $\{\nu : \mathit{array}\ \alpha \mid \forall i.\mathit{dom}\ i \to \mathit{rng}\ i\ \nu[i]\}$ is a subtype of $\{\nu : \mathit{array}\ \alpha \mid \forall i.\mathit{dom}'\ i \to \mathit{rng}'\ i\ \nu[i]\}$ amounts to prove the assertion $(\forall i.\mathit{dom}\ i \to \mathit{rng}\ i\ \nu[i]) \Rightarrow (\forall i.\mathit{dom}'\ i \to \mathit{rng}'\ i\ \nu[i])$. This kind of assertions can be managed by some SMTs under some conditions which will be explained in Sec. 3.

Another drawback of the type $\mathit{array}\langle \mathit{dom}, \mathit{rng}\rangle$ is that properties involving two quantifiers, such as the one shown above for the sort function, cannot be expressed. Our main technical contribution in this work is the extension of the Liquid type system in order to be able to infer properties involving quantification on the indices of the array in order to overcome the limitations explained above.

2.1 Features of the type system and inference

As mentioned above, there is a subtyping relation between refined types. This relation is defined by a set of rules of the form $\Gamma \vdash \tau_1 <: \tau_2$, meaning that τ_1 is a subtype of τ_2 under an environment Γ. The type system is path-sensitive, so the type environment does not only contain the types of the variables in scope, but also the conditions that are satisfied in the context of an expression (these are gathered, for example, when traversing `if` expressions). Among the typing rules of the system (see [13]), the most relevant one specifies that, under Γ, the type $\{\nu : B \mid e_1\}$ is a subtype of $\{\nu : B \mid e_2\}$ whenever B is a basic type and the formula $[\![\Gamma]\!] \wedge e_1 \Rightarrow e_2$ is valid. The notation $[\![\Gamma]\!]$ is a logical characterization of the environment in which each binding of the form $x : \{\nu : B \mid e\}$ is translated into the formula $e[x/\nu]$.

The inference algorithm assumes that a standard Hindley-Milner inference has been applied previously. After this, each type τ in the typing derivation is refined with a fresh template variable κ so as to obtain $\{\nu : \tau \mid \kappa\}$. Type inference consists in finding a substitution A from variables κ to conjunctions of $\mathbb{Q}^*$ such that, when applied to the typing derivation, the expression type checks. This solution is obtained by a standard fixpoint algorithm. Initially all refinement templates are mapped to $\bigwedge_{q \in \mathbb{Q}^*} q$, which is the strongest refinement. If it is a valid solution, the algorithm terminates. Otherwise, the subtyping rules must have generated an assertion $A([\![\Gamma]\!]) \wedge A(\kappa_1) \Rightarrow A(\kappa_2)$ that is not proven valid by the SMT. In this case the algorithm modifies the substitution A by removing from $A(\kappa_2)$ the qualifiers not being satisfied in the formula. Then, program is type checked again with the new substitution. This process is repeated until a solution is found. Since the set of conjunctions of elements of $\mathbb{Q}^*$ is finite, the algorithm is guaranteed to terminate.

3 Decidable Theories on Arrays

As explained before, when working with liquid types, refinements should be formalized using formulas whose satisfiability could be provable. Therefore, it is important to know which theories concerning arrays are decidable, in order to use formulas of such theories to specify array properties. First studies involving satisfiability decision procedures for array theories have focused on quantifier-free fragments [16], as the full theories are undecidable. Later, an extensional theory of arrays with equality between unbounded arrays has been formalized as a decidable fragment [15]. An extension of these theories is studied in [2]. The motivation is that most assertions and invariants of programs related to arrays require at least a universal quantifier over index variables. Usual array properties can be formalized by formulas having the form $(\forall \overline{j}.\varphi_I(\overline{j}) \rightarrow \varphi_V(\overline{j}))$ where $\overline{j}$ is a vector of index variables, the guard $\varphi_I(\overline{j})$ delimits the segment of the array we are interested in, while $\varphi_V(\overline{j})$ refers to the value constraint. Both the guard and the value constraint involve predicates referring to program variables.

In order to have a satisfiability procedure for universal quantified formulas with that shape, some limitations are imposed to the syntax of $\varphi_I(\overline{j})$ and $\varphi_V(\overline{j})$. These limitations restrict the set of predicates that can be used to build those formulas, but most of the common program invariants referring arrays can be expressed with the restricted set, as we will see. The form of an index guard $\varphi_I(\overline{j})$ is constrained according to the grammar:

$$\begin{aligned}
\mathit{guard} &::= \mathit{guard} \wedge \mathit{guard} \mid \mathit{guard} \vee \mathit{guard} \mid \mathit{atom} \\
\mathit{atom} &::= \mathit{expr} \leq \mathit{expr} \mid \mathit{expr} = \mathit{expr} \\
\mathit{expr} &::= \mathit{uvar} \mid \mathit{pexpr} \\
\mathit{pexpr} &::= z \mid z * \mathit{evar} \mid \mathit{pexpr} + \mathit{pexpr}
\end{aligned}$$

where z stands for Presburger arithmetic basic terms (i.e. terms built up from the constants 0, 1 and the functions + and −), *uvar* represents variables that will occur universally quantified, and *evar* represents integer variables that will occur existentially quantified. Notice that the relations $\neq$ and $<$ are not allowed between quantified indices, and that they cannot be simulated by using $\leq$ because terms like $j + 1$ are not valid in *pexpr* if j is a universally quantified variable. However, we will write $j < b$, where j is quantified and b is in *pexpr*, as an abbreviation of $j \leq b - 1$, which is allowed if b is not quantified.

The formula $\varphi_V(\bar{j})$ is constrained in such a way that any occurrence of a quantified variable $j \in \bar{j}$ must be as a read into an array, $a[j]$, for array term a, and nested array reads are not allowed. Other program variables and terms can occur everywhere in the formula. A formula of the form $(\forall \bar{j}.\varphi_I(\bar{j}) \rightarrow \varphi_V(\bar{j}))$ with the previous constraints for $\varphi_I(\bar{j})$ and $\varphi_V(\bar{j})$ is called an *array property*.

The theory consisting in all existentially-closed Boolean combinations of array properties, and quantifier-free formulas built from program variables and terms is decidable. However, when considering existentially-closed $\forall$-$\exists$-fragments, even with syntactic restrictions like those in the array property, the satisfiability problem becomes undecidable. Other theories also proved undecidable are the following extensions of the array property formulas: If the formula contains nested reads as $a_1[a_2[j]]$ and j is universally quantified, or if $a[j]$ appears in the guard and j is universally quantified, or if the formula includes general Presburger arithmetic expressions over universally quantified index variables (e.g., $j + 1$) in the index guard or in the value constraint.

4 Array Refinements

In order to bound the decidable fragment of the array theory, we realize that most of the array properties fall in some of the following categories:

– Some elements of an array satisfy individually a property. For example:

$$\forall j.0 \leq j < \mathit{len}\ v \land j\%2 = 0 \rightarrow v[j] > 0 \tag{2}$$

$$\forall j.a \leq j \leq b \rightarrow x < v[j] \land v[j] \leq y \tag{3}$$

– Some pairs of elements in a segment of an array satisfy a binary relation:

$$\forall j_1, j_2.a \leq j_1 < j_2 \leq b \rightarrow v[j_1] \neq v[j_2] \tag{4}$$

$$\forall j_1, j_2.a \leq j_1 \leq p \land p \leq j_2 \leq b \rightarrow v[j_1] \leq v[j_2] \tag{5}$$

Property (5) holds after partition in *quicksort*, being p the pivot position. Sometimes the binary relation concerns two different arrays. For example:

$$\forall j_1, j_2.a \leq j_1 \leq k - 1 \land i \leq j_2 \leq m \rightarrow v[j_1] \leq w[j_2] \tag{6}$$

is a property that holds while merging the two sorted halves $[a, m]$ and $[m + 1, b]$ of an array w into an ordered array v (see Example 3).

– Usually we also need properties related to the length of the array in order to guarantee that the array accesses are well defined. For instance, the property (3) can be completed with $(0 \leq a < \mathit{len}\ v) \land (0 \leq b < \mathit{len}\ v)$.

Some formulas listed above do not belong to the decidable fragment mentioned in the previous section. In particular, (2) is not in the fragment because operators over the quantified variables are not allowed, and (4) is not an array property, because relation $<$ is not allowed over the quantified indices. The remaining formulas are allowed[1], even more, they belong to a subset of the fragment that we are going to characterize in our formalization of array refinements.

[1] We consider *len* v to be a fixed integer rather than a function applied to v.

$\{0 \leq n < \mathit{len}\ v \wedge \mathit{ord}(v, 0, n-1)\}$

```
i = n-1; x = v[n];
while (i >= 0 && x < v[i])
  {v[i+1] = v[i]; i = i-1;}
v[i+1] = x;
```

$\{\mathit{ord}(v, 0, n)\}$

Fig. 1. *insert* algorithm

$\{\mathit{ord}(v, 0, \mathit{len}\ v - 1)\}$

```
a = 0; b = (len v) - 1;
while (a<=b)
 { m = (a+b)/2;
   if (v[m] < x) {a = m+1;}
   else {b = m-1;}  }
```

$\{\mathit{lt}(v, x, 0, a) \wedge \mathit{geq}(v, x, a, \mathit{len}\ v)\}$

Fig. 2. *binSearch* algorithm

$\{0 \leq a \leq m \leq b < \mathit{len}\ v \ \wedge \mathit{ord}(w, a, m) \wedge \mathit{ord}(w, m+1, b)\}$

```
i = a; j = m+1;k = a;
while (i <= m && j <= b)
  { if (w[i] <= w[j]) { v[k] = w[i]; i=i+1; }
    else  { v[k] = w[j]; j=j+1; }
    k=k+1; }
while (i <= m) {v[k]=w[i]; i=i+1; k=k+1;}
while (j <= b) {v[k]=w[j]; j=j+1; k=k+1;}
```

$\{\mathit{ord}(v, a, b)\}$

Fig. 3. *merge* algorithm

Considering these three kinds of array properties, we establish three kinds of refinements with the aim of inferring automatically array properties. We consider that they widely cover many of the invariants needed to verify programs dealing with arrays, including the most known sorting algorithms, as we will show in Sec. 6. We will call them respectively *simple* array refinements (denoted as ρ), *double* array refinements (denoted as $\rho\!\rho$) and *length* refinements (denoted as ψ).

Simple refinements have the shape $\rho(w) \equiv \forall j. I(j) \rightarrow E(w[j])$, where w is an array. In the liquid type this will be the array being refined, i.e. ν. Predicate I restricts the values of the indices whose elements satisfy the property, and E expresses the individual property satisfied by each considered element. The qualifiers allowed in both of them are constrained as explained in Sec. 3 to ensure decidability, and belong to the sets of qualifiers which are provided by the programmer. As we may have several simple refinements, we can consider predicate I to be just a conjunction of qualifiers due to the logical equivalence $(A \vee B) \rightarrow C \Leftrightarrow (A \rightarrow C) \wedge (B \rightarrow C)$. In order to reduce the search space in the inference process we have decided E to be a conjunction of qualifiers[2]. Due to the logical equivalence $A \rightarrow (B \wedge C) \Leftrightarrow (A \rightarrow B) \wedge (A \rightarrow C)$, we can consider that in fact E is a single qualifier. Note that the previous predicate (3) is a valid simple refinement.

Double refinements have the shape $\rho\!\rho(v, w) \equiv \forall j_1, j_2. I\!I(j_1, j_2) \rightarrow E\!E(v[j_1], w[j_2])$, where v, w are array variables. In the liquid type, at least one of them will

[2] This does not preclude that a qualifier could be a disjunction of atomic properties.

be the refined array ν, and in case the other is not, it has to be a free in scope variable. Predicate *II* restricts the values of the pairs of indices, and *EE* expresses the relation satisfied by each considered pair. Both of them must meet the constraints of the array property formulas. Similarly to simple refinements, *II* is a conjunction of qualifiers and *EE* is a single qualifier. Note that examples (5) and (6) are valid double refinements.

Length refinements are qualifiers relating the length of the array to other values or program variables, such as $a < \mathit{len}\ \nu$ or $\mathit{len}\ \nu = \mathit{len}\ w$.

Definition 1. *A* refined array type *has the following shape:*

$$\{\nu : \mathit{array}\ \tau \mid (\bigwedge_i \psi_i(\nu)) \wedge (\bigwedge_j \rho_j(\nu)) \wedge (\bigwedge_k \rho\rho_k(\nu, v_k))\}$$

where each v_k may be ν itself or a free array variable.

Example 1 Fig. 1 shows the specification and the imperative code corresponding to the algorithm *insert* used in the insertion sort, where $\mathit{ord}(v,l,r) \equiv \forall j_1, j_2 . l \leq j_1 \leq j_2 \leq r \rightarrow v[j_1] \leq v[j_2]$. The property $\forall j . i+2 \leq j \leq n \rightarrow x < \nu[j]$ is part of the refinement of array v in line 2, i.e. it is an invariant property of the loop. □

Example 2 Fig. 2 shows the specification and the imperative code corresponding to the binary search algorithm, where $\mathit{lt}(v,x,l,r) \equiv \forall j . l \leq j < r \rightarrow v[j] < x$ and $\mathit{geq}(v,x,l,r) \equiv \forall j . l \leq j < r \rightarrow x \leq v[j]$. The property $\mathit{geq}(v,x,b+1,\mathit{len}\ v)$ is part of the refinement of array v in line 2, i.e. it is an invariant property of the loop. □

Example 3 In Fig. 3 we show the specification and the imperative code corresponding to the algorithm *merge* used in the mergesort algorithm. The property

$$\begin{array}{l}(\forall j_1, j_2 \,.\, a \leq j_1 \leq k-1 \wedge i \leq j_2 \leq m \rightarrow \nu[j_1] \leq w[j_2]) \wedge \\ (\forall j_1, j_2 \,.\, a \leq j_1 \leq k-1 \wedge j \leq j_2 \leq b \rightarrow \nu[j_1] \leq w[j_2])\end{array}$$

is part of the refinement of array v in line 2, i.e. it is an invariant property of the first loop. It is also part of v's refinement in the second and third loops. □

5 The Type Inference Algorithm

The inference algorithm has the following phases:

1. A standard type checking algorithm decorates every variable with its conventional type. Our IR includes types at every defining occurrence. The type checking propagates this information to every applied variable occurrence.

2. Each type occurrence is then refined with a *liquid template* (see below) of the appropriate type. The template refining a type occurrence introduces a fresh liquid variable. The purpose of the inference algorithm is to find appropriate substitutions for these liquid variables.

3. The syntax-driven liquid typing rules of the IR are applied, and a set of *constraints* is obtained. These are to be satisfied in order the program be correctly typed in the liquid type sense. A constraint has the form $[\![\Gamma]\!] \wedge \theta_1.\iota_1 \Rightarrow \theta_2.\iota_2$, where ι_1 and ι_2 are liquid variables and θ_1, θ_2 *pending substitutions*, as in [13]. The purpose of the pending substitutions is to replace formal arguments by actual ones in function applications. In our IR, actual arguments are always variables. The liquid typing rule for application is as follows:

$$\frac{\Gamma \vdash e : (x : T_x \to T) \qquad \Gamma \vdash y : T_x}{\Gamma \vdash e\ y : T[y/x]}$$

4. The constraints are solved by an *iterative weakening* algorithm. The algorithm starts with the strongest possible mapping A for all the liquid variables, and at each step, a variable assignment is weakened in order to satisfy a constraint. If a fixpoint is reached, then the final mapping obtained, when applied to all the templates, gives us the liquid types for all the variables.

5.1 Liquid templates

The liquid types of the variables x that are not arrays are represented by a liquid variable κ with pending substitutions θ, as usual: $x : \{\nu : \tau \mid \theta.\kappa\}$. The range of $A(\kappa)$ are conjunctions of qualifiers taken from the set $\mathbb{Q}^*$, which is obtained from $\mathbb{Q}$ at each program location by substituting program variables in scope of the appropriated type for the wildcard $\star$. After applying A, the pending substitution θ is applied to the result.

The liquid types of the variables a of array type are dealt with similarly, except for the fact that we denote the liquid variable by μ, $a : \{\nu : \mathit{array}\ \tau \mid \theta.\mu\}$. In this case we assume that the programmer provides several qualifier sets $\mathbb{Q}_E$, $\mathbb{Q}_{EE}$, $\mathbb{Q}_I$, $\mathbb{Q}_{II}$ and $\mathbb{Q}_{len}$, explained in detail below. The range of $A(\mu)$ are *array refinements* obtained from conjunctions of *array refinements templates* by substitution. These templates may be:

- Simple array refinement templates, $\rho \stackrel{\text{def}}{=} (\forall j.\eta \to q)$, where q is a qualifier taken from the set $\mathbb{Q}^*_E$, and η is a liquid variable.
- Double array refinement templates, $\rho\!\rho \stackrel{\text{def}}{=} (\forall j_1, j_2.\eta\!\eta \to q)$, where q is a qualifier taken from the set $\mathbb{Q}^*_{EE}$ and $\eta\!\eta$ is a liquid variable.
- An array length refinement template ζ. This liquid variable represents properties restricting the length of the array.

We will use ξ to denote both a simple and a double array refinement template, so $A(\mu) = (\bigwedge_{i=1}^{n} A(\xi_i)) \wedge A(\zeta)$, where $A(\rho) = \forall j.A(\eta) \to q$, and $A(\rho\!\rho) = \forall j_1, j_2.A(\eta\!\eta) \to q$. The range of $A(\eta)$, $A(\eta\!\eta)$ and $A(\zeta)$ are conjunctions of qualifiers taken respectively from the sets $\mathbb{Q}^*_I$, $\mathbb{Q}^*_{II}$, and $\mathbb{Q}^*_{len}$. Only variables in scope are considered on these instances of the respective qualifier sets $\mathbb{Q}_E$, $\mathbb{Q}_{EE}$, $\mathbb{Q}_I$, $\mathbb{Q}_{II}$, $\mathbb{Q}_{len}$. These sets meet several constraints which guarantee that, when wildcards are instantiated, then the obtained qualifiers satisfy the restrictions imposed on the array property formulas, e.g. $\mathbb{Q}_I$ and $\mathbb{Q}_E$ use $\star$ and $\#$ as wildcards in the qualifiers, and only the bound variable j can be substituted for the wildcard $\#$.

From now on, we will consider fixed the sets $\mathbb{Q}$, $\mathbb{Q}_I$, $\mathbb{Q}_{II}$, $\mathbb{Q}_E$, $\mathbb{Q}_{EE}$ and $\mathbb{Q}_{len}$ and we denote by $\mathcal{Q}$ the collection of these six sets.

Definition 2. *A mapping A is* suitable to $\mathcal{Q}$ *if it assigns a value of their respective ranges to each κ, μ, ζ, η, and $\eta\!\!\eta$ variables, and for each η variable of a ρ template, $A(\eta)$ contains $0 \le j < len\ \nu$, where j is the universal quantified variable in ρ, and for each $\eta\!\!\eta$ variable of a $\rho\!\!\rho$ template, $A(\eta\!\!\eta)$ contains $0 \le j_1 < len\ a \wedge 0 \le j_2 < len\ b$, where j_1, j_2 are the universal quantified variables in $\rho\!\!\rho$, a and b are either ν, or the free array variable in scope substituted for $\star$ in the qualifier at the right-hand side of $\rho\!\!\rho$. We denote by $\mathcal{A}_{\mathcal{Q}}$ the set of all the mappings suitable to $\mathcal{Q}$.*

The κ, μ, ζ variables occur in logically positive positions in the templates, while η, and $\eta\!\!\eta$ variables occur in negative ones. As a consequence, weakening A may consist of weakening the assignment to a κ, a μ, or a ζ variable, or strengthening the assignment to a η or a $\eta\!\!\eta$ variable.

For any liquid variable ι, if Q is a set of qualifiers, or a set of array refinements, when we write $A(\iota) = Q$, Q denotes the conjunction of its elements. In the examples, we omit the component $0 \le j < len\ \nu$ of $A(\eta)$ when it is not relevant (analogously for $A(\eta\!\!\eta)$).

Example 4 In the *insert* algorithm of Fig. 1, from the template $(\forall j_1, j_2.\eta\!\!\eta \to q)$, and the sets $\mathbb{Q}_{II} = \{0 \le \#_1, \star + 2 \le \#_2, \#_1 \le \star, \#_2 \le \star\}$, and $\mathbb{Q}_{EE} = \{\nu[\#_1] \le \nu[\#_2]\}$, the predicate $\forall j_1, j_2. 0 \le j_1 \le i \wedge i + 2 \le j_2 \le n \to \nu[j_1] \le \nu[j_2]$ can be obtained. It is part of the refinement type for v. $\square$

5.2 The iterative weakening algorithm

Given a set C of constraints, and a collection $\mathcal{Q} = \{\mathbb{Q}, \mathbb{Q}_I, \mathbb{Q}_{II}, \mathbb{Q}_E, \mathbb{Q}_{EE}, \mathbb{Q}_{len}\}$, the purpose of the algorithm is to find a solution to C, in accordance to the following definition:

Definition 3. *Given $A \in \mathcal{A}_{\mathcal{Q}}$, we say that A* satisfies $c \in C$ *if $A(c)$ is a valid formula. We say that A is a* solution of C, *if the set $A(C)$ is a set of valid formulas, abbreviated $A(C)$ valid.*

Below we describe the steps of the weakening algorithm. It starts with the strongest possible mapping A suitable to $\mathcal{Q}$. This consists of:

1. For a κ variable, $A(\kappa)$ is the conjunction of all the well-typed qualifiers of $\mathbb{Q}^*$ containing variables in scope.
2. For a μ variable, $A(\mu)$ is the conjunction of as many instances $A(\rho)$ of ρ templates as well-typed qualifiers in $\mathbb{Q}_E^*$, and as many instances $A(\rho\!\!\rho)$ of $\rho\!\!\rho$ templates as well-typed qualifiers in $\mathbb{Q}_{EE}^*$. There is also an additional conjunction $A(\zeta)$ for qualifying the array length (with variables in scope in each case).

– For a ζ variable, $A(\zeta)$ is the conjunction of all the well-typed qualifiers of $\mathbb{Q}^*_{len}$ containing variables in scope.
– For the η variable of a ρ template, $A(\eta)$ is the weakest possible predicate, $0 \leq j < len\ \nu$, where j is the universally quantified variable in ρ.
– For the $\eta\!\eta$ variable of a $\rho\!\rho$ template, $A(\eta\!\eta)$ is the weakest possible predicate, $0 \leq j_1 < len\ a \wedge 0 \leq j_2 < len\ b$, where j_1, j_2 are the universally quantified variables in $\rho\!\rho$, a and b are either ν, or the free array variable in scope substituted for $\star$ in the qualifier at the right-hand side of $\rho\!\rho$.

Example 5 In the *binSearch* algorithm of Fig. 2, we have $\mathbb{Q}^*_E = \{x \leq \nu[j], x > \nu[j]\}$, $\mathbb{Q}^*_I = \{j \leq a-1,\ b+1 \leq j\}$ for the μ_3 variable corresponding to the array v at the beginning of each iteration. Then the refinement:

$$(\forall j\ .\ 0 \leq j \wedge j < len\ \ \nu \rightarrow x \leq \nu[j]) \wedge (\forall j\ .\ 0 \leq j \wedge j < len\ \ \nu \rightarrow x > \nu[j]) \qquad (7)$$

will be included in the strongest assignment to μ_3. □

At each iteration, the algorithm arbitrarily chooses a constraint $c \in C$ not satisfied by A. Then, A is *weakened* in order to make the constraint valid. If this is not possible, then the algorithm ends up with **failure**. If this is possible, A is replaced by its weakened form A', and the set C of constraints is inspected again looking for a new unsatisfied constraint. Because A has changed, some prior satisfied constraints may have turned into unsatisfied ones. If no unsatisfied constraint remains, then the algorithm ends up with **success**. The final mapping A, when applied to all the templates, and then applying the pending substitutions, gives the liquid type of each program variable.

The crucial step is then how to weaken the mapping A in order to satisfy a constraint c. Differently to the standard algorithm of [13], weakening A in our case may change the constraints themselves, and may introduce new liquid variables. Let us see the process in detail:

1. If c has the form $[\![\Gamma]\!] \wedge \theta_1.\iota \Rightarrow \theta_2.\kappa$, where ι denotes either a κ or a μ variable, and $A(\kappa) = q_1 \wedge \cdots \wedge q_r$, then the weakening removes from $A(\kappa)$ all the qualifiers q_i such that the formula $A([\![\Gamma]\!]) \wedge \theta_1.A(\iota) \Rightarrow \theta_2.q_i$ is not valid. This approach corresponds to the standard weakening of [13]. The ζ variable of an array refinement is dealt with exactly in the same way as a κ variable, so in what follows we will not insist in these ζ variables.
2. If c has the form $[\![\Gamma]\!] \wedge \theta_1.\iota \Rightarrow \theta_2.\mu$, and $A(\mu) = A(\xi_1) \wedge \cdots \wedge A(\xi_r)$, in a first step the weakening removes from $A(\mu)$ all the refinements $A(\xi_i)$ such that the formula $A([\![\Gamma]\!]) \wedge \theta_1.A(\iota) \Rightarrow \theta_2.A(\xi_i)$ is not valid and cannot not be made valid. If the formula is not valid, then it is tested whether it can be made valid by strengthening the antecedent of $A(\xi_i)$. To do this, the η or $\eta\!\eta$ variable of ξ_i is assigned the strongest possible value, i.e. the conjunction of all the qualifiers of its respective $\mathbb{Q}^*_I$ or $\mathbb{Q}^*_{II}$ set. This assignment makes the instance of ξ_i as weak as possible. If, in spite of being that weak, the formula is not valid, then $A(\xi_i)$ is discarded from $A(\mu)$.

3. For each not valid $A(\xi_i)$ in $A(\mu)$ which can be made valid by strengthening its antecedent as explained before, a search for the strongest possible valid forms of the ξ_i instance is performed. Let us assume for a moment that ξ_i is a simple refinement template ρ_1 of the form $\forall j.\eta_1 \rightarrow q$, and $A(\eta_1) = Q_1 \subseteq \mathbb{Q}_I^*$. The discussion would be similar for a double one. Conjunctions m_j of $|Q_1|+1$, $|Q_1|+2$, $|Q_1|+3$, etc. qualifiers from $\mathbb{Q}_I^*$, all of them supersets of Q_1, are tried in this order as possible mappings for the η_1 variable of ρ_1, until one of them, let us call it m_1, makes the formula valid. Then the algorithm refrains from trying any superset of m_1, instead, it continues the search by trying the rest of the conjunctions. It may be the case that more than one conjunction (excluding their respective supersets) succeeds. Let them be $m_2, \ldots, m_s$. Then, fresh copies of ρ_1, call them $\rho_2, \ldots, \rho_s$, of the form $\forall j.\eta_l \rightarrow q$, with η_l fresh variables $l = 2..s$, are created. Now A' is defined from A changing the component $A(\xi_i)$ of $A(\mu)$ by the conjunction $A'(\rho_1) \wedge \cdots \wedge A'(\rho_s)$, where $A'(\eta_1) = m_1, \ldots, A'(\eta_s) = m_s$.

Example 6 In the *binSearch* algorithm, the following constraint establishes the correctness of the initial iteration:

$$x : \kappa_1 \wedge v : \mu_1 \wedge a = 0 \wedge b = (len\ \ v) - 1 \Rightarrow v : \mu_3$$

This constraint is not valid under the initial assignment to μ_3 given in (7), but it can be made valid by strenghtening its antecedent, since for instance the first conjunct of (7) becomes:

$$(\forall j\ .\ 0 \leq j \wedge j \leq a - 1 \wedge b + 1 \leq j \wedge j < len\ \ \nu \rightarrow x \leq \nu[j])$$

The search for supersets refines this predicate into the following two:

$$(\forall j\ .\ 0 \leq j \wedge j \leq a - 1 \rightarrow x \leq \nu[j]) \wedge (\forall j\ .\ b + 1 \leq j \wedge j < len\ \ \nu \rightarrow x \leq \nu[j])$$

which are both valid because the j ranges over two empty sets. The first conjunct will disappear from the μ_3 assignment in subsequent weakenings. □

5.3 Soundness and completeness

We have proven the following properties of the inference algorithm:

1. The algorithm always terminates.
2. If the algorithm terminates with **failure**, there exists no mapping A satisfying all the constraints.
3. If the algorithm terminates with **success**, the result mapping A satisfies all the constraints and it is the strongest possible mapping satisfying them.

The proof starts by showing that the search space, i.e the set $\mathcal{A}_\mathcal{Q}$ of mappings, is a complete lattice, with the following definition of $\sqsubseteq$.

Definition 4. *Let $A, A' \in \mathcal{A}_\mathcal{Q}$. We say that $A \sqsubseteq A'$ if for all κ, $A(\kappa) \Rightarrow A'(\kappa)$ and for all μ, $A(\mu) \Rightarrow A'(\mu)$.*

Theorem 1 The partial ordered set $(\mathcal{A}_{\mathcal{Q}}, \sqsubseteq)$ is a (finite) complete lattice.

Sketch of the Proof: Since the liquid variables are mapped to conjunctions of formulas, the empty conjunction *true* is the weakest one, corresponding to the $\top$ of the lattice. The strongest possible mapping is the initial one A_0, i.e. $\bot = A_0$. It is easy to prove that the following definition $(A_1 \sqcap A_2)(\iota) = A_1(\iota) \cup A_2(\iota)$ makes $\sqcap$ a greatest lower bound, and the lowest upper bound can be defined in terms of $\sqcap$ in a standard way. Since all the $\mathbb{Q}^*$ sets are finite, so it is the set of formulas, and also the set $\mathcal{A}_{\mathcal{Q}}$ of mappings. $\square$

Moreover, the following theorem shows that each step of the weakening algorithm produces an output mapping weaker than the input one.

Theorem 2 Let $A \in \mathcal{A}_{\mathcal{Q}}$. If A' is obtained from A by one step of the inference algorithm, then $A' \in \mathcal{A}_{\mathcal{Q}}$ and $A \sqsubseteq A'$. $\square$

The following two theorems allow to prove that, if a solution exists for C, then the algorithm terminates in a finite number of steps, and gives the strongest mapping A^s as a result.

Theorem 3 Given a set C of constraints, if there exists a mapping $A \in \mathcal{A}_{\mathcal{Q}}$ such that A is a solution of C, then there exists a minimum mapping $A^s \in \mathcal{A}_{\mathcal{Q}}$ such as $A^s(C)$ is valid.

Sketch of the Proof: As the set of mappings making C valid is finite, it is enough to show that for every pair of mappings A_1, A_2 making C valid, their greatest lower bound $A_1 \sqcap A_2$ is also a solution of C. $\square$

Theorem 4 If the set C of constraints has a strongest solution $A^s \in \mathcal{A}_{\mathcal{Q}}$, and A is a mapping produced by the inference algorithm, then $A \sqsubseteq A^s$.

Proof: By induction on the number of weakening steps of the algorithm. $\square$

6 Implementation and Results

We have implemented our tool in two separated phases. The first one, called *Template Generator*, traverses the program text previously transformed to the platform IR, applies the typing rules, and generates the set of relevant constraints that should be valid in order the program to be well-typed in the liquid-type sense. These constraints contain κ and μ variables for respectively the unknown basic and array types. The second phase is properly the type inference algorithm explained in Sec. 5.2. It searches for a substitution of the κ and μ variables that will make all the constraints valid. It uses the Why3 platform [5] and its SMT solvers as the underlying proving machinery.

We have applied the tool to an assorted set of array algorithms, including several sorting ones, or pieces of them, the binary search in a sorted array, a simple linear search, the Dutch National Flag algorithm [4, pp. 111–116], and the *fill* algorithm filling an array with a fixed value. Some of them are iterative,

Function	**Array**	**Inferred liquid type**	**#C**	**#F**
fill	v (loop)	$\{\nu : array\ \alpha \mid (\forall j \,.\, 0 \le j < i \to \nu[j] = x) \wedge (i \le len\ \nu)$	13	49
insert	v (loop)	$\{\nu : array\ \alpha \mid (\forall j \,.\, i+2 \le j \le n \to x < \nu[j])$	30	593
		$\wedge(\forall j_1, j_2 \,.\, 0 \le j_1 \le j_2 \le i \to \nu[j_1] \le \nu[j_2])$		
		$\wedge(\forall j_1, j_2 \,.\, i+2 \le j_1 \le j_2 \le n \to \nu[j_1] \le \nu[j_2])$		
		$\wedge(\forall j_1, j_2 \,.\, 0 \le j_1 \le i \wedge i+2 \le j_2 \le n \to \nu[j_1] \le \nu[j_2])$		
		$\wedge(n < len\ \nu)\}$		
merge	v (1st loop)	$\{\nu : array\ \alpha \mid (\forall j_1, j_2 \,.\, a \le j_1 \le k-1 \wedge i \le j_2 \le m \to \nu[j_1] \le w[j_2])$	88	1.278
		$\wedge(\forall j_1, j_2 \,.\, a \le j_1 \le k-1 \wedge j \le j_2 \le b \to \nu[j_1] \le w[j_2])$		
		$\wedge(\forall j_1, j_2 \,.\, a \le j_1 \le j_2 \le k-1 \to \nu[j_1] \le \nu[j_2])$		
		$\wedge(a < len\ \nu) \wedge (b < len\ \nu)\}$		
partition	v (loop)	$\{\nu : array\ \alpha \mid (\forall j_1, j_2 \,.\, a+1 \le j_1 \le i-1 \wedge j_2 = a \to \nu[j_1] \le \nu[j_2])$	44	287
		$\wedge(\forall j_1, j_2 \,.\, j_1 = a \wedge j+1 \le j_2 \le b \to \nu[j_1] \le \nu[j_2]$		
		$\wedge(a < len\ \nu) \wedge (b < len\ \nu)\}$		
quicksort	v (before	$\{\nu : array\ \alpha \mid (\forall j_1, j_2 \,.\, a \le j_1 \le p-1 \wedge j_2 = p \to \nu[j_1] \le \nu[j_2])$		
	2nd call)	$\wedge(\forall j_1, j_2 \,.\, j_1 = p \wedge p+1 \le j_2 \le b \to \nu[j_1] \le \nu[j_2])$	18	203
		$\wedge(\forall j_1, j_2 \,.\, a \le j_1 \le j_2 \le p-1 \to \nu[j_1] \le \nu[j_2])$		
		$\wedge(a < len\ \nu) \wedge (b < len\ \nu)\}$		
selsort	v (outer	$\{\nu : array\ \alpha \mid (\forall j_1, j_2 \,.\, 0 \le j_1 \le j_2 < i \to \nu[j_1] \le \nu[j_2])$	30	233
	loop)	$\wedge(\forall j_1, j_2 \,.\, 0 \le j_1 < i \wedge i \le j_2 < len\ \nu \to \nu[j_1] \le \nu[j_2])$		
		$\wedge(i \le len\ \nu)\}$		
selsort	v (inner	$\{\nu : array\ \alpha \mid (\forall j_1, j_2 \,.\, j_1 = min \wedge i \le j_2 < j \to \nu[j_1] \le \nu[j_2])$		
	loop)	$\wedge(i \le len\ \nu) \wedge (j \le len\ \nu)\}$		
binSearch	v (loop)	$\{\nu : array\ \alpha \mid (\forall j_1, j_2 \,.\, 0 \le j_1 \le j_2 < len\ \ \nu \to \nu[j_1] \le \nu[j_2])$	25	206
		$\wedge(\forall j \,.\, 0 \le j \le a-1 \to x > \nu[j]) \wedge (\forall j \,.\, b+1 \le j < len\ \ \nu \to x \le \nu[j])\}$		
		$\wedge 0 \le a \le b+1 \le len\ \ v$		
linSearch	v (loop)	$(i \le len\ \nu) \wedge (\forall j \,.\, 0 \le j \le i-1 \to \nu[j] \ne x)$	19	193
DutchFlag	v (loop)	$\{\nu : array\ \alpha \mid (\forall j \,.\, 0 \le j < len\ \ \nu \to \nu[j] = R \vee \nu[j] = W \vee \nu[j] = B)$	40	2.935
		$\wedge(\forall j \,.\, 0 \le j < a \to \nu[j] = R) \wedge (\forall j \,.\, a \le j < b \to \nu[j]) = W)$		
		$\wedge(\forall j \,.\, c < j < len\ \ \nu \to \nu[j] = W) \wedge c < len\ \ v\}$		

Fig. 4. Some of the liquid types inferred for assorted examples of array algorithms

and some other are recursive. As explained in the introduction section, after transformed to our IR, all of them are recursive. In some cases, they call to an external function that has been separately inferred. This poses no special problems to the inference algorithm.

We have provided the liquid types of the arguments and the results, i.e. the equivalent to the preconditions and the postconditions of the algorithms, and left the system to infer all the intermediate types. The *quicksort* algorithm does not include the code of *partition*. The qualifier sets used for inferring the types of these algorithms are variants of the following ones:

$$\begin{array}{ll}
\mathbb{Q} & = \{\mathit{constant} \le \nu,\ \star \le \nu,\ \nu \le \star,\ \nu < \star\} \\
\mathbb{Q}_E & = \{\star < \nu[\#],\ \mathit{constant} = \nu[\#], \star \ne \nu[\#]\} \\
\mathbb{Q}_I & = \{\star \le \#,\ \# \le \star,\ \star < \#,\ \# < \star\} \\
\mathbb{Q}_{EE} & = \{\nu[\#_1] \le \nu[\#_2],\ \nu[\#_1] \le \star[\#_2]\} \\
\mathbb{Q}_{II} & = \{\star \le \#_1,\ \#_1 \le \star,\ \star \le \#_2,\ \#_2 \le \star,\ \#_1 \le \#_2,\ \#_1 = \star,\ \#_2 = \star\} \\
\mathbb{Q}_{len} & = \{\star < len\ \nu,\ \star \le len\ \nu,\ \star < len\ \star,\ \star \le len\ \star\}
\end{array}$$

For the sets $\mathbb{Q}_I$ and $\mathbb{Q}_{II}$, the qualifiers $0 \le \#$, $\# < len\ \nu$, $0 \le \#_1$, $\#_1 < len\ \nu$, $0 \le \#_2$, and $\#_2 < len\ \nu$ are automatically introduced by the tool, so programmers do not need to provide them. Also, the algorithm removes fake formulas (e.g. $\forall j \,.\, a \le j < a \to v[j] > x$) that it can prove equivalent to *true*.

Some of the relevant types obtained are shown in Fig. 4. There, we have observed the types in text positions corresponding to entering a loop iteration, or entering a recursive call, which amounts to inferring the relevant invariants of the respective programs. With these inferred invariants all the algorithms have been proven correct by our tool.

Column $\#C$ records the number of constraints generated for the example, and column $\#F$ the number of formulas sent to the SMT solvers. Our current prototype is extremely slow: in order to prove a formula, two processes (Why3 and Z3) are, each time, started and stopped. Due to that, we can only process 10-12 formulas per second. This leads to times of up to 4 minutes in one of the examples. We are improving the tool by implementing a direct interface to Z3 via its API, which will process 500-1000 formulas per second. Then, the most complex example of Fig. 4 could be solved in about 5 seconds.

We make note that the properties inferred are in general far from being trivial. Up to five array refinements are needed in some cases to completely express the property kept invariant by a loop. We believe that these results are encouraging enough to continue exploring the power of liquid types to assist the programmer in the verification of complex array manipulating algorithms.

7 Related Work

The nearest works related to this paper are those about liquid types. These have been already reviewed in Sec. 2, and we have explained their limitations regarding universally quantified formulas.

A related technique to infer invariants of imperative programs is *predicate abstraction*, a variant of abstract interpretation which is also part of the liquid type approach. This was applied by [1] and [6]. The starting point is to have a finite set $Q = \{p_1, \ldots, p_n\}$ of atomic predicates in a decidable logic, from which more complex predicates can be built. In [6], the domain contains all combinations of the p_i by $\wedge$ and $\vee$, i.e. the set of all boolean functions with n boolean arguments, that is 2^{2^n} functions. The abstract interpretation of a loop proceeds in the forward direction, by using a strongest postcondition semantics. After each loop iteration, the predicate obtained is joined by $\vee$ to the one obtained in the prior iteration, and the result is abstracted by the abstraction function to that domain. Since this one is finite, a least fixpoint is always reached, provided the loop invariant can be effectively expressed by combinations of the given atomic predicates. If the algorithm succeeds, it obtains the strongest invariant belonging to the domain. They report experimenting their system with a Java program consisting of 520 loops, and were able to infer invariants for 98% of these loops.

In [8], they propose an abstract interpretation domain with universally quantified predicates. In prior attempts, quantification was introduced by rather *ad-hoc* means, but the abstract domain did not contain quantified formulas. After looking at the shape of many invariants, the authors propose the general form $E \wedge \bigwedge_{j=1}^{n} \forall U_j (F_j \Rightarrow e_j)$ where E, all the F_j, and all the e_j, are formulas belonging to non-quantified domains. Both E and the F_j are conjunctions of atomic predicates, and the e_j are just atomic ones. Each U_j is a tuple of (quantified) variables occurring free in F_j and e_j. An example of invariant is $1 \leq i \leq n \wedge \forall k(0 \leq k < i \Rightarrow a[k] = 0)$. The authors define an infinite lattice where the elements are formulas with this shape, define widening and narrowing operators to ensure termination, and also give some heuristics in order to convert non-quantified facts into quantified ones, when at least two iterations have been done during the interpretation of a loop. They infer invariants for most of the usual sorting algorithms, for finding an element in an array, and for other similar examples. The main differences with our approach are that our lattice is finite, so termination is guaranteed, and that we need neither widening nor heuristics.

In [14], the user gives a *template* formula for each particular invariant. In the template, the predicates are represented by unknowns that the system must guess. For instance, in $a \wedge \forall k(b \Rightarrow c)$ the system must find a substitution of concrete predicates for the variables a, b and c. The user must also supply a set Q of atomic predicates, conjunctions of which will replace the template unknowns. If an invariant exists having the template shape and formed by conjunctions of predicates from Q, then the algorithm finds the strongest one. The reported examples include invariants for all the sorting algorithms, the binary search in an array, list insertion, and list deletion. A difference with our approach is that decidability of the formulas is not guaranteed. The authors recognize that they sometimes provide their SMT solver with additional hints (triggers) in order to deal with undecidable quantified formulas. Additionally, they need to give a template with the exact number of quantified conjuncts, which is sometimes difficult to guess. Our algorithm generates as many conjuncts as needed to prove the correctness of the input program.

A last group of related papers is the temporal sequence [7, 9, 3], based on abstract interpretation. The main insight is the definition of an abstract domain for arrays, where they are considered to be split into a finite number of slices, and each slice satisfies a possibly different property. Its contents is represented by a single abstract variable that is updated as long as the algorithm progresses. They succeed in obtaining invariants for some array processing algorithms, the most complex of which is insertion sort. The approach is limited to single **for** loops, and to slices described by a predicate with only one universally quantified index. Also, they would be forced to change the abstract domain each time they wish to infer a different property. All the reported examples can be dealt with by our approach, and they admit that, at present, they cannot infer *quicksort*.

8 Conclusions

We have presented an extension of the Liquid type approach to universally quantified formulas about arrays. Arrays are non-recursive data structures and cannot be dealt with by using the recursive refinements introduced in [10]. Additionally, arrays are normally updated in-place and so used in imperative languages, while the Liquid type approach seems to fit better with functional ones. We have circumvented both obstacles: the first one, by allowing predicates on arrays where the indices can be universally quantified, and the second one, by using our verification platform which transform imperative programs into functional ones. The array refinements introduced in this paper try to cover properties satisfied for all the elements of an array segment and properties between pair of elements, either of the same array, or of two different ones. Algorithms searching arrays for a certain property are also covered, since their invariant can usually be expressed by a universal quantification (saying that no element of the array segment currently explored satisfies the property). As future work, we would like to generate at least a part of the qualifiers directly from the code, so liberating the programmer from most of this task.

We believe that other general refinements for arrays could be defined in order to cover programs in which certain elements of an array segment are counted or operated in some way. The resulting constraints should still be automatically proved valid by the current SMT solver technology. In this way, more decidable array invariants could be rescued from the general undecidable problem of invariant synthesis.

References

1. T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2001)*, pages 203–213, 2001.
2. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation: 7th International Conference, VMCAI 2006*, pages 427–442, Berlin, Heidelberg, 2006. Springer.
3. P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2011*, pages 105–118, 2011.
4. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
5. J.-C. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In M. Felleisen and P. Gardner, editors, *ESOP*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.
6. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In J. Launchbury and J. C. Mitchell, editors, *29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2002*, pages 191–202. ACM, 2002.
7. D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005*, pages 338–350, 2005.
8. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL 2008*, pages 235–246, 2008.
9. N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In R. Gupta and S. P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 339–348. ACM, 2008.
10. M. Kawaguchi, P. M. Rondon, and R. Jhala. Type-based data structure verification. In M. Hind and A. Diwan, editors, *PLDI*, pages 304–315. ACM, 2009.
11. M. Montenegro, R. Peña, and J. Sánchez-Hernández. A generic intermediate representation for verification condition generation. In *LOPSTR 2015*, pages 227–243, 2015.

12. N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program synthesis from polymorphic refinement types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*, pages 522–538, 2016.
13. P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In R. Gupta and S. P. Amarasinghe, editors, *PLDI*, pages 159–169. ACM, 2008.
14. S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In M. Hind and A. Diwan, editors, *PLDI*, pages 223–234. ACM, 2009.
15. A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for an extensional theory of arrays. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS 2001)*, pages 29–37. IEEE Computer Society Press, 2001.
16. N. Suzuki and D. Jefferson. Verification decidability of presburger array programs. *J. ACM*, 27(1):191–205, 1980.
17. N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In M. Felleisen and P. Gardner, editors, *ESOP*, volume 7792 of *LNCS*, pages 209–228. Springer, 2013.
18. N. Vazou, E. L. Seidel, and R. Jhala. LiquidHaskell: experience with refinement types in the real world. In *ACM SIGPLAN symposium on Haskell 2014*, pages 39–51, 2014.
19. N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. P. Jones. Refinement types for Haskell. In *19th ACM SIGPLAN International Conference on Functional Programming, ICFP 2014*, pages 269–282, 2014.