Extending Liquid Types to Arrays

MANUEL MONTENEGRO, SUSANA NIEVA, RICARDO PEÑA, and CLARA SEGURA,
Universidad Complutense de Madrid

A liquid type is an ordinary Hindley-Milner type annotated with a logical predicate that states the properties satisfied by the elements of that type. Liquid types are a powerful tool for program verification, as programmers can use them to specify pre- and post conditions of their programs, whereas the predicates of intermediate variables and auxiliary functions are inferred automatically. Type inference is feasible in this context, as the logical predicates within liquid types are constrained to a quantifier-free logic to maintain decidability.

In this article, we extend liquid types by allowing them to contain quantified properties on arrays so that they can be used to infer invariants on array-related programs (e.g., implementations of sorting algorithms). Although quantified logic is, in general, undecidable, we restrict properties on arrays to a decidable subset introduced by Bradley et al. We describe in detail the extended type system, the verification condition generator, and the iterative weakening algorithm for inferring invariants. After proving the correctness and completeness of these two algorithms, we apply them to find invariants on a set of algorithms involving array manipulations.

CCS Concepts: • **Theory of computation** → **Invariants**; **Program verification**; *Logic and verification*; Program analysis;

Additional Key Words and Phrases: Dependent types, liquid types, invariant synthesis

ACM Reference format:
Manuel Montenegro, Susana Nieva, Ricardo Peña, and Clara Segura. 2019. Extending Liquid Types to Arrays. *ACM Trans. Comput. Logic* 21, 2, Article 13 (December 2019), 41 pages.
https://doi.org/10.1145/3362740

1 INTRODUCTION

A considerable amount of research has been carried out over the past decades in the context of verification platforms [1, 4, 11, 12, 21, 33]. These platforms allow a programmer to specify the behaviour of their programs and check whether their implementations work as intended. Obtaining provably correct programs is an attractive goal, but it comes at a price: verification requires manual intervention. In particular, the programmers have to provide invariants in their loops so that the verification platform can generate and check verification conditions accordingly. The latter step is

This work was partially funded by the Spanish Ministry of Economy and Competitiveness, State Research Agency, and the European Regional Development Fund under grant TIN2017-86217-R (MINECO/AEI/FEDER, EU) and by Comunidad de Madrid as part of the program S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the EU.
Authors' address: M. Montenegro, S. Nieva, R. Peña, and C. Segura, Computer Science School, Universidad Complutense de Madrid, Calle Profesor José García Santesmases 9, Madrid, 28040, Spain; emails: montenegro@fdi.ucm.es, nieva@ucm.es, {ricardo, csegura}@sip.ucm.es.

1529-3785/2019/12-ART13 $15.00
https://doi.org/10.1145/3362740

mechanical and can be automatically performed by a satisfiability modulo theories (SMT) solver as long as these conditions are kept in a decidable logic, but finding a suitable invariant requires insight in many cases. Fortunately, various techniques have been developed in the past few years, allowing some invariants to be synthesised without a programmer's assistance. Some of them [3, 8, 13–15, 18] are reviewed in Section 11. Logically qualified types, which we will refer to as *liquid types* [20, 27, 35–37] in this article, stand out among these approaches.

Liquid types are dependent types in the sense that they depend on the values computed by the program. A liquid type consists of an ordinary Hindley-Milner type with a logical predicate that refines the set of values denoted by that type. In this context, invariant synthesis amounts to inferring the predicates inside liquid types. The inference process tries to guess for each type different combinations of predicates until a correct combination is found. Each attempt involves the generation of verification conditions which are subsequently sent to the SMT solver to check whether the attempted combination was correct or not. In this way, the set of all possible combinations of predicates can be considered as a search space to be explored. To make this approach feasible, there are two main requirements:

- The set of logical predicates allowed inside liquid types has to be constrained so as to keep a finite number of possibilities, thus making the search space finite.
- The SMT solver in charge of checking the validity of each attempted combination has to be complete. In this way, each wrong attempt may guide the inference algorithm into pruning the search space. Completeness in an SMT solver can be achieved by restricting the logical predicates to a decidable logic.

Liquid types have been proven to be useful to synthesise nontrivial invariants. These include, for example, invariants on algorithms involving linear arithmetic and uninterpreted functions [27]. This idea is extended in further work [20] to support recursive data types such as binary search trees, sorted lists, and heaps. In the latter case, programmers are expected to provide a definition of the recursive data type that includes a predicate involving the immediate recursive constituents of a constructor application. For instance, a programmer can specify a subset of binary trees in which the left (respectively, right) child of a node contains a value lower (respectively, greater) than the value in that node, thus obtaining a specification of binary search trees. Although this work is carried out in the context of functional languages, liquid types have also been successfully applied to imperative languages, such as TypeScript [38], JavaScript [6], and C-like languages [2, 28].

The aim of the present work is to extend the set of allowed predicates so as to include properties on arrays. This kind of properties usually involves quantification over the indices of an array. For example, the fact that an array a is sorted can be expressed with the property $\forall i, j.\ 0 \leq i < j < \mathit{len}\, a \rightarrow a[i] \leq a[j]$. Quantified logic is undecidable in general, but we can restrict the set of properties to conjunctions of *array properties* as defined in Bradley et al. [5], thus maintaining decidability when checking the generated verification conditions. We could, in fact, reuse the original liquid type inference algorithm [27]. The only difference is that now, for each attempt, the generated verification conditions may contain quantified formulas. The algorithm would handle quantified formulas as atomic entities, and it would be the task of the SMT solver to get into their internal structure to determine the validity of the verification conditions. However, it is possible to improve the search for a correct combination of predicates if we allow the algorithm to manage the constituents of quantified formulas separately. That is why we introduce a novel extension of the liquid type inference algorithm that infers predicates occurring in the left-hand side of an implication, and splits a quantified formula on a given array segment into formulas involving smaller

segments. This constitutes a new form of weakening that has proved to be powerful enough to infer many complex invariants arising in programs dealing with arrays.

These contributions have been carried out in the context of the CAVI-ART verification platform [23]. This platform relies on an *intermediate representation* (IR), which is used as a target first-order functional language to which imperative and functional programs can be translated. The algorithms introduced in this work have been implemented and integrated in this platform.

This article extends previous work [22]. The new material addresses several technical aspects underlying our inference algorithm, such as a formal description of the type system as a set of rules, and a detailed definition of the verification condition generation (VCG) algorithm. Unlike previous work on liquid types, our VCG algorithm infers the most precise type for a given expression by introducing disjunctions and existential quantifiers in refinements. We prove that this VCG algorithm is correct and complete with respect to the type system. Another new contribution is a reworked tool that improves the somewhat prototypical implementation of our previous work [22]. Instead of generating plain-text conditions that were processed by the Why3 platform [11], and then discharged by a SMT solver, this tool now interfaces directly with the Z3 solver [24], leading to better execution times. New material includes a comparison of the performance of the tool when inferring invariants with and without a programmer's assistance.

This article is organized as follows. Section 2 introduces previous work on liquid types and their inference algorithm. Our extension involves the inclusion of formulas that belong to a decidable logic of array properties, which is reviewed in Section 3. Next, we survey in Section 4 the kind of array properties we are interested in and show that the logic of array properties introduced before is expressive enough for our purposes. In Section 5, we define the IR in which the analysis will be performed, and then we describe in Section 6 the type system lying the foundations of the formal development. After this, the two main phases of our type inference algorithm will be briefly sketched in Section 7 and further elaborated in Sections 8 and 9. The feasibility of our extension is assessed in Section 10 by a set of case studies. Finally, Section 11 introduces related work and Section 12 concludes.

2 LIQUID TYPES

In its most basic form, a liquid type [27] is a standard polymorphic Hindley-Milner type annotated with a logical predicate that constraints the values allowed by the former. More concretely, a basic liquid type has the form $\{\nu : \tau \mid \varphi\}$, where ν is a variable, τ is a basic Hindley-Milner type (integer or Boolean), and φ is a logical formula, also called *refinement*, which may depend on ν and other program variables in scope. The ν variable is bound inside the type $\{\nu : \tau \mid \varphi\}$. Intuitively, the type $\{\nu : \tau \mid \varphi\}$ represents those values b of type τ such that $\varphi[b/\nu]$ is valid modulo a given theory. For example, in the theory of quantifier-free linear arithmetic, $\{\nu : int \mid \nu \geq 0\}$ represents all non-negative integers, and $\{\nu : int \mid a \leq \nu < b\}$ denotes all integers contained within the interval $[a, b)$, being a and b program variables. In the following, and whenever the refinement is *true*, we shall abbreviate $\{\nu : \tau \mid true\}$ to τ.

To infer liquid types in a given program, we have to restrict the kind of formulas allowed in the refinements. In general, the choice of the underlying theory determines feasibility of type inference. In this system, type inference becomes decidable when there are finitely many choices for φ and all of those are kept into the theory of linear integer arithmetic with equality and uninterpreted functions (QF-EUFLIA). The finite set of refinements allowed is defined from a set of logical qualifiers. A *qualifier* q is a predicate that depends on ν and a placeholder variable (denoted by $\star$). Type inference assumes that a set $\mathbb{Q}$ of logical qualifiers has been fixed in advance. This set may have been given by the programmer or may have been automatically synthesized (e.g., by analysing the program). The larger this set, the more accurate refinements can be inferred, but the

larger the search space becomes. From this set, we can obtain another set $\mathbb{Q}^*$ with the instances of the logical qualifiers appearing in $\mathbb{Q}$. We say that q' is an instance of q if and only if q' results from replacing all placeholders in q by program variables. For example, $x - v \geq y$ is an instance of $\star - v \geq \star$. Notice that the several occurrences of a placeholder inside a qualifier can be replaced by different variables. As a result, type inference requires refinements inside liquid types to be conjunctions of qualifier instances obtained from $\mathbb{Q}^*$. Since there are finitely many variables in a program, if $\mathbb{Q}$ is finite, so will be $\mathbb{Q}^*$, and so will be the set of choices for the refinements inside liquid types. The resulting search space can be traversed in an orderly manner to prune infeasible choices. The details of the traversal are given in Rondon et al. [27], and they will be summarized later in Section 2.2.

As usual in dependent type systems, input parameters in functional type signatures have to be labelled with variables to be able to express the function's output in terms of the input values. For example, we can assign the following liquid type to a function subtracting two natural numbers:

$$\texttt{subtract} :: (x : \{v : int \mid v \geq 0\}) \to (y : \{v : int \mid 0 \leq v \leq x\}) \to \{v : int \mid v = x - y\}.$$

This signature specifies that the first parameter has to be non-negative and that the second one should be less than or equal to the former. In this case, the arithmetic theory allows the result of `subtract` to have type $\{v : int \mid v = x - y\}$ that uniquely characterizes the output in terms of the input.

2.1 Refinements on Arrays

The examples considered so far involve simple arithmetic properties on integers, but refinement predicates can also specify more complex properties involving data structures. For example, a function that obtains the first element of an array could have the following type:

$$\texttt{first} :: \forall\alpha.(x : \{v : array\,\alpha \mid len\,v \geq 1\}) \to \{v : \alpha \mid v = x[0]\},$$

where $len\,v$ denotes the length of the array v. These refinements fit into the underlying QF-EUFLIA theory provided that *len* and indexing on x are translated into uninterpreted function symbols. However, quantifier-free theories fall short when we have to specify more complex—and hence more interesting—properties on arrays. For example, the following declarations specify the type of a function `fill` that sets all of the positions of an array to a given value and a function `sort` for sorting an array:

$$\texttt{fill} :: \forall\alpha.(a : array\,\alpha) \to (x : \alpha) \to \{v : array\,\alpha \mid \forall i.0 \leq i < len\,v \to v[i] = x\}, \tag{1}$$

$$\texttt{sort} :: \forall\alpha.(a : array\,\alpha) \to \{v : array\,\alpha \mid (\forall i.0 \leq i \leq j < len\,v \to v[i] \leq v[j])\}. \tag{2}$$

The original work on liquid types [27] requires type refinements to be quantifier free for the sake of decidability when inferring types. Therefore, the types of `fill` and `sort` would be ill-formed in regards to this constraint. This issue is partially addressed in further work [20, 35], where the authors extend the type system by allowing types to be parametric on *refinement predicate variables*. This extended system supports types of the form $\tau\langle p\rangle$, where τ is a Hindley-Milner type (excluding function types) and p a refinement predicate variable. The parametric type $\tau\langle p\rangle$ denotes all elements x of type τ such that $p\,x$ holds. Whenever p is instantiated, it will be replaced by a concrete predicate of type $\tau \to bool$. For example, assume that we have a function `max` that computes the greatest of two given numbers. The following specification indicates that if the input parameters satisfy a given property p, so does the result.

$$\texttt{max} :: \forall(p :: int \to bool)\,.\,x : int\,\langle p\rangle \to y : int\,\langle p\rangle \to \{v : int\,\langle p\rangle \mid v \geq x \wedge v \geq y\}$$

Composite data structures can also be annotated with refinement predicate variables which can be later instantiated with n-ary relations ($n \geq 1$). For example, a list data type can be defined such that it is parametric with respect to a binary relation. More concretely, the type $[int]\langle r\rangle$ denotes those lists of integers in which the relation r holds between every element of the list and those appearing on its right. In this context, the list constructor instantiated to integer elements has the following signature:

$$(:) :: \forall(r :: int \to int \to bool)\,.\, x : int \to xs : [int\langle r\, x\rangle]\langle r\rangle \to [int]\langle r\rangle\,.$$

With this definition, the type $[int]\langle\leq\rangle$ can be used to denote increasingly sorted lists and $[int]\langle\neq\rangle$ to denote lists whose elements are pairwise distinct. These two examples involve relations that hold "regularly" along all elements of the list. Unfortunately, there are some properties on lists that are not fit for this type, particularly those properties involving list indices. For example, some properties which arise when verifying the insertion sort algorithm cannot be expressed such that only a given segment of a list is sorted, or that the elements in the first half of the list are smaller than the elements in the second half. These properties can be expressed and are inferred in our system (see function `insert` in Section 10).

As an alternative, a programmer may choose to use arrays instead of lists. In Vazou et al. [35], arrays have type $array\,\tau\,\langle dom, rng\rangle$, where τ is the Hindley-Milner type of the array elements, dom is a predicate of type $int \to bool$ that constrains the set of valid indices, and rng is a relation of type $int \to \tau \to bool$ which specifies the property to be satisfied by each element of the array. Thus, if x has type $array\,\tau\,\langle dom, rng\rangle$, and i is an integer number such that $dom\,i$ holds, then so does $rng\,i\,x[i]$. For example, the array $[0, 2, 4, 6, 8]$ can be given the type $array\,int\,\langle \lambda i.0 \leq i < 5, \lambda i\,x.x = 2i\rangle$. In this setting, the `get` and `set` functions for accessing and manipulating an array would have the following signatures:

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

where dom' abbreviates the predicate $\lambda k.dom\,k \wedge k \neq i$. Considering our `fill` function introduced earlier, the type shown in (1) would be still ill formed under this extension, but by using refinement predicate variables, the following type can be specified:

$$\begin{aligned}
&\texttt{fill} :: \forall\alpha.\forall(dom :: int \to bool).\forall(rng :: int \to \alpha \to bool).\\
&\qquad (a : array\,\alpha\langle dom, rng\rangle) \to (x : \alpha) \to array\,\alpha\langle dom, \lambda i.\lambda z.z = x\rangle.
\end{aligned}$$

In this way, a programmer can specify properties on array elements that also depend on their position. However, there is a limitation: the rng parameter characterizes each element by itself in terms of its index, without relating it to the remaining elements. Therefore, the type of the `sort` function shown in (2) cannot be expressed with this data type, as the result type of `sort` contains a property that should hold for every pair of elements in the array.

Another limitation of applying refinement predicate variables to arrays is that the subtyping relation may be affected by the way in which the data type is defined, regardless of whether the array type is built into the language, or it is defined by the user. Let us compare the type $array\,\tau\,\langle dom, rng\rangle$ with the type $\{v : array\,\tau \mid \forall i.dom\,i \to rng\,i\,v[i]\}$. If the type $array\,\tau\langle dom, rng\rangle$ is an alias for $(i : int\langle dom\rangle) \to \tau\langle rng\,i\rangle$, to prove that $array\,\tau\,\langle dom, rng\rangle$ is a subtype of $array\,\tau\,\langle dom', rng'\rangle$, the following formula has to be proved:

$$\forall i.\forall z.\;\; dom'\,i \wedge rng\,i\,z \Rightarrow dom\,i \wedge rng'\,i\,z.$$

The validity of this formula amounts to proving unsatisfiability of $\neg(dom'\ i \wedge rng\ i\ z \Rightarrow dom\ i \wedge rng'\ i\ z)$, which can be done by an SMT solver with a suitable quantifier-free logic. Notice that a sufficient condition for the preceding formula to hold is that $dom'\ i$ implies $dom\ i$ for every i and that $rng\ i\ z$ implies $rng'\ i\ z$ for every i and z. In other words, subtyping is contravariant with respect to dom and covariant with respect to rng. However, if arrays were implemented as a set of pairs of type $(i : int\langle dom\rangle, \tau\langle rng\ i\rangle)$, subtyping would be covariant with respect to both dom and rng, which is hard to justify with regards to the semantics of array types, since we could say that $array\ \tau\ \langle \lambda i.0 \leq i < 3, rng\rangle$ is a subtype of $array\ \tau\ \langle \lambda i.0 \leq i < 5, rng\rangle$ for every τ and rng. This means that it would be safe to use an array of length 3 wherever an array of length 5 is expected, which becomes unsafe, unless the programmer explicitly states that subtyping is contravariant in dom.[1] But if we allow quantified refinements and we want to prove that $\{\nu : array\ \tau \mid \forall i.dom\ i \rightarrow rng\ i\ \nu[i]\}$ is a subtype of $\{\nu : array\ \tau \mid \forall i.dom'\ i \rightarrow rng'\ i\ \nu[i]\}$, we have to send the following formula to the SMT solver:

$$(\forall i.dom\ i \rightarrow rng\ i\ \nu[i]) \Rightarrow (\forall i.dom'\ i \rightarrow rng'\ i\ \nu[i]).$$

A sufficient condition for this to hold is that $dom'\ i$ implies $dom\ i$ and that $rng\ i\ \nu[i]$ implies $rng'\ i\ \nu[i]$ for every i. We have contravariance with respect to the domain, which meets better the intuition of an array of length 5 being a subtype of an array of length 3. Moreover, this contravariance is obtained regardless of the specific implementation of arrays. However, to prove this formula, the underlying SMT solver has to deal with quantifier formulas, as the negation of the preceding formula cannot be transformed into another quantifier-free equisatisfiable formula.

2.2 Type System and Inference in Absence of Arrays

In this section, we shall give an overview on how liquid types are inferred in Rondon et al. [27]. The type system is defined as a set of syntax-directed typing rules for deriving judgements of the form $\Gamma \vdash e :: S$, where S is a functional type schema whose constituents are basic types of the form $\{\nu : \tau \mid \varphi\}$. Some of the assumptions in the typing derivation of a given judgement may be subtyping judgements, which are derived, in turn, with the help of a set of subtyping rules. One of these rules states that $\{\nu : \tau \mid \varphi_1\}$ is a subtype of $\{\nu : \tau \mid \varphi_2\}$ under a given environment Γ provided the formula $[\![\Gamma]\!] \wedge \varphi_1 \Rightarrow \varphi_2$ holds. In this case, $[\![\Gamma]\!]$ denotes a logic formula that expresses the assumptions in the environment. This formula does not only contain the refinements of every variable occurring in Γ but also the conditions that are known to hold in the corresponding context. The latter are accumulated in the environment while traversing conditional expressions. Thus, the type system is path sensitive.

Type checking. To check whether an expression has a given type, one can apply a set of syntax-directed typing and subtyping rules to obtain a derivation tree whose leaves are verification conditions of the form $[\![\Gamma]\!] \wedge \varphi_1 \Rightarrow \varphi_2$, as explained earlier. If these conditions are discharged by an SMT solver, then the expression is well typed.

Type inference. Assuming that Hindley-Milner types have been inferred in a previous phase, the type inference algorithm decorates each type τ in the derivation tree with a fresh template variable κ to get a type $\{\nu : \tau \mid \kappa\}$. Similarly to type checking, some verification conditions are generated from the premises of the derivation tree, but now these conditions contain template variables. Type inference amounts to finding a solution to the verification conditions—that is, a mapping A from template variables to subsets of $\mathbb{Q}^*$ such that all verification conditions hold. The first attempt is done with the strongest mapping, which assigns the conjunction of all elements in $\mathbb{Q}^*$ to each template variable (i.e., $A(\kappa) = \bigwedge \mathbb{Q}^*$ for every κ). If the verification conditions

[1] Although not mentioned in other works [35–37], *liquid Haskell* supports variance annotations in user-defined types.

hold under this assignment, then A is a solution. Otherwise, there exists a verification condition $A(\llbracket\Gamma\rrbracket) \wedge A(\kappa_1) \Rightarrow A(\kappa_2)$ that does not hold. The algorithm then removes conjuncts from $A(\kappa_2)$ (thus weakening the mapping) until the offending verification condition becomes valid. The remaining verification conditions are checked again under the updated assignment. This weakening process is repeated until a solution is found. Since $\mathbb{Q}^*$ is finite, termination is guaranteed.

3 DECIDABLE THEORIES ON ARRAYS

As explained earlier, 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, 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 [31], as the full theories are undecidable. Later, an extensional theory of arrays with equality between unbounded arrays has been formalized as a decidable fragment [30]. An extension of these theories is studied in Bradley et al. [5]. 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 to be specified are that an assertion ($\geq 0, \leq x, \ldots$) holds for all elements in a given index range, or that every pair of elements of a segment of the array satisfy a relationship ($\leq, \neq, =, \ldots$), or a comparison between the elements of the array and the elements of another one. These array properties can be formalized by formulas having the following form:

$$\forall \bar{j}.\varphi_I(\bar{j}) \rightarrow \varphi_V(\bar{j}), \tag{3}$$

where $\bar{j}$ is a vector of index variables, the guard $\varphi_I(\bar{j})$ delimits the segment of the array we are interested in, and $\varphi_V(\bar{j})$ refers to the value constraint. Both the guard and the value constraint involve qualifiers referring to program variables. For instance, if v, w are array variables and i is an integer variable, the equality of a segment of these arrays can be encoded by the following:

$$\forall j.0 \leq j < i \rightarrow v[j] = w[j]. \tag{4}$$

To have a satisfiability procedure for universal quantified formulas with the shape of (3), some limitations are imposed to the syntax of $\varphi_I(\bar{j})$ and $\varphi_V(\bar{j})$. These limitations restrict the set of qualifiers 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.

Following Bradley et al. [5], the form of an index guard $\varphi_I(\bar{j})$ 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 the variables that will occur universally quantified, and *evar* represents the integer variables that will occur existentially quantified. Notice that the relations $\neq$ and $<$ are not allowed between quantified indices, and they cannot be simulated using $\leq$ because terms as $j + 1$ 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 of all existentially closed Boolean combinations of array property formulas and quantifier-free formulas built from program variables and terms is decidable. The satisfiability procedure introduced in Bradley et al. [5] reduces satisfiability of a formula of such fragment to satisfiability of a quantifier-free formula in the combined theory of equality with uninterpreted functions, Presburger arithmetic, and the element theories (corresponding to the types of the program terms). Every universal quantified assertion is converted to a finite conjunction by instantiating the quantified index variables over a finite set of index terms. Restrictions on guards ensure that such an index set is finite.

However, when considering existentially closed ∀-∃-fragments, even with syntactic restrictions like those in the array property, the satisfiability problem becomes undecidable. Other theories are also proved in Bradley et al. [5] to be undecidable, and this is the case of 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
- If $a[j]$ appears in the guard and j is universally quantified
- 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.

In Habermehl et al. [17], a more powerful decidable logic on arrays is presented. For instance, accesses to array elements such as $a[i + k]$, where i is a universally quantified variable and k is a constant, are allowed. In addition, expressions such as $i \,\%\, 2 = 0$ may occur in the guard. Unfortunately, the decision procedure is based on Büchi automata, and conventional SMT solvers do no support them.

4 ARRAY REFINEMENTS

When verifying programs which deal with arrays, many properties we express about them require universal quantification as explained earlier. To bound the decidable fragment of the array theory we actually need, we observe that those properties often fall into some of the following general categories:

- Properties expressing that some elements of an array satisfy individually a property—for example,

$$\forall j.0 \leq j < len\, v \rightarrow v[j] = 0, \tag{5}$$

$$\forall j.0 \leq j < len\, v \wedge j\%2 = 0 \rightarrow v[j] > 0, \tag{6}$$

$$\forall j.a \leq j \leq b \rightarrow x < v[j] \wedge v[j] \leq y. \tag{7}$$

- Properties expressing that some pairs of elements in a segment of an array satisfy a binary relation.[2]—for example,

$$\forall j_1, j_2.a \leq j_1 \leq b \wedge a \leq j_2 \leq b \wedge j_1 \neq j_2 \rightarrow v[j_1] \neq v[j_2], \tag{8}$$

$$\forall j_1, j_2.0 \leq j_1 \leq j_2 < len\, v \rightarrow v[j_1] \leq v[j_2], \tag{9}$$

$$\forall j_1, j_2.a \leq j_1 \leq p \wedge p \leq j_2 \leq b \rightarrow v[j_1] \leq v[j_2]. \tag{10}$$

  The last property holds after partition in *quicksort*, with p being the resulting pivot position.

[2]Ternary or bigger relations are less frequent, and they make the inference process too expensive.

In addition, many times the binary relation concerns two different arrays—for example,

$$\forall j_1, j_2 . a \leq j_1 \leq k - 1 \wedge i \leq j_2 \leq m \rightarrow v[j_1] \leq w[j_2] \tag{11}$$

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 in the following).

- Usually, the preceding properties need to be completed with a property related to the length of the array (as $len\, v > x$) expressing the conditions on the ends of a segment so that the array accesses are well defined. For instance, the property (7) can be completed as follows:

$$(\forall j . a \leq j \leq b \rightarrow x < v[j] \wedge v[j] \leq y) \wedge \underbrace{0 \leq a < len\, v \wedge 0 \leq b < len\, v}_{\text{segment limits}} . \tag{12}$$

In particular, the properties inferred in the examples shown in Section 10 fall into these categories. These include different sorting properties inferred in several sorting and searching algorithms, and also properties describing the specific contents of an array (e.g., those inferred in the dutch flag algorithm). All of the properties can be defined on delimited valid segments of the involved arrays.

Some formulas listed earlier do not belong to the decidable fragment mentioned in the previous section. In particular, (6) is not in the fragment because there cannot be operators over the quantified variables, and (8) is not an array property because relation $\neq$ is not allowed over the indices. The remaining formulas are allowed,[3] and even more, they belong to a subset of the fragment that we are going to characterize in our formalization of array refinements.

Keeping in mind the preceding 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 show in Section 10. We will respectively call them simple array refinements (denoted as ρ), double array refinements (denoted as $\rho\rho$), and length refinements (denoted as δ).

Simple refinements can be expressed using universal quantification over a variable representing the indices of the array whose elements meet the property. The array refinements corresponding to these properties have the following shape:

$$\rho(w) \equiv \forall j . I(j) \rightarrow E(w[j]),$$

where w is the array to which the refinement refers. In the liquid type, this will be the array being refined (i.e., v). I is a predicate which restricts the values of the indices whose elements satisfy the property. E is a predicate which expresses the individual property satisfied by each considered element.

The qualifiers allowed in I and E are constrained as explained in the previous section to ensure decidability and belong to 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 $(A \vee B) \rightarrow C \Leftrightarrow (A \rightarrow C) \wedge (B \rightarrow C)$. For example, the property $\forall j . 0 \leq j < len\, v \wedge (j < a \vee j > b) \rightarrow v[j] \geq 0$ will be written as a conjunction of refinements:

$$(\forall j . 0 \leq j < len\, v \wedge j < a \rightarrow v[j] \geq 0) \wedge (\forall j . 0 \leq j < len\, v \wedge j > b \rightarrow v[j] \geq 0).$$

To reduce the search space in the inference process, we will also consider that E is a conjunction of qualifiers—that is, we do not allow disjunction of qualifiers. 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 an atomic property about

[3] We consider $len\ v$ to be a fixed integer rather than a function applied to v.

one element of the array. For example, the property $\forall j.0 \leq j < len\, v \rightarrow v[j] \geq 0 \wedge v[j] \leq 100$ is in fact expressed as the conjunction of two simple refinements:

$$(\forall j.0 \leq j < len\, v \rightarrow v[j] \geq 0) \wedge (\forall j.0 \leq j < len\, v \rightarrow v[j] \leq 100).$$

Then, the previous predicates (5) and (7) are also valid simple refinements. And notice that properties about one element of the array can be expressed with simple refinements. For example, the property $v[3] > 0$ can be expressed as $\forall j.j = 3 \rightarrow v[j] > 0$.

Double refinements can be expressed using universal quantification over two variables representing the pairs of indices of the array whose elements meet the relation. They have the following shape:

$$\rho\rho(v, w) \equiv \forall j_1, j_2.\mathit{II}(j_1, j_2) \rightarrow \mathit{EE}(v[j_1], w[j_2]),$$

where v, w are the variables representing the arrays to which the refined array is related. In the liquid type, at least one of them will be the refined array (i.e., v), and in case the other is not, it has to be a free variable which is in scope. II is a predicate which restricts the values of the pairs of indices whose elements meet the relation. EE is a predicate which expresses the relation satisfied by each considered pair. Of course, both II and EE must hold the constraints of the array property formulas. Similarly to simple refinements, II is a conjunction of qualifiers and EE is a single qualifier concerning two elements of the involved arrays. Note that the previous examples (9), (10), and (11) are valid double refinements.

A *length refinement* δ is a single qualifier referring the length of the array, such as $i \leq len\, v$ and $len\, v \leq len\, w$. In (12), several valid length refinements are used to delimit a segment $[a, b]$ on an array where a property holds.

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

$$\{v : array\, \tau \mid \underbrace{(\bigwedge_i \delta_i(v))}_{\text{length refinements}} \wedge \underbrace{(\bigwedge_j \rho_j(v))}_{\text{simple refinements}} \wedge \underbrace{(\bigwedge_k \rho\rho_k(v, v_k))}_{\text{double refinements}}\},$$

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

Example 1. Figure 1 shows the specification and the imperative code corresponding to the algorithm *insert* used in the insertion sort, where

$$ord(v, l, r) \equiv \forall j_1, j_2.l \leq j_1 \leq j_2 \leq r \rightarrow v[j_1] \leq v[j_2].$$

The property $\forall j.i + 2 \leq j \leq n \rightarrow x < v[j]$ is part of the refinement of array v in line 3—that is, it is an invariant property of the loop.

Example 2. Figure 2 shows the specification and the imperative code corresponding to the binary search algorithm, where

$$lt(v, x, l, r) \equiv \forall j.l \leq j < r \rightarrow v[j] < x$$
$$geq(v, x, l, r) \equiv \forall j.l \leq j < r \rightarrow x \leq v[j].$$

The property $geq(v, x, b + 1, len\, v)$ is part of the refinement of array v in line 3—that is, it is an invariant property of the loop.

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

$$(\forall j_1, j_2\ .\ a \leq j_1 \leq k - 1 \wedge i \leq j_2 \leq m \rightarrow v[j_1] \leq w[j_2])$$
$$\wedge(\forall j_1, j_2\ .\ a \leq j_1 \leq k - 1 \wedge j \leq j_2 \leq b \rightarrow v[j_1] \leq w[j_2])$$

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

```
1  i = n - 1;
2  x = v[n];
3  while (i >= 0 && x < v[i]) {
4     v[i + 1] = v[i];
5     i = i - 1;
6  }
7  v[i + 1] = x;
```

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

Fig. 1. *insert* algorithm.

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

```
1  a = 0;
2  b = (len v) - 1;
3  while (a <= b) {
4     m = (a + b) / 2;
5     if (v[m] < x) {
6        a = m + 1;
7     } else {
8        b = m - 1;
9     }
10 }
```

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

Fig. 2. *binSearch* algorithm.

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

```
1  i = a;
2  j = m + 1;
3  k = a;
4  while (i <= m && j <= b) {
5     if (w[i] <= w[j]) {
6        v[k] = w[i];
7        i = i + 1;
8     } else {
9        v[k] = w[j];
10       j = j + 1;
11    }
12    k = k + 1;
13 }
14 while (i <= m) {
15    v[k] = w[i];
16    i = i + 1;
17    k = k + 1;
18 }
19 while (j <= b) {
20    v[k] = w[j];
21    j = j + 1;
22    k = k + 1;
23 }
```

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

Fig. 3. *merge* algorithm.

is part of the refinement of array v in line 4—that is, 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 PROGRAMMING LANGUAGE OF THE CAVI-ART PLATFORM

In this work, we apply our extended liquid type system to programs written in the CAVI-ART IR [23]. The CAVI-ART platform is oriented towards program verification. A key aspect of it is the IR to which source programs, written in a variety of languages, can be transformed. Once programs have undergone this transformation, the remaining activities (invariant synthesis,

$$
\begin{array}{lcll}
a & ::= & c & \{\text{ constant }\} \\
 & | & x & \{\text{ variable }\} \\
be & ::= & a & \{\text{ atomic expression }\} \\
 & | & f\ \overline{a_i} & \{\text{ function application }\} \\
 & | & \langle \overline{a_i} \rangle & \{\text{ tuple construction }\} \\
 & | & C\ \overline{a_i} & \{\text{ constructor application }\} \\
e & ::= & be & \{\text{ binding expression }\} \\
 & | & \textbf{let}\ \langle \overline{x_i :: \tau_i} \rangle = be\ \textbf{in}\ e & \{\text{ let binding }\} \\
 & | & \textbf{letfun}\ \overline{def_i}\ \textbf{in}\ e & \{\text{ mutually recursive function definitions }\} \\
 & | & \textbf{case}\ x\ \textbf{of}\ \overline{alt_i} \mid [\_ \rightarrow e] & \{\text{ case distinction with optional default branch }\} \\
tldef & ::= & \textbf{define}\ \{\psi_1\}\ def\ \{\psi_2\} & \{\text{ top level function definition with pre- and post-conditions }\} \\
def & ::= & f\ (\overline{x_i :: \tau_i}) :: (\overline{y_j :: \tau_j}) = e & \{\text{ function definition }\} \\
alt & ::= & C\ \overline{x_i :: \tau_i} \rightarrow e & \{\text{ case branch }\} \\
\tau & ::= & \alpha & \{\text{ type variable }\} \\
 & | & T_c\ \overline{\tau_i} & \{\text{ type constructor application }\}
\end{array}
$$

Fig. 4. CAVI-ART IR syntax.

termination analysis, verification condition generation, and proving) can be performed in a language-independent way. While verifying programs, the generated verification conditions are sent to the Z3 solver [24], which supports array properties.

The syntax of the IR is shown in Figure 4. It is a desugared first-order functional language with tuples, data constructors, and a polymorphic type discipline. The restriction of being first order is due to the fact that the CAVI-ART platform was intended for assisted verification of programs, and most of this task is accomplished in the context of imperative languages, in which higher-order constructs are moderately used. We could extend our IR with lambda expressions, as it is done in the original liquid type framework, or else we could allow higher-order constructs in the source language and defunctionalize it during its transformation to our IR, by following the usual steps [26]. However, for the purpose of this work, higher order does not play any relevant role.

Because of its language-dependent nature, the set of type constructors is left unspecified. We assume that the type definitions specific to each language have been given to Z3 in advance. This also applies to the axiomatization of the behaviour of built-in functions of the source language. In this work, we assume that each of these axiomatizations can be translated into a liquid type, and that there exists a type constructor for the array data type with the functions `get` and `set` having the following types:

$$
\begin{aligned}
\texttt{get} :: \forall\alpha.(a : array\ \alpha) &\rightarrow (i : \{v : int \mid 0 \leq v \wedge v < len\, a\}) \rightarrow \{v : \alpha \mid v = a[i]\} \\
\texttt{set} :: \forall\alpha.(a : array\ \alpha) &\rightarrow (i : \{v : int \mid 0 \leq v < len\, a\}) \\
&\rightarrow (x : \alpha) \\
&\rightarrow \{v : array\ \alpha \mid v = a[i \leftarrow x]\},
\end{aligned}
$$

where the notation $a[i \leftarrow x]$ denotes the result of storing x in the i-th position of the array a. These functions are directly mapped to their counterparts in Z3's array theory. This theory assumes that arrays are of unlimited length, so we have to attach a variable to each array denoting the length of the latter. When generating verification conditions, all calls to *len* are translated into accesses to this variable.

We transform imperative programs with mutable state into the IR by computing the control flow graph (CFG) of the input program. Each block in the CFG is transformed into single-static assignment SSA form. The resulting blocks are translated into a set of mutually recursive functions. The SSA transformation is applied locally to each block, so there is no need for ϕ functions in

```
define insert n v =
  letfun f1 n v     = let i = (−) n 1 in
                      let x = get v n in
                      f2 i x v
        f2 i x v    = let b = (≥) i 0 in
                      case b of true → f3 i x v
                                false → f5 i x v
        f3 i x v    = let z = get v i in
                      let b = (<) x z in
                      case b of true → f4 i x v
                                false → f5 i x v
        f4 i x v    = let i1 = (+) i 1 in
                      let z = get v i in
                      let v' = set v i1 z in
                      let i2 = (−) i 1 in
                      f2 i2 x v'
        f5 i x v    = let i1 = (+) i 1 in
                      let v1 = set v i1 x in v1
  in f1 n v
```

Fig. 5. IR representation of the *insert* algorithm.

each node of the CFG. The ϕ functions are emulated by parameter passing in the resulting set of mutually recursive functions. As a last step, we flatten **let** expressions to obtain an IR program in A-normal form (see Montenegro et al. [23] for details). Figure 5 shows the result of transforming the *insert* function of Figure 1 into the IR.

6 THE LIQUID TYPE SYSTEM

In this section, we define the rules of our type system. We will use B to denote *basic* liquid types, T for either basic types or dependent tuple types, and LT for the previous ones and functional liquid types:

$$\begin{array}{lll} B & ::= \{v : \tau \mid \varphi\} & \text{Basic types} \\ T & ::= B \mid \langle \overline{x_i :: B_i} \rangle & \text{Nonfunctional types} \\ LT & ::= T \mid T_1 \to T_2 & \text{Liquid types} \end{array}$$

Although in the program each function receives a sequence of arguments ($\overline{x_i :: B_i}$), we will use a dependent tuple type, $\langle \overline{x_i :: B_i} \rangle$ to express the dependency between them in its functional type. If the function returns several results, the tuple is explicitly built in the function body, and this type also encloses the dependency between such results $\langle \overline{y_j :: B'_j} \rangle$.

We define the α-equivalence relation between liquid types as usual. We write $LT \equiv_\alpha LT'$ to represent that LT and LT' are equal except for a possible renaming of the quantified variables in the refinements and of the variables used to label the tuple components. In particular, $\langle \overline{x_i :: \{v : \tau_i \mid \varphi_i\}} \rangle \equiv_\alpha \langle \overline{y_i :: \{v : \tau_i \mid \varphi'_i[\overline{y_i}/\overline{x_i}]\}} \rangle$, if $\varphi_i \equiv_\alpha \varphi'_i$ and $\overline{y_i}$ are fresh variables not occurring in $\overline{\varphi_i}'$.

Type schemes are obtained by quantifying type variables:

$$S ::= LT \mid \forall \alpha S.$$

The rules of the liquid type system can be classified into three groups: expression typing rules, a function definition typing rule, and subtyping rules. The left-hand side of the sequents of these rules is a typing environment Γ containing information about the liquid types of free variables, function, and type constructor symbols, as well as Boolean conditions denoted by φ. More precisely,

$$\begin{array}{l} \Gamma ::= \epsilon \mid \Gamma, \gamma \\ \gamma ::= \varphi \mid x :: B \mid f :: S \mid C :: S. \end{array}$$

So that that the types are well formed, we require that the refinements are logic formulas whose free variables are bound in the environment. Rules guaranteeing well formedness of types are similar to those in the original system [27], and we do not show them here. Notice that in type environments there are no variables with tuple types, because by syntactic construction the individual components of tuples are pattern matched in let bindings.

$$\frac{\Gamma \vdash e :: LT_1 \quad \Gamma \vdash LT_1 <: LT_2}{\Gamma \vdash e :: LT_2} \ (LTSub)$$

$$\frac{\tau = hmtype(\Gamma(x))}{\Gamma \vdash x :: \{v : \tau \mid v = x\}} \ (LTvar) \qquad \frac{\tau = hmtype(c)}{\Gamma \vdash c :: \{v : \tau \mid v = c\}} \ (LTconst)$$

$$\frac{\Gamma \vdash a :: B \quad \Gamma, x :: B \vdash \langle \overline{a_i} \rangle :: \left\langle \overline{x_i :: B_i} \right\rangle}{\Gamma \vdash \langle a, \overline{a_i} \rangle :: \left\langle x :: B, \overline{x_i :: B_i} \right\rangle} \ (LTtuple)$$

where x, $\overline{x_i}$ are fresh variables

$$\frac{f :: S \in \Gamma \quad Inst(\left\langle \overline{x_i :: B_i} \right\rangle \rightarrow T, S) \quad \Gamma \vdash \langle \overline{a_i} \rangle :: \left\langle \overline{x_i :: B_i} \right\rangle}{\Gamma \vdash f\ \overline{a_i} :: T[\overline{a_i}/\overline{x_i}]} \ (LTapp)$$

($LTctr$) for $C\ \overline{a_i} :: B$ is defined similarly as ($LTapp$)

$$\frac{\Gamma \vdash be :: \left\langle \overline{x_i' :: B_i} \right\rangle \quad \Gamma, \overline{x_i :: B_i[x_i/x_i']} \vdash e :: T}{\Gamma \vdash \textbf{let}\ \langle \overline{x_i :: \tau_i} \rangle = be\ \textbf{in}\ e :: T} \ (LTlet)$$

where $hmtype(B_i) = \tau_i$, and the $\overline{x_i}$ are not free in T

$$\frac{\forall i\ C_i :: S_i \in \Gamma \quad Inst(\left\langle \overline{x_{ij} :: B_{ij}} \right\rangle \rightarrow \{v : \tau \mid \varphi_i\}, S_i) \quad \Gamma, \overline{x_{ij} :: B_{ij}}, \varphi_i[x/v] \vdash e_i :: T}{\Gamma \vdash \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij} :: \tau_{ij}} \rightarrow e_i}\ :: T} \ (LTcase)$$

where the $\overline{\overline{x_{ij}}}$ are not free in T

$$\frac{\forall i \quad \Gamma, \overline{f_i :: \left\langle \overline{x_{ij} :: B_{ij}} \right\rangle \rightarrow \left\langle \overline{y_{ij} :: B'_{ij}} \right\rangle}, \overline{x_{ij} :: B_{ij}} \vdash e_i :: \left\langle \overline{y_{ij} :: B'_{ij}} \right\rangle \quad \Gamma, \overline{f_i :: S_i} \vdash e :: T}{\Gamma \vdash \textbf{letfun}\ \overline{f_i\ (\overline{x_{ij} :: \tau_{ij}}) :: (\overline{y_{ij} :: \tau'_{ij}}) = e_i}\ \textbf{in}\ e :: T} \ (LTfun)$$

where $S_i = gen(\left\langle \overline{x_{ij} :: B_{ij}} \right\rangle \rightarrow \left\langle \overline{y_{ij} :: B'_{ij}} \right\rangle, \Gamma), \quad \tau_{ij} = hmtype(B_{ij}), \quad \tau'_{ij} = hmtype(B'_{ij})$

Fig. 6. Typing rules for expressions.

6.1 Typing Rules for Expressions and Function Definitions

The typing rules for expressions are shown in Figure 6. The sequents of these rules have the form $\Gamma \vdash e :: T$. By $\tau = hmtype(B)$, we denote that the Hindley-Milner base type of B is τ. We also use $hmtype(c)$ to denote the Hindley-Milner type of a constant c. By $gen(LT, \Gamma)$, we denote the *generalization of LT with respect to* Γ, and it is defined by $gen(LT, \Gamma) = \forall \overline{\alpha}.LT$, where $\overline{\alpha}$ are the free fresh type variables of LT not occurring in Γ. The relation $Inst(LT, S)$, used in the premises of the ($LTapp$) and ($LTcase$) rules, denotes that the liquid type *LT is an instance of the scheme S*. It is defined as follows:

- $Inst(LT, LT')$ if and only if $LT \equiv_\alpha LT'$
- $Inst(LT, \forall \alpha.S)$ if and only if there is τ such that $Inst(LT, S[\tau/\alpha])$.

$$\frac{\Gamma,\ f :: \left\langle \overline{x_i :: B_i} \right\rangle \to \left\langle \overline{y_j :: B'_j} \right\rangle, \overline{x_i :: T_i} \vdash e :: \left\langle \overline{y_j :: B'_j} \right\rangle \quad \psi_1 \Rightarrow \bigwedge_i \varphi_i[x_i/\nu] \quad \bigwedge_j \varphi'_j[y_j/\nu] \Rightarrow \psi_2}{\Gamma \vdash \textbf{define}\ \{\psi_1\} f(\overline{x_i :: \tau_i}) :: (\overline{y_j :: \tau'_j}) = e\ \{\psi_2\} :: gen(\left\langle \overline{x_i :: B_i} \right\rangle \to \left\langle \overline{y_j :: B'_j} \right\rangle, \Gamma)}\ (LTdef)$$

where $B_i = \{\nu : \tau_i \mid \varphi_i\}, \quad B'_j = \{\nu : \tau'_j \mid \varphi'_j\}$

Fig. 7. Typing rule for a top-level function definition.

The system has a rule for each syntactically different expression, except the rule (*LTSub*), which can be applied anywhere in a type derivation.

The rules (*LTvar*) and (*LTconst*) give the strongest possible liquid typesrespectively to a free program variable and a literal constant. The rule (*LTtuple*) gives types to the tuple components from left to right, thus allowing the type of a component to depend on the value of a prior component. The same strategy is used to type the arguments of a function application in the rule (*LTapp*). Notice in this rule that the actual arguments are substituted for the formal ones in the result type. Fortunately, in our IR, an actual argument can only be a constant or a variable, keeping this substitution simple. In the (*LTlet*) rule, the bound variables $\overline{x_i}$ are out of scope after the **let** expression, so we forbid them to occur free in T. We admit, however, that the refinement of type T contains an existentially quantified formula (e.g., $\exists \overline{x_i}.\varphi$). Similarly, in the (*LTcase*) rule, T's refinement may contain an existentially quantified formula (e.g., $\exists \overline{\overline{x_{ij}}}.\varphi$). Notice in this latter rule that the predicate φ_i satisfied by the result of the constructor C_i in the i-th branch of the **case** is satisfied by the discriminant x in this branch, and that this knowledge is added to the typing environment to type the branch expression.

Finally, the (*LTfun*) rule is the only one that may introduce polymorphism and mutual recursion between functions. Several different instances of a polymorphic function definition may be used in the **letfun** main expression, hence the type generalization done before typing this expression.

The (*LTdef*) typing rule for a top-level function definition is very similar to that of **letfun**, and it is shown in Figure 7. The main difference is that we allow the function to contain *pre/post* conditions ψ_1 and ψ_2, and so the relation between them and the liquid types of the function arguments and results must be checked for validity.

6.2 Subtyping Rules

The sequents of these rules have the form $\Gamma \vdash LT <: LT'$, where $LT <: LT'$ represents that LT is a subtype of LT'. They appear in Figure 8, where $[\![\Gamma]\!]$ represents the semantics of the environment Γ and is defined recursively as follows:

- $[\![\ ]\!] = true$
- $[\![\Gamma, x :: \{\nu : \tau \mid \varphi\}]\!] = [\![\Gamma]\!] \wedge \varphi[x/\nu]$
- $[\![\Gamma, f :: S]\!] = [\![\Gamma]\!]$
- $[\![\Gamma, C :: S]\!] = [\![\Gamma]\!]$
- $[\![\Gamma, \varphi]\!] = [\![\Gamma]\!] \wedge \varphi$

To prove that a basic type is a subtype of another one, the rule (*STbase*) introduces as a proof obligation checking the validity of a logical formula. The rule (*STtuple*) extends the relation in the obvious way to tuples, and the rule (*STfun*) does the same for functional types. In this case, the relation is covariant in the result and contravariant in the arguments.

Trivially, since in the classic first-order logic the implication behaves as a reflexive and transitive relation we can prove, by induction on the depth of derivations, that subtype relation between

$$\frac{[\![\Gamma]\!] \wedge \varphi_1 \Rightarrow \varphi_2}{\Gamma \vdash \{\nu : \tau \mid \varphi_1\} <: \{\nu : \tau \mid \varphi_2\}} \ (STbase)$$

$$\frac{\Gamma \vdash B <: B' \quad \Gamma, x :: B \vdash \left\langle \overline{x_i :: B_i} \right\rangle <: \left\langle \overline{x_i' :: B_i'} \right\rangle}{\Gamma \vdash \left\langle x :: B, \overline{x_i :: B_i} \right\rangle <: \left\langle x' :: B', \overline{x_i' :: B_i'[x'/x]} \right\rangle} \ (STtuple)$$

where $x', \overline{x_i'}$ fresh variables not free in Γ

$$\frac{\Gamma \vdash \left\langle \overline{x_i' :: B_i'} \right\rangle <: \left\langle \overline{x_i :: B_i} \right\rangle \quad \Gamma, \overline{x_i' :: B_i'} \vdash T[\overline{x_i}'/\overline{x_i}] <: T'}{\Gamma \vdash \left\langle \overline{x_i :: B_i} \right\rangle \to T <: \left\langle \overline{x_i' :: B_i'} \right\rangle \to T'} \ (STfun)$$

Fig. 8. Subtype system.

types is reflexive and transitive. The following lemmas state that the liquid type and subtype systems are closed under α-equivalence.

Lemma 1. *Let Γ be a liquid type environment. Let LT_1, LT_2, LT_1', LT_2' liquid types such that $LT_1 \equiv_\alpha LT_1'$ and $LT_2 \equiv_\alpha LT_2'$. If $\Gamma \vdash LT_1 <: LT_2$, then $\Gamma \vdash LT_1' <: LT_2'$.*

Lemma 2. *Let Γ be a liquid type environment. Let T_1, T_1' be nonfunctional liquid types such that $T_1 \equiv_\alpha T_1'$, and let e be an expression. If $\Gamma \vdash e :: T_1$, then $\Gamma \vdash e :: T_1'$ has a derivation of the same depth.*

As a consequence of Lemma 2, we can simplify the (*LTlet*) rule as follows:

$$\frac{\Gamma \vdash be :: \left\langle \overline{x_i :: B_i} \right\rangle \quad \Gamma, \overline{x_i :: B_i} \vdash e :: T}{\Gamma \vdash \textbf{let } \langle \overline{x_i :: \tau_i} \rangle = be \textbf{ in } e :: T} \ (LTlet).$$

Similarly, renamings in the subtyping rules (*STtuple*) and (*STfun*) can be removed.

The following lemma asserts that strengthening the environment allows to prove the same typing and subtyping relations. It will be useful later (see Proposition 4) to prove completeness of our VCG algorithm.

Lemma 3. *Let Γ and Γ' be liquid type environment such that $[\![\Gamma']\!] \Rightarrow [\![\Gamma]\!]$. It holds that*

- *If LT_1, LT_2 are liquid types such that $\Gamma \vdash LT_1 <: LT_2$, then $\Gamma' \vdash LT_1 <: LT_2$.*
- *If T is a liquid type and e an expression such that $\Gamma \vdash e :: T$, then $\Gamma' \vdash e :: T$ has a derivation of the same depth.*

6.3 Type Operators

In the algorithm described later in Section 8.2 we will use the following operators defined on types and also some of their properties, which we show in the following.

The disjunction of simple liquid types, $\bigvee_i T_i$, is recursively defined as follows:

- $\bigvee_i \{\nu : \tau \mid \varphi_i\} = \{\nu : \tau \mid \bigvee_i \varphi_i\}$
- $\bigvee_i \langle \overline{x_{ij} :: B_{ij}} \rangle = \langle \overline{x_j :: \bigvee_i B_{ij}} \rangle$, assuming that $x_{ij} = x_j$ for every i.

The conjunction, $T \wedge \psi$, is recursively defined as follows:

- $\{\nu : \tau \mid \varphi\} \wedge \psi = \{\nu : \tau \mid \varphi \wedge \psi\}$
- $\langle \overline{x_i :: B_i} \rangle \wedge \psi = \langle \overline{x_i :: B_i \wedge \psi} \rangle$, assuming that $\overline{x_i}$ are not free in ψ.

The existential quantification, $\exists x.T$, represents a liquid type whose refinement will contain existentially quantified formulas, and it is defined as follows:

- $\exists x.\{v : \tau \mid \varphi\} = \{v : \tau \mid \exists x.\varphi\}$
- $\exists x.\langle \overline{x_i :: \{v : \tau_i \mid \varphi_i\}}\rangle = \langle \overline{x_i :: \{v : \tau_i \mid \exists x.(\varphi_1[x_1/v] \wedge \ldots \wedge \varphi_i)\}}\rangle$ assuming that $x \notin \overline{x_i}$.

The reason for this complexity is that the quantified variable x should be instantiated in such a way that it satisfies at the same time the current predicate φ_i and all predicates to its left in the dependent tuple.

Example 4. We show an example of existential quantification over a tuple type:

$$\exists x.\, \langle y_1 :: \{v : int \mid v > x \wedge x > 0\}, y_2 :: \{v : int \mid v > x \wedge v < y_1\}\rangle$$
$$= \langle y_1 :: \{v : int \mid \exists x.(v > x \wedge x > 0)\}, y_2 :: \{v : int \mid \exists x.(y_1 > x \wedge x > 0 \wedge v > x \wedge v < y_1)\}\rangle\,.$$

Now we show some properties that will be useful to prove soundness and completeness of the VCG algorithm (see Sections 8.3 and 8.4). Some of their proofs can be found in the electronic appendix.

Lemma 4. *Let Γ be a type environment and T a nonfunctional liquid type. It holds that $\Gamma \vdash T <: \exists x.T$.*

Lemma 5. *Let Γ be an environment and $\overline{T_i}$ nonfunctional liquid types. It holds that $\Gamma \vdash T_i <: \bigvee_i T_i$ for every i.*

Lemma 6. *Let Γ be a type environment and T, T' nonfunctional liquid types. If x is not free in T' and $\Gamma \vdash T <: T'$, then $\Gamma \vdash \exists x.T <: T'$.*

Lemma 7. *Let Γ be a type environment and $\overline{T_i}$ nonfunctional liquid types. If $\Gamma \vdash \exists x.T_i <: T$ for every i, then $\Gamma \vdash \bigvee_i \exists x.T_i <: T$.*

Proof. If T and $\overline{T_i}$ are basic, the proof holds because $\Gamma \vdash \exists x.T_i <: T$ for every i means that for certain formulas $\overline{\varphi_i}$ and φ, it holds $[\![\Gamma]\!] \wedge \exists x.\varphi_i \Rightarrow \varphi$ for every i. Then $[\![\Gamma]\!] \wedge \bigvee_i \exists x.\varphi_i \Rightarrow \varphi$ holds as wanted. If $T, \overline{T_i}$ are tuples, we can assume that they are of the form $T_i = \langle \overline{x_j :: B_{ij}}\rangle$, $B_{ij} = \{v : \tau_j \mid \varphi_{ij}\}$, $T = \langle \overline{x_j :: \{v : \tau_j \mid \psi_j\}}\rangle$. Notice that we have types whose refinements contain existentially quantified formulas and that for every j, $\exists x.(\varphi_{i1}[x_{i1}/v] \wedge \ldots \wedge \varphi_{ij}) \Rightarrow \exists x.\varphi_{i1}[x_{i1}/v] \wedge \ldots \wedge \exists x.(\varphi_{i1}[x_{i1}/v] \wedge \ldots \wedge \varphi_{ij})$. Using this fact and from the hypothesis that it holds $\Gamma \vdash \exists x.T_i <: T$ for every i, we have that $[\![\Gamma]\!] \wedge \exists x.(\varphi_{i1}[x_{i1}/v] \wedge \ldots \wedge \varphi_{ij}) \Rightarrow \psi_j$ for every i and every j. Hence, $[\![\Gamma]\!] \wedge \bigvee_i \exists x.(\varphi_{i1}[x_{i1}/v] \wedge \ldots \wedge \varphi_{ij}) \Rightarrow \psi_j$ for every j. So, obviously it holds

$$[\![\Gamma]\!] \wedge \bigvee_i \exists x.\varphi_{i1}[x_{i1}/v] \wedge \ldots \wedge \bigvee_i \exists x.(\varphi_{i1}[x_{i1}/v] \wedge \ldots \wedge \varphi_{ij}) \Rightarrow \psi_j,$$

which allows us to conclude $\Gamma \vdash \bigvee_i \exists x.\langle \overline{x_j :: B_{ij}}\rangle <: T$, using rules (*STtuple*) and (*STbase*). □

Lemma 8. *Let Γ be a type environment and T a nonfunctional liquid type. If $[\![\Gamma]\!] \Rightarrow \varphi$, then $\Gamma \vdash T <: T \wedge \varphi$.*

Lemma 9. *Let Γ be an environment, $B = \{v : \tau \mid \varphi\}$ a basic liquid type, and T, T' nonfunctional liquid types. If $\Gamma, x : B \vdash T <: T'$, then $\Gamma \vdash T \wedge \varphi[x/v] <: T'$.*

7 THE TYPE INFERENCE ALGORITHM

The liquid type inference algorithm has the following phases:

(1) A standard type checking algorithm is run on the program, and we assume that it type checks. Then every variable is annotated with τ, its conventional HM type. Remember that 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 fresh variable ι of the appropriate kind, representing a predicate, so obtaining type annotations of the form $x :: \{\nu : \tau \mid \iota\}$. These variables are called *template variables*.[4] Refined types containing such variables are called *template types* (see Section 8.1). An initial template environment containing those template types, as well as liquid types for predefined function and type constructor symbols, is built to be used in the next phases.
(3) The syntax-driven liquid typing rules presented in Section 6 are applied. Starting from the initial template environment and the main expression to be verified, a *set* Φ *of constraints* is obtained. This is done by our VCG algorithm, presented in Section 8. These constraints come in the end from the premise of the rule (*STbase*) of Figure 8 and are to be satisfied so that the program can be correctly typed in the liquid type sense. A basic constraint has the form $[\![\Gamma]\!] \wedge \varphi_1 \Rightarrow \varphi_2$, where the φ_1 and φ_2 are template variables with *pending substitutions*, such as in Rondon et al. [27]. The purpose of these pending substitutions is to replace formal arguments by actual ones in function applications (see the rule (*LTapp*) of Figure 6). $[\![\Gamma]\!]$ may contain additional template variables.

    The aim of the inference algorithm is to solve the set of constraints Φ obtained by the VCG algorithm. That means to find an appropriate substitution A that maps all template variables in Φ to predicates, in such a way that A satisfies Φ.
(4) The constraints are solved by an *iterative weakening* algorithm. Roughly speaking, the algorithm starts with the *strongest possible mapping* A for all template variables, and at each step a variable assignment is weakened to satisfy a constraint. If a fixpoint is reached, then the final mapping obtained, when applied to all templates, gives us the liquid types for all of the variables. In this phase, it is important to note that the range of considered substitutions A is restricted, for each template variable, to conjunctions of some predefined qualifiers (see Section 9.1). By the strongest possible mapping, we mean the substitution that, satisfying the preceding conditions, converts the set of constraints to the strongest possible one.

As it has been said, in Section 8.5 we present the VCG algorithm. There, we do not restrict the form the predicates may have. Its purpose is to obtain a set of constraints as strong as possible so that, should these constraints be solved, then we would obtain the strongest possible liquid type for each type annotation. We prove that this algorithm is both sound and complete with respect to the type system of Section 6.

However, the whole purpose of the liquid type inference is to have a decision procedure for typing a restricted class of programs dealing with arrays. In this sense, we do not want arbitrarily complex predicates in our types, because checking the validity of the constraints generated by *VCG* will be delegated to an SMT solver supporting a set of decidable, but also restricted, theories.

For that reason, in Section 8.5 we restrict the form of the generated constraints so that the ones obtained by the actual algorithm, let us call it *VCG′*, are less involved than the ones generated

[4]Note that there will be as many template variables as type annotations.

by *VCG*. In particular, we get rid of disjunctions of types. The types obtained are still sound, but completeness is lost—that is, some valid types will never be obtained by the algorithm. This is not a very important issue because, as it has been said, the iterative weakening algorithm restricts even more the form of the predicates allowed in the type refinements. In particular, predicates must be conjunctions of some atomic qualifiers taken from a given finite set, refinements containing existentially quantified formulas are not allowed, and a restricted form of universal quantification is allowed only in the refinements of array types. All of these restrictions are imposed to have a terminating algorithm. Nevertheless, some form of completeness is still achieved in the following sense: if the program admits a typing with liquid types having the syntactic restrictions imposed to the predicates, then our algorithm terminates and it finds the strongest possible types having this shape.

With respect to the original liquid type framework [27], the main differences of the approach presented here are the following:

- In addition to the κ template variables, we introduce four new kinds of template variables in our template types—all of them related to the array type.
- Our first VCG algorithm is complete with respect to the type system given in Section 6. The second one, VCG', deals with the types whose refinements contain existentially quantified formulas, introduced by **let** and **case** expressions, by transforming the existential variables to universally quantified ones (see Section 8.5).
- The iterative weakening algorithm is much more complex than the conventional one, due to the way in which the new template variables are initialised and weakened. An additional complication is that although weakening a template variable, new template variables may be created on the fly.

8 THE VCG ALGORITHM

Here we describe our algorithm, called *VCG*, which generates constraints over template variables by mainly following the syntax-driven liquid typing rules given in Section 6. The algorithm *VCG* takes as input a template environment Γ and an expression e whose type is to be inferred, and returns as output a pair $(\hat{T}, \Phi)$, where $\hat{T}$ is the inferred template type corresponding to e, and Φ is a set of constraints that must be satisfied to type check e. The originality of the method with respect to that of Rondon et al. [27] lies basically in the explicit use of refinements containing existentially quantified formulas to capture the type of the **let** and **case** expressions, to force bound variables not to be free in the resulting type $\hat{T}$, as well as the use of disjunctions to model the different branches of a **case**. The reason for this is to infer the most precise type for each expression in such a way that completeness is preserved.

First, we will specify what template types and environments are. After introducing the algorithm, we will show that if $VCG(\Gamma, e) = (\hat{T}, \Phi)$, there is a map A from template variables to predicates that satisfies Φ, if and only if e has a valid liquid type derivation of $A(\hat{T})$ from $A(\Gamma)$.

8.1 Template Types and Environments

A basic *template type* is a refined type $\{\nu : \tau \mid \varphi\}$, where the refinement predicate φ may contain template variables and pending substitutions θ. Template variables are to be replaced by actual predicates during the constraint solving phase. We will use the notation κ for template variables refining the type of a variable that is not an array and μ for template variables refining an array type (see Section 9.1).

Tuple and functional template types are constructed in the obvious way from basic template types. Basic template types are denoted by $\hat{B}$, nonfunctional template types by $\hat{T}$, and general

template types by $\widehat{LT}$. A *template scheme* is defined trivially from a template type and is denoted by $\hat{S}$.

We will make use of the following notation. Let A be a mapping from template variables to predicates, and let φ be a formula containing template variables and pending substitutions. By $A(\varphi)$, we denote the instance of φ that result by replacing in it every template variable ι by $A(\iota)$ and then apply the pending substitutions of φ. $A(\{\nu : \tau \mid \varphi\})$ represents the liquid type $\{\nu : \tau \mid A(\varphi)\}$. $A(\widehat{LT})$ denotes the natural extension of the notation $A(\hat{B})$ to the general case. If Φ is a set of constraints, we say that A satisfies Φ if $A(\varphi)$ is *true* for every $\varphi \in \Phi$.

A *template environment* Γ is a type environment which may contain template types and schemes, as well as ordinary liquid types and guards including template variables. If Γ is a template environment and A a mapping from template variables to predicates, $A(\Gamma)$ represents $\{A(\gamma) \mid \gamma \in \Gamma\}$.

By $\overline{\Gamma}$, we denote the *shape* of Γ—that is, the result of replacing in Γ all refinement predicates by *true*, and also eliminating the guards.

8.2 The Algorithm

The algorithm *VCG* assumes the existence of an algorithm *THM* that infers the Hindley-Milner type of a function definition $f(\overline{x_i :: \tau_i}) :: (\overline{y_j :: \tau'_j}) = e$ from a template environment shape $\overline{\Gamma}$, and returns a template type using the inferred types to generate templates, by introducing appropriate fresh template variables to represent the unknown refinements corresponding to the subexpressions in the inferred Hindley-Milner type:

$$THM(\overline{\Gamma}, f(\overline{x_i :: \tau_i}) :: (\overline{y_j :: \tau'_j}) = e) = \langle \overline{x_i :: \{\nu : \tau_i \mid \iota_i\}} \rangle \to \langle \overline{y_j :: \{\nu : \tau'_j \mid \iota'_j\}} \rangle$$

where $\overline{\iota_i}$ and $\overline{\iota_j}'$ are fresh template variables with respect to $\overline{\Gamma}$.

Two properties hold:

- If $THM(\overline{\Gamma}, f(\overline{x_i :: \tau_i}) :: (\overline{y_j :: \tau'_j}) = e) = \langle \overline{x_i :: \hat{B}_i} \rangle \to \langle \overline{y_j :: \hat{B}'_j} \rangle$, then $hmtype(\hat{B}_i) = \tau_i$ and $hmtype(\hat{B}'_j) = \tau'_j$, trivially by definition.
- Given a liquid type $LT = \langle \overline{x_i :: \{\nu : \tau_i \mid \varphi_i\}} \rangle \to \langle \overline{y_j :: \{\nu : \tau'_j \mid \varphi'_j\}} \rangle$ and its template $THM(\overline{\Gamma}, f(\overline{x_i :: \tau_i}) :: (\overline{y_j :: \tau'_j}) = e) = \langle \overline{x_i :: \hat{B}_i} \rangle \to \langle \overline{y_j :: \hat{B}'_j} \rangle$, then there exists a substitution A such that $A(\langle \overline{x_i :: \hat{B}_i} \rangle \to \langle \overline{y_j :: \hat{B}'_j} \rangle) = LT$. Trivially, any A such that $A(\iota_i) = \varphi_i$ and $A(\iota'_j) = \varphi'_j$ meets the property.

Now we show the formal equations defining the algorithm *VCG*:

- $VCG(\Gamma, x) = (\{\nu : hmtype(\Gamma(x)) \mid \nu = x\}, \emptyset)$.
- $VCG(\Gamma, c) = (\{\nu : hmtype(c) \mid \nu = c\}, \emptyset)$.
- $VCG(\Gamma, \langle a, \overline{a_i} \rangle) =$
  $\texttt{let}(\hat{B}, \Phi_1) = VCG(\Gamma, a)\ \texttt{in}$
  $\texttt{let}(\langle \overline{x_i :: \hat{B}_i} \rangle, \Phi_2) = VCG(\Gamma; x :: \hat{B}, \langle \overline{a_i} \rangle)\ \texttt{in}$
  $(\langle x :: \hat{B}, \overline{x_i :: \hat{B}_i} \rangle, \Phi_1 \cup \Phi_2)$.
- $VCG(\Gamma, f\ \overline{a_i}) =$
  $\texttt{let}\ S = \Gamma(f)\ \texttt{in}$
  $\texttt{let}(\overline{\langle x_i :: \hat{B}_i \rangle}, \Phi_1) = VCG(\Gamma, \langle \overline{a_i} \rangle)\ \texttt{in}$
  $\texttt{let} \overline{\langle x_i :: \hat{B}'_i \rangle} \to \hat{T} = get\_inst(S, \overline{(x_i :: hmtype(\hat{B}_i))})\texttt{in}$

$\texttt{let}\ \Phi_2 = subt(\Gamma, \langle \overline{x_i :: \hat{B}_i} \rangle, \langle \overline{x_i :: \hat{B}'_i} \rangle)\texttt{in}$
$(\hat{T}[\overline{a_i/x_i}], \Phi_1 \cup \Phi_2)$.

- $VCG(\Gamma, C\ \overline{a_i})$ is defined analogously to the $f\ \overline{a_i}$ case.
- $VCG(\Gamma, \textbf{let}\ \langle \overline{x_i :: \tau_i} \rangle = be\ \textbf{in}\ e) =$
  $\texttt{let}(\langle \overline{x'_i :: \hat{B}_i} \rangle, \Phi_1) = VCG(\Gamma, be)\ \texttt{in}$
  $\texttt{let}(\hat{T}, \Phi_2) = VCG(\Gamma; \overline{x_i :: \hat{B}_i[\overline{x_i/x_i}']}, e)\ \texttt{in}$
  $(\exists \overline{x_i}.(\hat{T} \wedge \bigwedge_i \varphi_i[\overline{x_i/x_i}'][x_i/\nu]), \Phi_1 \cup \Phi_2)$
  where $\hat{B}_i = \{\nu : \tau_i \mid \varphi_i\}$.
- $VCG(\Gamma, \textbf{letfun}\ \overline{f_i(\overline{x_{ij} :: \tau_{ij}}) :: (\overline{y_{ij} :: \tau'_{ij}}) = e_i}\ \textbf{in}\ e) =$
  $\texttt{let}\overline{\langle \overline{x_{ij} :: \hat{B}_{ij}} \rangle \rightarrow \langle \overline{y_{ij} :: \hat{B}'_{ij}} \rangle} = THM(\overline{\Gamma}, \overline{f_i(\overline{x_{ij} :: \tau_{ij}}) :: (\overline{y_{ij} :: \tau'_{ij}}) = e_i})\ \texttt{in}$
  $\texttt{let}\overline{(\hat{T}_{e_i}, \Phi^1_i) = VCG(\Gamma_i, e_i)}\texttt{in}$
  $\texttt{let}\overline{\Phi^2_i = subt(\Gamma_i, \hat{T}_{e_i}, \langle \overline{y_{ij} :: \hat{B}'_{ij}} \rangle)}\ \texttt{in}$
  $\texttt{let}\overline{\hat{S}_i = gen(\langle \overline{x_{ij} :: \hat{B}_{ij}} \rangle \rightarrow \langle \overline{y_{ij} :: \hat{B}'_{ij}} \rangle, \Gamma)}\ \texttt{in}$
  $\texttt{let}(\hat{T}, \Phi) = VCG(\Gamma; \overline{f_i :: \hat{S}_i}, e)\ \texttt{in}$
  $(\hat{T}, \Phi \cup \bigcup_i (\Phi^1_i \cup \Phi^2_i))$
  where $\Gamma_i = \Gamma; \overline{f_i :: \langle \overline{x_{ij} :: \hat{B}_{ij}} \rangle \rightarrow \langle \overline{y_{ij} :: \hat{B}'_{ij}} \rangle}; \overline{x_{ij} :: \hat{B}_{ij}}$.
- $VCG(\Gamma, \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij} :: \tau_{ij}} \rightarrow e_i}) =$
  $\texttt{let}\tau = hmtype(\Gamma(x))\ \texttt{in}$
  $\texttt{let}\ \overline{\hat{S}_i = \Gamma(C_i)}\texttt{in}$
  $\texttt{let}\ \overline{\langle \overline{x_{ij} :: \hat{B}_{ij}} \rangle \rightarrow \{\nu : \tau \mid \varphi_i\} = get\_inst(\hat{S}_i, (\overline{x_{ij} :: \tau_{ij}}))}\texttt{in}$
  $\texttt{let}\overline{(\hat{T}_i, \Phi_i) = VCG(\Gamma; \overline{x_{ij} :: \hat{B}_{ij}}; \varphi_i[x/\nu], e_i)}\texttt{in}$
  $(\bigvee_i \exists \overline{x_{ij}}.(\hat{T}_i \wedge \bigwedge_j \varphi_{ij}[x_{ij}/\nu] \wedge \varphi_i[x/\nu]), \bigcup_i \Phi_i)$
  where $\hat{B}_{ij} = \{\nu : \tau_{ij} \mid \varphi_{ij}\}$,

In the definition of *VCG* for **case** and function application cases, *get_inst* is used to obtain an instance of a template scheme. Let $\hat{S}$ be a template scheme and τ_i be ground types. The function $get\_inst(\hat{S}, (\overline{x_i :: \tau_i}))$ returns the instance of $\hat{S}$ that makes the arguments of $\hat{S}$ match $(\overline{x_i :: \tau_i})$, if it is possible:

- $get\_inst(\forall \overline{\alpha}.\overline{\langle y_i :: \hat{B}_i \rangle} \rightarrow \hat{T}, (\overline{x_i :: \tau_i})) =$
  $\texttt{let}\ \theta = match(\overline{\alpha}, \overline{hmtype(\hat{B}_i)}, \overline{\tau_i})\texttt{in}\ \overline{\langle x_i :: \hat{B}_i[\overline{x_i/y_i}] \rangle}\theta \rightarrow \hat{T}[\overline{x_i/y_i}]\theta$,
  where $match(\overline{\alpha}, \overline{\tau_i}, \overline{\tau'_i})$ is the unique substitution $[\overline{\tau}/\overline{\alpha}]$, that makes $\overline{\tau_i[\overline{\tau}/\overline{\alpha}]} = \overline{\tau_i}'$, if any.
- $get\_inst(\hat{S}, (\overline{x_i :: \tau'_i}))$ is undefined when the types of the arguments do not match, or $\hat{S}$ does not have the expected form of function type.

We consider $\overline{\alpha}$ empty as a particular case, in which $\overline{\tau}$ and $\overline{\tau}'$ must be the same.

The algorithm *subt* which generates verification constraints from subtyping conditions is a direct translation of the subtyping rules defined in Section 6.2. In particular, $subt(\Gamma, \widehat{LT}_1, \widehat{LT}_2)$, returns a set of constraints Φ that must be satisfied so that $\widehat{LT}_1$ is a subtype of $\widehat{LT}_2$:

- $subt(\Gamma, \{\nu : \tau \mid \varphi_1\}, \{\nu : \tau \mid \varphi_2\}) = \{[\![\Gamma]\!] \wedge \varphi_1 \Rightarrow \varphi_2\}$.
- $subt(\Gamma, \langle x :: \hat{B}, \overline{x_i :: \hat{B}_i}\rangle, \langle x' :: \hat{B}', \overline{x'_i :: \hat{B}'_i}\rangle) =$
  $subt(\Gamma, \hat{B}, \hat{B}') \cup subt(\Gamma; x :: \hat{B}, \langle x_i :: \hat{B}_i\rangle, \langle \overline{x'_i :: \hat{B}'_i[x/x']}\rangle)$.
- $subt(\Gamma, \langle \overline{x_i :: \hat{B}_i}\rangle \to \hat{T}, \langle \overline{x'_i :: \hat{B}'_i}\rangle \to \hat{T}') =$
  $subt(\Gamma, \langle \overline{x'_i :: \hat{B}'_i}\rangle, \langle \overline{x_i :: \hat{B}_i}\rangle) \cup subt(\Gamma; \overline{x'_i :: \hat{B}'_i}, \hat{T}[\overline{x_i}'/\overline{x_i}], \hat{T}')$.

The algorithm *VCGD* obtains the template scheme and the verification conditions for a function definition, in the context of a template environment:

- $VCGD\ (\Gamma, \textbf{define}\ \{\psi_1\} f\ (\overline{x_i :: \tau_i}) :: (\overline{y_j :: \tau'_j}) = e\ \{\psi_2\}) =$
  $\texttt{let}\langle \overline{x_i :: \hat{B}_i}\rangle \to \langle \overline{y_j :: \hat{B}'_j}\rangle = THM(\overline{\Gamma}, f\ (\overline{x_i :: \tau_i}) :: (\overline{y_j :: \tau'_j}) = e)\ \texttt{in}$
  $\texttt{let}(\hat{T}, \Phi_1) = VCG(\Gamma', e)\ \texttt{in}$
  $\texttt{let}\Phi_2 = subt(\Gamma', \hat{T}, \langle \overline{y_j :: \hat{B}'_j}\rangle)\ \texttt{in}$
  $(gen(\langle \overline{x_i :: \hat{B}_i}\rangle \to \langle \overline{y_j :: \hat{B}'_j}\rangle, \Gamma), \Phi_1 \cup \Phi_2 \cup \{\psi_1 \Rightarrow \bigwedge_i \varphi_i[x_i/\nu], \bigwedge_j \varphi'_j[y_j/\nu] \Rightarrow \psi_2\}),$
  where $\hat{B}_i = \{\nu : \tau_i \mid \varphi_i\}$, $\hat{B}'_j = \{\nu : \tau'_j \mid \varphi'_j\}$, and $\Gamma' = \Gamma; f :: \langle \overline{x_i :: \hat{B}_i}\rangle \to \langle \overline{y_j :: \hat{B}'_j}\rangle; \overline{x_i :: \hat{B}_i}$.

The following lemmas state that the algorithms *subt* and *VCG* are closed under α-equivalence, as it happens for the liquid type and subtype systems.

Lemma 10. *Let Γ be a template environment. Let $\widehat{LT}_1, \widehat{LT}_2, \widehat{LT}'_1, \widehat{LT}'_2$ be template types such that $\widehat{LT}_1 \equiv_\alpha \widehat{LT}'_1$ and $\widehat{LT}_2 \equiv_\alpha \widehat{LT}'_2$. If $subt(\Gamma, \widehat{LT}_1, \widehat{LT}_2) = \Phi$, then $subt(\Gamma, \widehat{LT}'_1, \widehat{LT}'_2) = \Phi'$, where $\Phi \equiv_\alpha \Phi'$.*

Lemma 11. *Let Γ be a template environment. Let $\hat{T}_1, \hat{T}'_1$ be nonfunctional template types such that $\hat{T}_1 \equiv_\alpha \hat{T}'_1$, and let e be an expression. If $VCG(\Gamma, e) = (\hat{T}_1, \Phi)$, then $VCG(\Gamma, e) = (\hat{T}'_1, \Phi')$, where $\Phi \equiv_\alpha \Phi'$.*

As a consequence of these lemmas, we can choose $\overline{x_i}' = \overline{x_i}$ and take off the substitution $[\overline{x_i}/\overline{x_i}']$ in the rule of *VCG* for the expression **let**, as well as in the rules of *subt* for tuple and functional types.

8.3 Soundness of the VCG Algorithm

Now we show that the algorithm *VCGD* is sound with respect to the liquid type system. The proof is based on the soundness of the algorithms *VCG* and *subt* that we prove next.

To simplify the reading, we introduce the following notation. Let Γ be a template environment, e an IR expression, $\hat{T}$ a template type, and A a mapping from template variables to predicates. By $\Gamma \vdash_A e :: \hat{T}$, we denote $A(\Gamma) \vdash e :: A(\hat{T})$. Analogously, if $\widehat{LT}_1$ and $\widehat{LT}_2$ are template types, by $\Gamma \vdash_A \widehat{LT}_1 <: \widehat{LT}_2$ we denote $A(\Gamma) \vdash A(\widehat{LT}_1) <: A(\widehat{LT}_2)$.

Lemma 12. *Let $\hat{S}$ be a template type scheme, A be a mapping from at least all the template variables in $\hat{S}$ to predicates. Then $Inst(A(\overline{\langle x_i :: \hat{B}_i\rangle \to \hat{T}}), A(\hat{S}))$ if and only if $get\_inst(\hat{S}, (\overline{x_i :: \tau_i})) = \overline{\langle x_i :: \hat{B}_i\rangle \to \hat{T}}$.*

Proposition 1. *Let Γ be a template environment, $\widehat{LT}_1, \widehat{LT}_2$ be two template types, and Φ be a set of constraints such that $subt(\Gamma, \widehat{LT}_1, \widehat{LT}_2) = \Phi$. Then for every mapping A satisfying Φ, it holds that $\Gamma \vdash_A \widehat{LT}_1 <: \widehat{LT}_2$ has a derivation in the subtype system.*

Proof. By induction on the structure of $\widehat{LT}_1$ (see the electronic appendix). □

Proposition 2. *Let Γ be a template environment, e an expression, $\hat{T}$ a template type, and Φ a set of constraints such that $VCG(\Gamma, e) = (\hat{T}, \Phi)$. Then for every mapping A satisfying Φ, it holds that $\Gamma \vdash_A e :: \hat{T}$ has a valid derivation in the liquid type system.*

Proof. By induction on the structure of e. We show some cases here, and the others are detailed in the electronic appendix:

- Let $e = f\ \overline{a_i}$. If $VCG(\Gamma, e) = (\hat{T}, \Phi)$, then $\hat{T} = \hat{T}'[\overline{a_i}/\overline{x_i}]$, $\Phi = \Phi_1 \cup \Phi_2$, such that

  (i) $(\overline{\langle x_i :: \hat{B}'_i \rangle}, \Phi_1) = VCG(\Gamma, \overline{\langle a_i \rangle})$,
  (ii) $\Phi_2 = subt(\Gamma, \overline{\langle x_i :: \hat{B}'_i \rangle}, \overline{\langle x_i :: \hat{B}_i \rangle})$,
  (iii) $\overline{\langle x_i :: \hat{B}_i \rangle} \to \hat{T}' = get\_inst(\hat{S}, \overline{(x_i :: hmtype(\hat{B}'_i))})$, where $\hat{S} = \Gamma(f)$.

  As before, A satisfies Φ_1 and Φ_2. Applying the induction hypothesis to (i), a proof of the sequent $\Gamma \vdash_A \overline{\langle a_i \rangle} :: \overline{\langle x_i :: \hat{B}'_i \rangle}$ can be derived. In addition, we have $\Gamma \vdash_A \overline{\langle x_i :: \hat{B}'_i \rangle} <: \overline{\langle x_i :: \hat{B}_i \rangle}$, by Proposition 1 applied to (ii). Moreover by applying Lemma 12 to (iii), we obtain $Inst(A(\overline{\langle x_i :: \hat{B}_i \rangle} \to \hat{T}'), A(\hat{S}))$, with $\hat{S} = \Gamma(f)$, so $A(\hat{S}) = A(\Gamma)(f)$. Therefore, the following is a proof of $\Gamma \vdash_A e :: \hat{T}$.

$$\dfrac{\dfrac{\Gamma \vdash_A \overline{\langle a_i \rangle} :: \overline{\langle x_i :: \hat{B}'_i \rangle} \quad \Gamma \vdash_A \overline{\langle x_i :: \hat{B}'_i \rangle} <: \overline{\langle x_i :: \hat{B}_i \rangle}}{\Gamma \vdash_A \overline{\langle a_i \rangle} :: \overline{\langle x_i :: \hat{B}_i \rangle}}\ (LTSub) \qquad Inst(A(\overline{\langle x_i :: \hat{B}_i \rangle} \to \hat{T}'), A(\hat{S}))}{\Gamma \vdash_A f\ \overline{a_i} :: \hat{T}'[\overline{a_i}/\overline{x_i}]}\ (LTapp)$$

- Let $e = \textbf{let}\ \langle x_i :: \tau_i \rangle = be\ \textbf{in}\ e'$. If $VCG(\Gamma, e) = (\exists \overline{x_i}.\hat{T}, \Phi)$, then $\hat{T} = \hat{T}' \wedge \bigwedge_i \varphi_i[x_i/v])$, $\Phi = \Phi_1 \cup \Phi_2$ such that

  (i) $(\overline{\langle x_i :: \{v : \tau_i \mid \varphi_i\} \rangle}, \Phi_1) = VCG(\Gamma, be)$,
  (ii) $(\hat{T}', \Phi_2) = VCG(\Gamma; \overline{x_i :: \{v : \tau_i \mid \varphi_i\}}, e')$.

  Applying the induction hypothesis to (i) and to (ii) (A satisfies Φ_1 and Φ_2), we deduce the following:

  (iii) $\Gamma \vdash_A be :: \overline{\langle x_i :: \{v : \tau_i \mid \varphi_i\} \rangle}$
  (iv) $\Gamma, \overline{x_i :: \{v : \tau_i \mid \varphi_i\}} \vdash_A e' :: \hat{T}'$

  Notice that $\hat{T}$ and $\hat{T}'$ differ only on $\bigwedge_i \varphi_i[x_i/v]$, and $[\![\Gamma, \overline{x_i :: \{v : \tau_i \mid \varphi_i\}}]\!] \Rightarrow \bigwedge_i \varphi_i[x_i/v]$, so by Lemma 8 we obtain that $\Gamma, \overline{x_i :: \{v : \tau_i \mid \varphi_i\}} \vdash_A \hat{T}' <: \hat{T}$. Then by Lemma 4, we have that $\Gamma, \overline{x_i :: \{v : \tau_i \mid \varphi_i\}} \vdash_A \hat{T} <: \exists \overline{x_i}.\hat{T}$. Finally, by transitivity we get the following:

  (v) $\Gamma, \overline{x_i :: \{v : \tau_i \mid \varphi_i\}} \vdash_A \hat{T}' <: \exists \overline{x_i}.\hat{T}$.

  From (iv) and (v), we can derive the sequent $\Gamma, \overline{x_i :: \{v : \tau_i \mid \varphi_i\}} \vdash_A e' : \exists \overline{x_i}.\hat{T}$, using ($LTsubt$). Taking this sequent and (iii), as premises we obtain a derivation of

  $$\Gamma \vdash_A \textbf{let}\ \overline{\langle x_i :: \tau_i \rangle} = be\ \textbf{in}\ e' :: \exists \overline{x_i}.\hat{T}$$

  using the rule ($LTlet$), because $\overline{x_i}$ are not free in $\exists \overline{x_i}.\hat{T}$. □

Theorem 1. *Let Γ be a template environment,* **define** $\{\psi_1\} f (\overline{x_i :: \tau_i}) :: (\overline{y_j :: \tau'_j}) = e\ \{\psi_2\}$ *be a function definition, $\hat{S}$ a template scheme, and Φ a set of constraints such that*
VCGD $(\Gamma,$ **define** $\{\psi_1\} f (\overline{x_i :: \tau_i}) :: (\overline{y_j :: \tau'_j}) = e\ \{\psi_2\}) = (\hat{S}, \Phi)$.
Then for every mapping A satisfying Φ, it holds that

$$\Gamma \vdash_A \textbf{define}\ \{\psi_1\} f (\overline{x_i :: \tau_i}) :: (\overline{y_j :: \tau'_j}) = e\ \{\psi_2\} :: \hat{S}.$$

Proof. It is a direct consequence of Propositions 1 and 2 (see the electronic appendix). □

8.4 Completeness of the VCG Algorithm

Here we prove that *VCG* as well as *VCGD* are complete with respect to the liquid type system. First, we prove the completness of *subt* with respect to the subtype system.

Proposition 3. *Let* Γ *be a template environment,* $\widehat{LT}_1, \widehat{LT}_2$ *be two template types, and* A *be a mapping from template variables to predicates. If* $\Gamma \vdash_A \widehat{LT}_1 <: \widehat{LT}_2$ *has a derivation in the subtype system, then there is a set of constraints* Φ *such that* $subt(\Gamma, \widehat{LT}_1, \widehat{LT}_2) = \Phi$ *and* A *satisfies* Φ.

Proof. By induction on the depth of the proof of $\Gamma \vdash_A \widehat{LT}_1 <: \widehat{LT}_2$. The full proof is detailed in the electronic appendix. □

Proposition 4. *Let* Γ *be an environment,* e *an expression, and* T *a nonfunctional type. If* $\Gamma \vdash e :: T$ *has a derivation in the liquid type system, then for any template environment* Γ' *and any mapping from template variables to predicates* A *such that* $A(\Gamma') = \Gamma$*, there exist a template type* $\hat{T}$*, a mapping* A' *extending* A*, and a set of constraints* Φ *such that*

- $VCG(\Gamma', e) = (\hat{T}, \Phi)$,
- $\Gamma \vdash A'(\hat{T}) <: T$,
- A' *satisfies* Φ.

Proof. By induction on the depth of the derivation of $\Gamma \vdash e :: T$. We show here the proof when the last rule applied in the derivation is (*LTcase*). The rest are in the electronic appendix.

Let Γ' and A be such that $A(\Gamma') = \Gamma$. If (*LTcase*) is the last rule of the derivation of $\Gamma \vdash e :: T$, then $e = \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij} :: \tau_{ij}} \to e_i}$, $hmtype(\Gamma(x)) = \tau$, and for all i we have $\Gamma(C_i) = S_i$ and $Inst(\langle \overline{x_{ij} :: B_{ij}} \rangle \to B_i, S_i)$. In addition,

(i) $\Gamma, \overline{x_{ij} :: B_{ij}}, \varphi_i[x/v] \vdash e_i :: T$, $\overline{\overline{x_{ij}}}$ are not free in T,
$B_{ij} = \{v : \tau_{ij} \mid \varphi_{ij}\}$ and $B_i = \{v : \tau \mid \varphi_i\}$.

From $hmtype(\Gamma(x)) = \tau$, we have $hmtype(\Gamma'(x)) = \tau$. Since $A(\Gamma') = \Gamma$, for every i, $\Gamma(C_i) = S_i$, there is $\hat{S}_i$ such that $\Gamma'(C_i) = \hat{S}_i$ and $A(\hat{S}_i) = S_i$, for each i. Moreover, $Inst(\langle \overline{x_{ij} :: B_{ij}} \rangle \to B_i, S_i)$ implies that if $\langle \overline{x_{ij} :: \hat{B}_{ij}} \rangle \to \hat{B}_i = get\_inst(\hat{S}_i, (\overline{x_{ij} :: \tau_{ij}}))$, then $A(\langle \overline{x_{ij} :: \hat{B}_{ij}} \rangle \to \hat{B}_i) = \langle \overline{x_{ij} :: B_{ij}} \rangle \to B_i$. So, $\hat{B}_i = \{v : \tau \mid \psi_i\}, A(\psi_i) = \varphi_i$ and $\hat{B}_{ij} = \{v : \tau_{ij} \mid \psi_{ij}\}, A(\psi_{ij}) = \varphi_{ij}$. Hence, applying the induction hypothesis to (i), for each i, there are $\hat{T}_i$, A_i (extension of A), and Φ_i such that

(ii) $VCG(\Gamma'; \overline{x_{ij} :: \hat{B}_{ij}}; \psi_i[x/v], e_i) = (\hat{T}_i, \Phi_i)$,
(iii) $\Gamma; \overline{x_{ij} :: B_{ij}}; \varphi_i[x/v] \vdash A_i(\hat{T}_i) <: T$,
(iv) A_i satisfies Φ_i.

Without loss of generality, we can assume that the new template variables occurring in $\hat{T}_i$ and Φ_i are different for each i, so the domains of the mapping extensions A_i only have in common the domain of A. Then there is $A' = \bigcup_i A_i$, a common extension of A, satisfying (iii) and (iv) replacing A_i by A', for every i.

Hence, there exists an extension A' of A, a template type $\hat{T} = \bigvee_i \exists \overline{\overline{x_{ij}}}.(\hat{T}_i \wedge \bigwedge_j \psi_{ij}[x_{ij}/v] \wedge \psi_i[x/v])$, and a set of constraints $\Phi = \bigcup_i \Phi_i$ such that the result of $VCG(\Gamma', \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij} :: \tau_{ij}} \to e_i})$ is $(\hat{T}, \bigcup_i \Phi_i)$, because $hmtype(\Gamma'(x)) = \tau$, and for every i, $\Gamma'(C_i) = \hat{S}_i$, $get\_inst(\hat{S}_i, (\overline{x_{ij} :: \tau_{ij}})) = \langle \overline{x_{ij} :: \hat{B}_{ij}} \rangle \to \hat{B}_i$ and (ii). Moreover, A' satisfies Φ, by (iv).

Now we need to prove $\Gamma \vdash A'(\hat{T}) <: T$. Since A' verifies (iii) for every i, by Lemma 9 we obtain that

$$\Gamma \vdash A'\left(\hat{T}_i \wedge \bigwedge_j \psi_{ij}[x_{ij}/v] \wedge \psi_i[x/v]\right) <: T.$$

By Lemma 6,

$$\Gamma \vdash A'\left(\exists\overline{\overline{x_{ij}}}.\left(\hat{T}_i \wedge \bigwedge_j \psi_{ij}[x_{ij}/v] \wedge \psi_i[x/v]\right)\right) <: T$$

for every i, because $\overline{\overline{x_{ij}}}$ are not free in T.

In addition, using Lemma 7,

$$\Gamma \vdash A'\left(\bigvee_i \exists\overline{\overline{x_{ij}}}.\left(\hat{T}_i \wedge \bigwedge_j \psi_{ij}[x_{ij}/v] \wedge \psi_i[x/v]\right)\right) <: T$$

as we wanted to prove. □

Theorem 2. *Let Γ be a template environment and $e' = \textbf{define}\,\{\psi_1\}f(\overline{x_i :: \tau_i}) :: (\overline{y_j :: \tau'_j}) = e\,\{\psi_2\}$ be a function definition. If*

$$\Gamma \vdash \textbf{define}\,\{\psi_1\}f(\overline{x_i :: \tau_i}) :: (\overline{y_j :: \tau'_j}) = e\,\{\psi_2\} :: S,$$

then for any template environment Γ' and any mapping from template variables to predicates A such that $A(\Gamma') = \Gamma$, there are a template scheme $\hat{S}$, a mapping A' extending A, and a set of constraints Φ such that

- VCGD $(\Gamma', e') = (\hat{S}, \Phi)$,
- $A'(\hat{S}) = S$,
- *A' satisfies Φ.*

Proof. Assuming that (*LTdef*) has been applied to obtain the derivation of $\Gamma \vdash_A e' :: S$, then it is the case that $S = gen(\langle\overline{x_i :: B_i}\rangle \to \langle\overline{y_j :: B'_j}\rangle, \Gamma)$, where $B_i = \{\nu : \tau_i \mid \varphi_i\}$, $B'_j = \{\nu : \tau'_j \mid \varphi'_j\}$ and

(i) $\Gamma,\ f :: \langle\overline{x_i :: B_i}\rangle \to \langle\overline{y_j :: B'_j}\rangle,\ \overline{x_i :: B_i} \vdash e :: \langle\overline{y_j :: B'_j}\rangle$,
(ii) $\psi_1 \Rightarrow \bigwedge_i \varphi_i[x_i/v]$,
(iii) $\bigwedge_j \varphi'_j[y_j/v] \Rightarrow \psi_2$.

Let $\langle\overline{x_i :: \hat{B}_i}\rangle \to \langle\overline{y_j :: \hat{B}'_j}\rangle = THM(\overline{\Gamma'}, f(\overline{x_i :: \tau_i}) :: (\overline{y_j :: \tau'_j}) = e)$, $\hat{B}_i = \{\nu : \tau_i \mid \hat{\varphi}_i\}$, $\hat{B}'_j = \{\nu : \tau'_j \mid \hat{\varphi}'_j\}$.

Then there is A' extension of A such that $A'(\langle\overline{x_i :: \hat{B}_i}\rangle \to \langle\overline{y_j :: \hat{B}'_j}\rangle) = \langle\overline{x_i :: B_i}\rangle \to \langle\overline{y_j :: B'_j}\rangle$. Moreover, if we denote $gen(\langle\overline{x_i :: \hat{B}_i}\rangle \to \langle\overline{y_j :: \hat{B}'_j}\rangle, \Gamma')$ by $\hat{S}_i$, we have

(iv) $\overline{A'(\hat{\varphi}_i)} = \overline{\varphi_i}$ and $\overline{A'(\hat{\varphi}'_j)} = \overline{\varphi'_j}$,
(v) $A'(\hat{S}_i) = S_i$.

Then, applying Proposition 4 to (i), there is A'' extending A', $\hat{T}_e$ and Φ_1 such that

(vi) $VCG(\Gamma'; f :: \langle\overline{x_i :: \hat{B}_i}\rangle \to \langle\overline{y_j :: \hat{B}'_j}\rangle; \overline{x_i :: \hat{B}_i}, e) = (\hat{T}_e, \Phi_1)$,
(vii) $\Gamma,\ f :: \langle\overline{x_i :: B_i}\rangle \to \langle\overline{y_j :: B'_j}\rangle,\ \overline{x_i :: B_i} \vdash A''(\hat{T}_e) <: \langle\overline{y_j :: B'_j}\rangle$,
(viii) A'' satisfies Φ_1.

Therefore, by Proposition 3 and (vii), there is Φ_2 such that

(ix) $\Phi_2 = subt(\Gamma'; f :: \langle \overline{x_i :: \hat{B}_i} \rangle \to \langle \overline{y_j :: \hat{B}'_j} \rangle; \overline{x_i :: \hat{B}_i}, \hat{T}_e, \langle \overline{y_j :: \hat{B}'_j} \rangle)$,
(x) A'' satisfies Φ_2.

In addition, A'' satisfies $\psi_1 \Rightarrow \bigwedge_i \hat{\varphi}_i[x_i/v]$, because A'' is an extension of A', (ii), and (iv), and A'' satisfies $\bigwedge_j \hat{\varphi}'_j[y_j/v] \Rightarrow \psi_2$, because A'' is an extension of A', (iii), and (iv). So we can guarantee that there exists A'', extending A, a set of constraints $\Phi = \Phi_1 \cup \Phi_2 \cup \{\psi_1 \Rightarrow \bigwedge_i \hat{\varphi}_i[x_i/v], \bigwedge_j \hat{\varphi}'_j[y_i/v] \Rightarrow \psi_2\}$ such that A'' satisfies Φ (see (viii) and (x)), and $VCGD(\Gamma, e') = (\hat{S}, \Phi)$, by (vi) and (ix), where $A''(\hat{S}) = S$, by (v), since the template variables of $\hat{S}$ are in the domain of A' and A'' is an extension of A'. □

8.5 A Practical Verification Condition Generator

The VCG algorithm presented in prior sections has good mathematical properties, but it produces rather complex verification conditions. Given that the inference algorithm will discharge these conditions with the aid of an automatic verifier (an SMT solver), it is important to keep the formulas simple enough so that they can remain inside the decidable domain of the solver.

To this aim, the actually implemented algorithm presents some differences with respect to the ideal *VCG* presented up to now. In what follows, we will refer to the actual algorithm as VCG' when applied to an expression and as $VCGD'$ when applied to a function definition. These differences are the following:

(1) It does not generate existentially quantified formulas in **let** or **case** expressions.
(2) It does not generate disjunction of types in **case** expressions.
(3) It generates the constraints in the form of a set of *goals*. Given a function definition, the goals generated for it have the following shape:

$$[\![\Gamma]\!] \wedge \theta_1.\iota_1 \wedge \cdots \wedge \theta_n.\iota_n \Rightarrow \theta_{n+1}.\iota_{n+1}$$

where ι_i, $i = 1, \ldots, n+1$, denote template variables, and θ_i, $i = 1, \ldots, n+1$, denote program variable substitutions. The goal represents the set of constraints that must hold in a path through the function body. The template variables in the left-hand side represent the predicates held by each function argument, whereas the one in the right-hand side corresponds either to the predicate held by the function result or by one of the arguments of a function application. The substitutions arise in the function application rule of *VCG* (see Section 8.2).

As a practical issue, VCG' starts with an initial typing environment holding very precise predefined liquid types for function symbols belonging to the solver decidable theories, such as $+$, $-$, $<$, $\leq$, *get*, *set*, . . .For instance, the type of $-$ is

$$(-) :: \langle x : \{v : int\}, y : \{v : int\} \rangle \to \{v : int \mid v = x - y\}.$$

When an expression such as **let** $z = x \oplus y$ **in** e is processed by VCG', being $\oplus$ a symbol predefined in the initial environment, the application and **let** rules force the constraint $z = x \oplus y$ to be added to the typing environment. As a consequence, many of the **let** binding expressions are kept in the current typing environment as additional conditions on the **let** bound variables. This feature is also assumed to be present in the VCG algorithm.

With respect to item (1), nothing essential is lost given the syntactic restrictions of the IR. As stated in Section 5, the IR programs are in A-normal form. So, a basic block consists of a sequence of **let** expressions ended either in a basic expression or in a **case** expression. In the latter case,

each branch of the **case** follows in turn the same pattern. What we call a goal collects in $[\![\Gamma]\!]$ the conditions of a complete path through the body of a function definition until either a function application or a basic expression e is found. These conditions have either the shape $x = be$, should they come from a **let** binding, or essentially the shape $y = C\,\overline{y_j}$, should they come from a **case** branch. So, a goal for a function f, in the case that no function application is found, can be depicted as follows:

$$\bigwedge_i (\theta_i.\iota_i) \wedge \bigwedge_i (\exists x_i.x_i = be_i) \wedge \bigwedge_i (\exists \overline{y_{ij}}.y_i = C\,\overline{y_{ij}}) \Rightarrow [e/res_f].\iota_{n+1}$$

where res_f is the name of the result returned by the function f being defined. So, all existential quantifiers are moved outside the implication as universal ones, and the formula is given in this form to the SMT solver to be checked for validity. If an application to a function g is found, then the latter substitution must replace the result returned by f by the result returned by g. In addition, some additional goals are created to imply the argument predicates of function g.

As an example, we show some of the goals arising when VCG' is applied to function f_3 of Figure 5:

$$\begin{array}{l}
\forall i, x, v, z, b\ .\ (\iota_{31} \wedge \iota_{32} \wedge \iota_{33} \wedge (z = v[i]) \wedge (b = x < z) \wedge b \Rightarrow id\ .\ \iota_{41}) \\
\forall i, x, v, z, b\ .\ (\iota_{31} \wedge \iota_{32} \wedge \iota_{33} \wedge (z = v[i]) \wedge (b = x < z) \wedge b \Rightarrow id\ .\ \iota_{42}) \\
\forall i, x, v, z, b\ .\ (\iota_{31} \wedge \iota_{32} \wedge \iota_{33} \wedge (z = v[i]) \wedge (b = x < z) \wedge b \Rightarrow id\ .\ \iota_{43}) \\
\forall i, x, v, z, b, res_3, res_4\ .\ (\iota_{31} \wedge \iota_{32} \wedge \iota_{33} \wedge (z = v[i]) \wedge (b = x < z) \wedge b \Rightarrow [res_3/res_4]\ .\ \iota_{res_4}),
\end{array}$$

where ι_{ij} is the template variable corresponding to the argument j of function f_i. In this example, the first three substitutions are the identity because function f_4 is being called with its actual arguments having the same names as the formal ones, but this is not necessarily the case.

With respect to item (2), and when applying the **case** rule, VCG' never generates a disjunction of types. Instead, it assumes a fresh type T for the **case** expression. This fresh type is usually the template variable of a postcondition, which must hold in all **case** branches. Let us call ι_T to this variable, and let us denote ϕ_i to the condition collected while traversing the branch i of the **case**. Then VCG' generates the conjunction of the following set of goals:

$$\{\phi_i \Rightarrow \theta.\iota_T \mid i = 1, \ldots, n\}$$

being n the number of branches of the **case** expression. This is logically equivalent to the single goal:

$$\left(\bigvee_i \phi_i\right) \Rightarrow \theta.\iota_T.$$

When the corresponding constraints are solved, it is not guaranteed that T will be the minimum possible type, so it may be bigger than the disjunction in the left-hand side. In this sense, VCG' still gives a sound type, but it may lose completeness with respect to the ideal one.

To generate a goal belonging to a function definition $f\,(\overline{x_i :: \tau_i}) :: (\overline{y_j :: \tau'_j}) = e$, VCG' first introduces in the environment Γ the type bindings of the arguments $\overline{x_i :: \tau_i}\ \overline{x_i :: \hat{B}_i}$, with $hmtype(\hat{B}_i) = \tau_i$, and then it infers the type of e, as expressed in the rule ($LTfun$) of Figure 6. When VCG' traverses e, it collects in the environment all conditions arising from **let** bindings and from the pattern matchings done as long as **case** branches are taken in the path. Finally, the goal ends up in an implication to a template variable, possibly affected by a pending substitution.

For all of these reasons, in what follows we will denote a goal in the simplified form $[\![\Gamma]\!] \Rightarrow \theta\ .\ \iota$, or even simpler, $\phi \Rightarrow \theta\ .\ \iota$.

9 THE ITERATIVE WEAKENING ALGORITHM

First, we introduce the different kinds of template variables used by the algorithm and the sets of qualifiers used to instantiate these variables. The algorithm is then explained in detail. Finally, its soundness and relative completeness are investigated.

9.1 Refinement Templates

As stated, the simplest form of a template type is a basic refined type whose refinement predicate contains exactly one template variable. The template type of a variable x which is not an array contains a κ template variable as usual, $x : \{\nu : \tau \mid \kappa\}$. The range of a A for κ variables consists of a conjunction of qualifiers taken from the set $\mathbb{Q}^*$, which is obtained from a user-given set $\mathbb{Q}$ by substituting program variables in scope in this text location, and of the appropriate type, for the wildcard $\star$.

The template type of an array variable a is dealt with similarly, except for the fact that the template variable is denoted by μ, $a : \{\nu : array\ \tau \mid \mu\}$. The range of $A(\mu)$ is an *array refinement*, which consists of a conjunction of *array refinement templates*, by previously substituting the template variable inside the template type. These templates may have three possible shapes:

- Simple array refinement templates, $\rho \stackrel{\text{def}}{=} (\forall j.\eta \rightarrow q)$, where q is a qualifier taken from the set $\mathbb{Q}^*_E$ (see the following) and η is a template variable.
- Double array refinement templates, $\rho\rho \stackrel{\text{def}}{=} (\forall j_1, j_2.\eta\eta \rightarrow q)$, where q is a qualifier taken from the set $\mathbb{Q}^*_{EE}$ (see the following) and $\eta\eta$ is a template variable.
- A length refinement template variable ζ, which refines the array length. This variable will be instantiated by a conjunction of properties restricting the array length.

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 $A(\rho) = (\forall j.A(\eta) \rightarrow q)$ and $A(\rho\rho) = (\forall j_1, j_2.A(\eta\eta) \rightarrow q)$. The range of $A(\eta)$ are conjunctions of qualifiers taken from the set $\mathbb{Q}^*_I$; the range of $A(\eta\eta)$ are conjunctions of qualifiers taken from the set $\mathbb{Q}^*_{II}$, and the range of $A(\zeta)$ are conjunctions of qualifiers taken from the set $\mathbb{Q}^*_{len}$.

The previously mentioned sets of qualifiers, $\mathbb{Q}^*_I$, $\mathbb{Q}^*_I$, $\mathbb{Q}^*_E$, $\mathbb{Q}^*_{EE}$, and $\mathbb{Q}^*_{len}$, are obtained by instantiating specific wildcards in the corresponding following sets with variables in scope of the appropriate type:

- The set $\mathbb{Q}_I$ uses $\star$ and # as wildcards in the qualifiers, and only the bound variable j can be substituted for the wildcard #. The # must occur in each qualifier.
- The set $\mathbb{Q}_{II}$ uses $\star$, $\#_1$ and $\#_2$ as wildcards in the qualifiers, and only the bound variables j_1 and j_2 can be respectively substituted for the wildcards $\#_1$ and $\#_2$. At least $\#_1$ or $\#_2$ must occur in each qualifier.
- The sets $\mathbb{Q}_I$ and $\mathbb{Q}_{II}$ are such that when wildcards are instantiated, the corresponding qualifiers satisfy the restrictions imposed on the guards of array property formulas as explained in Section 3.
- The set $\mathbb{Q}_E$ uses $\star$ and # as wildcards in the qualifiers, and only the bound variable j can be substituted for the wildcard #. This wildcard only occurs in expressions of the form $\nu[\#]$, and this expression must occur in each qualifier.
- The set $\mathbb{Q}_{EE}$ uses $\star$, $\#_1$, and $\#_2$ as wildcards in the qualifiers, and only the bound variables j_1 and j_2 can be respectively substituted for the wildcards $\#_1$ and $\#_2$. These wildcards only occur in expressions of the form $\nu[\#_1]$, $\star[\#_1]$, $\nu[\#_2]$, and $\star[\#_2]$, and both wildcards, and ν must occur in each qualifier.

- $\mathbb{Q}_E$ and $\mathbb{Q}_{EE}$ are such that the corresponding instantiated qualifiers satisfy the restrictions imposed on the value constraint formulas of the array property (see Section 3).
- The set $\mathbb{Q}_{len}$ uses $\star$ as the wildcard, and each qualifier must mention either $len\, \nu$ or $len\, \star$.

Example 5. In the *insert* algorithm, the following predicate is part of the refinement type for v:

$$\forall j_1, j_2.0 \leq j_1 \leq i \wedge i + 2 \leq j_2 \leq n \rightarrow \nu[j_1] \leq \nu[j_2].$$

It can be obtained from the template $\rho\rho \stackrel{\text{def}}{=} (\forall j_1, j_2.\eta\eta \rightarrow q)$, and the sets $\mathbb{Q}_{II} = \{0 \leq \#_1, \star + 2 \leq \#_2, \#_1 \leq \star, \#_2 \leq \star\}$ and $\mathbb{Q}_{EE} = \{\nu[\#_1] \leq \nu[\#_2]\}$.

In the *merge* algorithm, the following predicate is part of the refinement type for v:

$$\forall j_1, j_2.a \leq j_1 \leq k - 1 \wedge i \leq j_2 \leq m \rightarrow \nu[j_1] \leq w[j_2].$$

It can be obtained from c, and the sets $\mathbb{Q}_{II} = \{\star \leq \#_1, \star \leq \#_2, \#_1 \leq \star - 1, \#_2 \leq \star\}$ and $\mathbb{Q}_{EE} = \{\nu[\#_1] \leq \star[\#_2]\}$. Another part of v's refinement type expresses that a segment is sorted:

$$\forall j_1, j_2.a \leq j_1 \leq j_2 \leq k - 1 \rightarrow \nu[j_1] \leq \nu[j_2].$$

It can be obtained from the same template, by adding the qualifier $\#_1 \leq \#_2$ to the set $\mathbb{Q}_{II}$ and the qualifier $\nu[\#_1] \leq \nu[\#_2]$ to the set $\mathbb{Q}_{EE}$.

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

We will say that a mapping A is *suitable to* 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 $0 \leq j < len\, \nu$, where j is the universally quantified variable in ρ, and for each $\eta\eta$ variable of a $\rho\rho$ template, $A(\eta\eta)$ contains $0 \leq j_1 < len\, a \wedge 0 \leq j_2 < len\, b$, 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 $\rho\rho$. We will denote the set of all mappings suitable to a given Q as A_Q.

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

In what follows, we will make the following abuse of notation. When we write $A(\diamond)$, where $\diamond$ can be a template type (or a scheme), an environment, a constraint, or a set of constraints, we will denote the object $\diamond$, after replacing every template variable ι inside $\diamond$ by $A(\iota)$.

The final aim of the type inference algorithm is to find a mapping A suitable to a given Q such that A is a solution of all generated verification conditions. We start with the strongest possible mapping and weaken it until a solution is found, in accordance with the following definition.

Definition 2. Given a set of constraints Φ and a collection $Q = \{\mathbb{Q}, \mathbb{Q}_I, \mathbb{Q}_{II}, \mathbb{Q}_E, \mathbb{Q}_{EE}, \mathbb{Q}_{len}\}$, we say that A is a *solution of* Φ *with respect to* Q if $A \in A_Q$ and A satisfies Φ.

Notice that κ, μ, ζ variables occur in logically positive positions in the templates, whereas 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.

9.2 The Iterative Weakening Algorithm

Given a set of goals Φ and a collection $Q = \{\mathbb{Q}, \mathbb{Q}_I, \mathbb{Q}_{II}, \mathbb{Q}_E, \mathbb{Q}_{EE}, \mathbb{Q}_{len}\}$, the purpose of the algorithm is to find a solution to Φ with respect to to Q.

Next, we describe the steps of the weakening algorithm. It starts with the strongest possible mapping A suitable to Q. This consists of the following:

(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, which is $0 \leq j < len\, v$, 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, which is $0 \leq j_1 < len\, a \wedge 0 \leq j_2 < len\, b$, where j_1, j_2 are the universally quantified variables in $\rho\rho$; a and b are either v, or a free array variable in scope substituted for $\star$ in the qualifier at the right-hand side of $\rho\rho$.

Example 6. In the *binSearch* algorithm of Figure 2, we have $\mathbb{Q}^*_E = \{x \leq v[j], x > v[j]\}$, $\mathbb{Q}^*_I = \{j \leq a - 1,\ b + 1 \leq j\}$. If we denote by μ_3 the template variable corresponding to the array v at the beginning of each iteration, then the refinement

$$(\forall j\ .\ 0 \leq j \wedge j < len\ v \rightarrow x \leq v[j]) \wedge (\forall j\ .\ 0 \leq j \wedge j < len\ v \rightarrow x > v[j]) \tag{13}$$

will be included in the strongest assignment to μ_3.

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

The crucial step is then how to weaken the mapping A to satisfy a goal φ. A difference with the standard algorithm of Rondon et al. [27] is that, in our case, weakening A may change the goals themselves and may introduce new template variables. Let us see the process in detail:

(1) If φ has the form $[\![\Gamma]\!] \Rightarrow \theta.\kappa$, and $A(\kappa) = q_1 \wedge \cdots \wedge q_r$, then the weakening removes from $A(\kappa)$ all qualifiers q_i such that the formula $A([\![\Gamma]\!])) \Rightarrow \theta.q_i$ is not valid. This approach corresponds to the standard weakening of Rondon et al. [27].

    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 on these ζ variables.
(2) If φ has the form $[\![\Gamma]\!] \Rightarrow \theta.\mu$, and $A(\mu) = A(\xi_1) \wedge \cdots \wedge A(\xi_r)$, in a first step the weakening removes from $A(\mu)$ all refinements $A(\xi_i)$ such that the formula $A([\![\Gamma]\!]) \Rightarrow \theta.A(\xi_i)$ is not valid and cannot 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—that is, the conjunction of all 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 despite 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 form of each 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 built from A, by 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 7. In the *merge* algorithm described in Figure 3, the following formula cannot be proved to be valid as a refinement $A(\mu)$ of the result array u:

$$\forall j_1, j_2.a \leq j_1 \leq k-1 \wedge 0 \leq j_2 < len\, v \rightarrow v[j_1] \leq w[j_2].$$

This is because the mapping $A(\eta) = \{a \leq j_1, j_1 \leq k-1, 0 \leq j_2, j_2 < len\, v\}$ is too weak. By creating a fresh refinement and a fresh variable η' and doing $A'(\eta) = A(\eta) \cup \{i \leq j_2, j_2 \leq m\}$, $A'(\eta') = A(\eta) \cup \{j \leq j_2, j_2 \leq b\}$, we get

$$\begin{array}{l}(\forall j_1, j_2.a \leq j_1 \leq k-1 \wedge 0 \leq j_2 < len\, v \wedge i \leq j_2 \leq m \rightarrow v[j_1] \leq w[j_2]) \\ \wedge(\forall j_1, j_2.a \leq j_1 \leq k-1 \wedge 0 \leq j_2 < len\, v \wedge j \leq j_2 \leq b \rightarrow v[j_1] \leq w[j_2]),\end{array}$$

which can be shown to be a valid refinement. This refinement means that the elements of the already-sorted segment $v[0..k-1]$ in the result are less than or equal to those in the segments $w[i..m]$, $w[j..b]$, which are still to be sorted.

Example 8. 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 (13), but it can be made valid by strengthening its antecedent, since for instance the first conjunct of (13) becomes

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

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

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

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.

9.3 Soundness, Completeness, and Cost of the Weakening Algorithm

Fixed a family of qualifier sets Q, we will show the following properties:

(1) The weakening algorithm always terminates.
(2) If the weakening algorithm terminates with **failure**, then there exists no mapping $A \in A_Q$ satisfying all of the goals.
(3) If the weakening algorithm terminates with **success**, then not only the mapping found A satisfies all of the goals but also A is the strongest possible mapping satisfying them in A_Q.

We will start by showing that the search space (i.e., the set A_Q of mappings suitable to Q) is a complete lattice, with the following definition of $\sqsubseteq$.

Definition 3. Let $A, A' \in A_Q$. We say that $A \sqsubseteq A'$ if for all κ, $A(\kappa) \Rightarrow A'(\kappa)$, and for all μ, $A(\mu) \Rightarrow A'(\mu)$.

This relation is indeed a partial order, where = corresponds to logical equivalence of formulas. In fact, it is a complete lattice in which the bottom-most (and hence strongest) element is the initial mapping A_0.

Theorem 3. *The partial ordered set $(A_Q, \sqsubseteq)$ is a (finite) complete lattice.*

Our goal now is to prove that each step of the weakening algorithm produces an output mapping weaker than the input one.

Proposition 5. *Let $A \in A_Q$. If $A(\mu) = A(\xi)$ and $A'(\mu) = A'(\xi_1) \wedge \cdots \wedge A'(\xi_s)$ is obtained by the process described in the step 3 of the weakening algorithm, then $A' \in A_Q$ and $A \sqsubseteq A'$.*

Proof. For the sake of simplicity, let us assume that $\xi = (\forall j.\eta_1 \rightarrow q)$ and for all i, $\xi_i = (\forall j.\eta_i \rightarrow q)$. In addition, let us assume $A(\eta_1) = \mathbb{Q}_1$.

We remind that $(\forall j.\varphi_1 \rightarrow q) \wedge \cdots \wedge (\forall j.\varphi_s \rightarrow q)$ is equivalent to $(\forall j.\varphi_1 \vee \cdots \vee \varphi_s \rightarrow q)$. Moreover, for all $i = 1..s$, $A'(\eta_i) = \mathbb{Q}_1 \cup \mathbb{Q}'_i$ for some conjunction of qualifiers $\mathbb{Q}'_i$, and therefore

$$\begin{aligned} & A'(\mu) = A'(\xi_1) \wedge \cdots \wedge A'(\xi_s) \\ \Leftrightarrow\ & \forall j.((\mathbb{Q}_1 \wedge \mathbb{Q}'_1) \vee \cdots \vee (\mathbb{Q}_1 \wedge \mathbb{Q}'_s)) \rightarrow q \\ \Leftrightarrow\ & \forall j.\mathbb{Q}_1 \wedge (\mathbb{Q}'_1 \vee \cdots \vee \mathbb{Q}'_s) \rightarrow q. \end{aligned}$$

The antecedent of $A'(\mu)$ is stronger than the antecedent of $A(\mu) = (\forall j.A(\eta_1) \rightarrow q) = (\forall j.\mathbb{Q}_1 \rightarrow q)$, so $A(\mu) \sqsubseteq A'(\mu)$. □

Theorem 4. *Let $A \in A_Q$. If A' is obtained from A by one step of the inference algorithm, then $A' \in A_Q$ and $A \sqsubseteq A'$.*

Proof. That steps 1 and 2 of the weakening process described in Section 9.2 lead to a mapping A' weaker than the prior one A is rather trivial, because some conjuncts are being removed from the mapping of κ, ζ, or μ variables. That the result holds for the step 3 is a direct application of Proposition 5. □

We show now that given a fixed set of goals Φ, if there exists one or more mappings in A_Q that turn all of the goals in C into valid formulas, then

(1) There exists a minimum or *strongest mapping* A^s in A_Q which makes Φ valid (Theorem 5).
(2) All mappings $A \in A_Q$ produced by the inference algorithm are below A^s (i.e., $A \sqsubseteq A^s$) (see Theorem 6).

Theorem 5. *Given a set Φ of goals, if there exists a mapping $A \in A_Q$ that is a solution with respect to Φ, then there exists a minimum mapping $A^s \in A_Q$ such that $A^s(\Phi)$ is valid.*

Proof. It is enough to show that for every pair of mappings A_1, A_2 making Φ valid, their greatest lower bound $A_1 \sqcap A_2$ is also a solution of Φ. As the set of mappings making Φ valid is finite, its minimum element would satisfy Φ and this would be A^s.

We make note that the definition of $A_1 \sqcap A_2$ is the same for both a κ and for a μ template variables. If we indistinctly denote by ι to any one of these variables, we have defined in both cases $(A_1 \sqcap A_2)(\iota) = A_1(\iota) \cup A_2(\iota)$. Since the range of a mapping is a set of formulas that we interpret as the conjunction of all of them, we will write $A_1(\iota) \cup A_2(\iota)$ as $A_1(\iota) \wedge A_2(\iota)$. Let $\varphi \in \Phi$ be any of the goals, and let us assume that it has the form $\phi \Rightarrow \theta.\iota$, where ϕ is a formula possibly having

additional template variables and θ is a pending substitution. Then by hypothesis, we have the following:

$$A_1(\phi) \Rightarrow \theta.A_1(\iota) \text{ is valid}$$
$$A_2(\phi) \Rightarrow \theta.A_2(\iota) \text{ is valid.}$$

It is clear that $A_1(\phi) \wedge A_2(\phi) \Rightarrow \theta.(A_1(\iota) \wedge A_2(\iota))$ is also valid. So, there exists a minimum or strongest mapping A^s such that $A^s(\Phi)$ is valid. □

Theorem 6. *If the set Φ of goals has a strongest solution $A^s \in A_Q$, and A is a mapping produced by the inference algorithm, then $A \sqsubseteq A^s$.*

Proof. By induction on the number of weakening steps done by the algorithm. For the initial mapping A_0, we have $A_0 \sqsubseteq A^s$ because A_0 is the minimum of the lattice A.

Let $0 \leq n$, and assume by induction hypothesis that for every k, $0 \leq k \leq n$, $A_k \sqsubseteq A^s$. We prove that the weakening steps produce a new mapping A_{n+1} such that $A_{n+1} \sqsubseteq A^s$. Let ι' be the template variable at the right-hand side of the goal solved in the step $n+1$, since $A_{n+1}(\iota) = A_n(\iota)$, for any $\iota \neq \iota', A_{n+1}(\iota) \Rightarrow A^s(\iota)$, by the induction hypothesis. Let us prove that $A_{n+1}(\iota') \Rightarrow A^s(\iota')$ by reductio ad absurdum. Suppose there is an element $\varphi \in A^s(\iota')$ such that $A_{n+1}(\iota') \not\Rightarrow \varphi$. We distinguish two cases:

- If ι' is a κ variable, then φ is a qualifier q such that $q \notin A_{n+1}(\kappa)$. Hence, there is a previous mapping, A_k, $k \leq n$, such that q was eliminated from $A_k(\kappa)$ when solving a goal $\phi \Rightarrow \theta.\kappa$, because $A_k(\phi) \not\Rightarrow \theta.q$. Then $A^s(\phi) \not\Rightarrow \theta.q$, since $A_k \sqsubseteq A^s$ by the induction hypothesis. But this is a contradiction, because A^s is a solution of Φ and then it holds $A^s(\phi) \Rightarrow \theta.A^s(\kappa)$. So, $A_{n+1}(\iota') \Rightarrow A^s(\iota')$ for κ variables.
- If ι' is a μ variable, then φ is an instance of a template ξ. We show the simple case $\xi = \forall j.\eta \rightarrow q$. The proof for a double template is similar.
  If $A_{n+1}(\mu) \not\Rightarrow (\forall j.A^s(\eta) \rightarrow q)$, then we can affirm that there is no instance of ξ, $(\forall j.Q \rightarrow q) \in A_{n+1}(\mu)$, such that $Q \subseteq A^s(\eta)$. Notice that, assuming A^s is a solution of C and $(\forall j.A^s(\eta) \rightarrow q) \in A^s(\mu)$, we can affirm that these instances of ξ have not been discarded in step 2 of the algorithm. Then for any $Q \subseteq A^s(\eta)$ there is $k \leq n$ such that a goal with μ at the right-hand side, say $\phi \Rightarrow \theta.\mu$, has been solved in accordance with step 3 of the algorithm, at the k iteration and $\forall j.Q \rightarrow q$ has been discarded because

$$A_k(\phi) \not\Rightarrow \theta.(\forall j.Q \rightarrow q). \tag{14}$$

  Otherwise $(\forall j.Q' \rightarrow q) \in A_{n+1}(\mu)$, with $Q' \subseteq Q \subseteq A^s(\eta)$, contradicting that $A_{n+1}(\mu) \not\Rightarrow (\forall j.A^s(\eta) \rightarrow q)$. Since for every $Q \subseteq A^s(\eta)$ there is k such that (14) holds, taking $Q = A^s(\eta)$, and by applying the induction hypothesis $A_k \sqsubseteq A^s$, we conclude that $A^s(\phi) \not\Rightarrow \theta.(\forall j.A^s(\eta) \rightarrow q)$, which is a contradiction because A^s is a solution of C. Then $A_{n+1}(\iota') \Rightarrow A^s(\iota')$ also for μ variables.

Therefore, $A_{n+1} \sqsubseteq A^s$, finishing the proof of $A \sqsubseteq A^s$. □

Putting it all together, let Γ be the initial template environment obtained in phase 1 (see Section 7) of the process of verifying a block definition $bd = \textbf{define}\ \{\psi_1\} f\,(\overline{x_i :: \tau_i}) :: (\overline{y_j :: \tau'_j}) = e\ \{\psi_2\}$, using the qualifier set Q. If $VCGD'(\Gamma, bd) = (\hat{S}, \Phi)$, then under the conditions of A_Q being a finite complete lattice, and by Theorems 5 and 6, if a solution with respect to Q exists for Φ, then the weakening algorithm terminates, and it gives the strongest mapping A_s as a result. In addition, and in accordance with Theorem 1, there is a derivation of the sequent $A_s(\Gamma) \vdash bd :: A_s(\hat{S})$ in the liquid type system.

The cost of the iterative weakening algorithm is dominated by the number of formulas sent for validity checking to the SMT solver. This, in turn, takes place during the weakening of a template variable. To compute a cost expression, let us first introduce some size parameters:

- Let $n_\kappa, n_\mu, n_\zeta, n_\eta$, and $n_{\eta\eta}$ respectively be the number of κ, μ, ζ, η, and $\eta\eta$ variables of the inferred program.
- Let $n_*, n_I, n_{II}, n_E, n_{EE}$, and n_{len} respectively be the maximum sizes of sets $\mathbb{Q}^*$, $\mathbb{Q}^*_I$, $\mathbb{Q}^*_{II}$, $\mathbb{Q}^*_E$, $\mathbb{Q}^*_{EE}$, and $\mathbb{Q}^*{}_{len}$ in the different program locations.

Then the maximum number of weakenings a κ variable may undergo is n_*, and each one involves proving a formula. Similarly, the number of weakenings μ and ζ variables may undergo respectively are $n_E + n_{EE}$ and n_{len}, since these are the number of conjuncts that each of these variables is mapped to after the initialization. But sometimes an η or $\eta\eta$ variable must be strengthened to weaken a conjunct ξ. This strengthening is in fact a search among the set of subsets of respectively $\mathbb{Q}^*_I$ and $\mathbb{Q}^*_{II}$. This involves proving in the worst case 2^{n_I} and $2^{n_{II}}$ formulas. So, in the worst case, the total number of formulas submitted by the algorithm is in the following complexity class:

$$O(n_\kappa n_* + n_\zeta n_{len} + n_\mu(n_E + n_{EE}) + n_\eta 2^{n_I} + n_{\eta\eta} 2^{n_{II}}).$$

It can be appreciated that this time cost is very sensitive to the number of qualifiers in the sets $\mathbb{Q}^*_I$ and $\mathbb{Q}^*_{II}$.

10 IMPLEMENTATION AND RESULTS

To assess the feasibility of the algorithm introduced in this article, we have implemented a tool[5] that infers liquid types in an IR program (see Section 5). Our tool receives a file containing the following:

- The liquid types of the external functions that may be used in function definitions being analysed. These are kept in an initial environment Γ_0.
- The definitions of the $\mathbb{Q}$, $\mathbb{Q}_I$, $\mathbb{Q}_{II}$, $\mathbb{Q}_E$, $\mathbb{Q}_{EE}$, and $\mathbb{Q}_{len}$ sets.
- The definitions of the functions to analyse. Each one may have a precondition and/or postcondition, and may already include liquid types, if the user chooses to supply them.

Our implementation proceeds as follows. After performing a standard Hindley-Milner type checking, it traverses the types of all parameters and results of each function and transforms each of them into a liquid type containing a fresh template variable. If the user has already specified an explicit liquid type for a given parameter or result, no template variable will be generated for that type. Then for each template variable, it instantiates the $\mathbb{Q}$, $\mathbb{Q}_E$, etc. sets by replacing the $\star$ placeholders by variables in scope, so each template variable has its specific collection of $\mathbb{Q}^\star$, $\mathbb{Q}^\star_E$, etc. sets. Finally, the tool generates goals as described in the algorithm *VCG′* (Section 8.5) and applies the iterative weakening algorithm on them. In the latter phase, the goals are sent to the Z3 SMT solver [24], which determines their validity. The decision procedure of Z3 implements a specific logic for arrays (*CAL*) that is subsequently translated to the theory of uninterpreted functions [25].

We have applied the tool to a series of algorithms involving arrays. The simplest one (`fill`) fills all positions of an array with a fixed value. We also have implementations of insertion sort (functions `insert` and `insert_sort`), quicksort (functions `partition` and `quicksort`), two versions of selection sort (functions `minimum`, `sel_sort_simple` and `sel_sort_full`), implementations

[5] Available at https://github.com/manuelmontenegro/liquidarrays.

of linear search (`linsearch`), binary search (`binsearch_simple`, `binsearch`), and the Dutch National Flag algorithm [9, pp. 111–116] (function `dutch_flag`).

We have evaluated two versions of selection sort due to this example having two nested loops. In the first version (`sel_sort_simple`), we extract the inner loop—which computes the minimum value contained within a subvector—into its own function `minimum`, infer the type of the latter, and then analyse the outer loop. In an imperative language, this would be as if the programmer had manually supplied the postcondition of the inner loop. However, `sel_sort_full` combines both loops into a single function, so the invariants are inferred all at once, without the programmer's assistance.

Regarding binary search, `binsearch_simple` and `binsearch` differ in the case in which the element being searched for is not found. The former returns the index where the missing element should be inserted, whereas the latter returns `-1`.

In the electronic appendix, we show the specification given for each function. Figure 9 contains the invariants inferred. These invariants are obtained from the inferred refinements in the parameters of the auxiliary functions of each example by replacing the ν with the corresponding variable. For example, when translating `insert_sort` into IR, an auxilary function $f_1\ k\ v$ is generated. This function inserts $v[k]$ into the subarray $v[0..k)$, (assuming that the latter is sorted) to obtain another sorted array v' and calls itself recursively with $k+1$ and v' as arguments. The type inferred for f_1 is the following:

$$\begin{aligned} f_1\ ::\ & k : \{\nu : \mathit{int} \mid 0 \le \nu\} \\ & \to v : \{\nu : \mathit{array\ int} \mid \forall j_1, j_2 . 0 \le j_1 \le j_2 < k \to \nu[j_1] \le \nu[j_2] \wedge k \le \mathit{len}\,\nu\} \\ & \to \{\nu : \mathit{array\ int} \mid \forall j_1, j_2 . 0 \le j_1 \le j_2 < \mathit{len}\,\nu \to \nu[j_1] \le \nu[j_2]\}. \end{aligned}$$

If we replace each ν in the refinement types of k and v by their corresponding program variables, we get the invariant shown in Figure 9.

Figure 10 contains execution statistics for each example. The *#G* column shows the number of goals generated. However, some of them are valid for any assignment of the template variables, so they are discarded and hence not sent to the iterative weakening algorithm. The *#NG* column shows the number of goals remaining after this preliminary screening. The number of template variables generated for each example is shown under $\#\kappa$ and $\#\mu$. The columns labelled with *#S* specify how many steps have taken the algorithm for each example. A step consists of finding a nonvalid goal and weakening its consequent until the goal becomes valid. A step involves sending a number of formulas to the SMT solver. The *#F* indicate how many formulas have been sent during the whole execution of iterative weakening.

As mentioned earlier, the users may specify liquid types in their functions. If every type is decorated with liquid types, then no template variables are generated, but the goals are still solved, so our tool is also suitable for type checking. The columns under **Typecheck** in Figure 10 show the execution statistics when the tool is used for this purpose. However, if no liquid types are provided by the user, they still have to specify a precondition and a postcondition, and let the system infer all intermediate types. The columns under **Full** contain the results when the tool is run under variants of the following sets:

$$\begin{aligned} \mathbb{Q} &= \{0 \le \nu, \star \le \nu, \star < \nu\} \\ \mathbb{Q}_E &= \{\star \le \nu[\#], \nu[\#] < \star, \nu[\#] = \star, \nu[\#] \ne \star, \nu[\#] = 0, \nu[\#] = 1, \nu[\#] = 2\} \\ \mathbb{Q}_I &= \{\star \le \#, \# < \star, \# = \star\} \\ \mathbb{Q}_{EE} &= \{\nu[\#_1] \le \nu[\#_2], \nu[\#_1] \le \star[\#_2]\} \\ \mathbb{Q}_{II} &= \{\star \le \#_1, \#_1 \le \star, \star \le \#_2, \#_2 \le \star, \#_1 \le \#_2, \#_1 = \star, \#_2 = \star\} \\ \mathbb{Q}_{len} &= \{\star < \mathit{len}\,\nu, \star \le \mathit{len}\,\nu, \star < \mathit{len}\,\star, \star \le \mathit{len}\,\star\}. \end{aligned}$$

`fill` v e	$(0 \leq n) \wedge (\forall j \,.\, 0 \leq j < n \rightarrow v[j] = e)$ v is traversed from left to right and n is the position being currently set to e.
`insert` k v	$(0 \leq m \leq k \leq len\ v) \wedge (\forall j_1, j_2 \,.\, 0 \leq j_1 \leq j_2 < m \rightarrow v[j_1] \leq v[j_2])$ $\wedge\ (\forall j_1, j_2 \,.\, m \leq j_1 \leq j_2 \leq k \rightarrow v[j_1] \leq v[j_2])$ $\wedge\ (\forall j_1, j_2 \,.\, 0 \leq j_1 < m < j_2 \leq k \rightarrow v[j_1] \leq v[j_2])$ Assuming that $v[0..k)$ is sorted, it inserts $v[k]$ in it by successively swapping it with the element on its left. m is the current position of the element being moved.
`insert_sort` v	$(0 \leq k \leq len\ v) \wedge (\forall j_1, j_2 \,.\, 0 \leq j_1 \leq j_2 < k \rightarrow v[j_1] \leq v[j_2])$ k is the index of the element being inserted in the subvector $v[0..k)$, being the latter already sorted.
`linsearch` e v	$(0 \leq k \leq len\ v) \wedge (\forall j \,.\, 0 \leq j < k \rightarrow v[j] \neq e)$ v is traversed from left to right, and k is the index of the element currently visited.
`binsearch_simple` e v	$(0 \leq init \leq end \leq len\ v) \wedge (\forall j_1, j_2 \,.\, 0 \leq j_1 \leq j_2 < len\ v \rightarrow v[j_1] \leq v[j_2])$ $\wedge(\forall j \,.\, 0 \leq j < init \rightarrow v[j] < e) \wedge (\forall j \,.\, end \leq j < len\ v \rightarrow e \leq v[j])$ The element e is searched for in the subvector $v[init..end)$.
`binsearch` e v	$(0 \leq init \leq end \leq len\ v) \wedge (\forall j_1, j_2 \,.\, 0 \leq j_1 \leq j_2 < len\ v \rightarrow v[j_1] \leq v[j_2])$ $\wedge(\forall j \,.\, 0 \leq j < init \rightarrow v[j] \neq e) \wedge (\forall j \,.\, end \leq j < len\ v \rightarrow v[j] \neq e)$ The element e is searched for in the subvector $v[init..end)$.
`partition` v	$(0 \leq p < l \leq len\ v) \wedge (\forall j_1, j_2 \,.\, 0 \leq j_1 < p \wedge j_2 = p \rightarrow v[j_1] \leq v[j_2])$ $\wedge(\forall j_1, j_2 \,.\, j_1 = p \wedge l \leq j_2 < len\ v \rightarrow v[j_1] \leq v[j_2])$ p is the index of the pivot, which is initially 0 and moves from left to right, whereas l starts at the end of the vector and moves towards the beginning.
`quicksort` v	$(0 \leq init \leq end \leq len\ v)$ $\wedge(\forall j_1, j_2 \,.\, 0 \leq j_1 < init \leq j_2 < end \rightarrow v[j_1] \leq v[j_2])$ $\wedge(\forall j_1, j_2 \,.\, init \leq j_1 < end \leq j_2 < len\ v \rightarrow v[j_1] \leq v[j_2])$ $v[init..end)$ is the part of the vector being sorted in the current recursive call.
`minimum` *init end* v	$(0 \leq init \leq min < k \leq end \leq len\ v)$ $\wedge(\forall j_1, j_2 \,.\, j_1 = min \wedge init \leq j_2 < k \rightarrow v[j_1] \leq v[j_2])$ The index k traverses the subvector $v[init..end)$, with *min* being the index of the minimum found so far.
`sel_sort_simple` v	$(0 \leq k \leq len\ v) \wedge (\forall j_1, j_2 \,.\, 0 \leq j_1 < k \leq j_2 < len\ v \rightarrow v[j_1] \leq v[j_2])$ $\wedge(\forall j_1, j_2 \,.\, 0 \leq j_1 \leq j_2 < k \rightarrow v[j_1] \leq v[j_2])$ The minimum element of the subvector starting at k is swapped with $v[k]$, where k goes from left to right.
`sel_sort_full` v	The combination of the invariants of `minimum` and `sel_sort_simple`
`dutch_flag` v	$(0 \leq k \leq l \leq m \leq len\ v) \wedge (\forall j \,.\, 0 \leq j < len\ v \rightarrow 0 \leq v[j] \leq 2)$ $\wedge(\forall j \,.\, 0 \leq j < k \rightarrow v[j] = 0) \wedge (\forall j \,.\, l \leq j < m \rightarrow v[j] = 1)$ $\wedge(\forall j \,.\, m \leq j < len\ v \rightarrow v[j] = 2)$ k, l and m are the indices that delimit the classified segments of v. The subvector $v[k..l)$ is the segment of the vector to be traversed in order to classify its elements.

Fig. 9. Invariants inferred for each example.

The specific variant being used depends on the example. Regarding sets $\mathbb{Q}_I$ and $\mathbb{Q}_{II}$, the qualifiers $0 \leq \#$, $\# < len\,v$, $0 \leq \#_1$, $\#_1 < len\,v$, $0 \leq \#_2$, and $\#_2 < len\,v$ are automatically introduced by the tool, so the user does not have to provide them. We have found that, in some cases, the instantiated sets $\mathbb{Q}^\star$, $\mathbb{Q}^\star_E$, etc. may contain qualifier instances which are known not to occur in any valid inferred refinement. For example, if our `lin_search` algorithm looks for a given integer x in an array of integers, the $\mathbb{Q}^\star_I$ set may contain qualifier instances such as $\# \leq x$, which are not expected in this set since x does not denote an index in the array. This is why our tool allows the user to manually remove some of these instances. By doing so, we obtain the results shown under the header **Pruned**. These show that hints provided by the user may lead to substantially better execution times.

	#G	#NG	#κ	#μ	Typecheck			Full			Pruned		
					#S	#F	Time	#S	#F	Time	#S	#F	Time
fill	22	13	3	4	9	9	9	26	53	1192	20	26	784
insert	27	17	3	4	17	17	35	52	747	14581	42	103	197
insert_sort	19	11	1	4	10	10	14	20	36	61	19	33	53
linsearch	26	16	4	4	14	14	13	35	115	127	32	47	53
binsearch_simple	37	25	6	4	21	21	26	72	1347	2865	61	118	238
binsearch	41	27	6	4	23	23	27	76	3551	21484	53	70	161
partition	39	26	4	4	26	26	34	87	556	3819	68	111	206
quicksort	26	17	2	4	16	16	45	42	507	2924	28	203	749
minimum	35	23	6	4	23	23	29	85	3343	10742	42	66	121
sel_sort_simple	26	16	1	4	15	15	28	47	114	683	44	82	525
sel_sort_full	54	30	4	6	29	29	52	129	1517	33386	95	174	2932
dutch_flag	57	41	7	4	41	41	56	119	1052	7149	108	262	1143

#G	number of goals	#S	number of steps
#NG	number of nontrivial goals	#F	number of formulas solved
#κ	number of κ template variables	Time	execution time, in milliseconds
#μ	number of μ template variables		

Results run under 2.3GHz Intel Core i5-6200U CPU running Fedora Workstation 28 and Z3 v4.6.0

Fig. 10. Execution statistics for the examples involving arrays.

With respect to running times, we believe that there is room for improvement in some cases in which the user does not manually remove instantiated qualifiers. We are currently studying the use of triggers in Z3 to reduce the search space needed for solving a formula. Still, 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.

11 RELATED WORK

The closest related work is about liquid types. This has been already reviewed in Section 2, and we have explained its limitations regarding universally quantified formulas. Let us extend our discussion to two recent works on this framework.

Vazou et al. [34] introduced a notion of bounded quantification for refinement types. Bounds are translated into function types, representing Horn constraints. In this way, decidability is preserved, and the abstract interpretation of liquid typing of Vazou et al. [35] can be reused to infer concrete refinements. The applications of their proposal rely on the definition of particular data types, over which some constraints (bounds) are imposed. Arrays—where arbitrary refinements could be undecidable—are not considered. They impose the restriction that subtyping constraints only have implications inside the refinements corresponding to supertypes (i.e., T_2 in a relation $T_1 <: T_2$), whereas our inference system allows subtyping between array refinements, then implications occur in both sub- and supertypes. In this sense, our inference system supports a wider handling of implications.

A new technique called *fusion* was recently proposed by Cosman and Jhala [7] to infer liquid types in situations where the conventional framework was unable to succeed, or it succeeded with a much higher computational cost. The new situations include polymorphism and higher-order functions. Their constraint generation algorithm is somewhat related to ours in that they introduce existential quantifications in **let** expressions and disjunctions in situations similar to our **case** expressions.

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 Ball et al. [3] and Flanagan and Qadeer [13]. 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 Flanagan and Qadeer [13], 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 which are 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, some of them involving arrays. The main drawback of the approach, when compared with the one presented here, is that a number of annotations given by the programmer—in some examples, up to 15—are needed in each loop.

Gulwani et al. [15] proposed 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 F_j, and all e_j are formulas belonging to nonquantified 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 to convert nonquantified 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.

Srivastava and Gulwani [29] proposed a system where 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 of 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) to deal with undecidable quantified formulas. Additionally, they need to give the system 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.

The F* system [33] allows full dependent typing via SMT solvers using a higher-order universally quantified logic. Regarding expressiveness, F * has the advantage of processing higher-order languages, but unlike in our system, the generated constraints may fall outside the SMT decidable theories. This makes the type system undecidable, so in practice they have a dependency on the solver's unpredictable quantifier instantiation heuristics. The first versions of F* required a heavy annotation burden on the programmer, as predicates had to be explicitly instantiated. Although the

current version of F* [32] may require fewer annotations, the advantages of our approach related to verifying array properties are, on the one hand, that it ensures decidable checking and templates are automatically instantiated, and, on the other hand, that it requires minimal programmer annotations.

A last group of related works is the temporal sequence [8, 14, 18], 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 are 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. In addition, they would be forced to change the abstract domain each time they wish to infer a different property. All reported examples can be dealt with by our approach, and they admit that, at present, they cannot infer *quicksort*.

12 CONCLUSION

We have presented an extension of the liquid type approach to universally quantified formulas about arrays. Arrays are nonrecursive data structures and cannot be dealt with by using the recursive refinements introduced in Kawaguchi et al. [20]. Additionally, arrays are normally updated in-place and so used in imperative languages, whereas 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 article try to cover properties satisfied for all 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, thus liberating the programmer from most of this task.

We believe that other general refinements for arrays could be defined 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] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. 2000. The KeY approach: Integrating object oriented design and formal verification. In *JELIA 2000: Logics in Artificial Intelligence*. Lecture Notes in Computer Science, Vol. 1919. Springer, 21–36. DOI: https://doi.org/10.1007/3-540-40006-0_3

[2] Alexander Bakst and Ranjit Jhala. 2016. Predicate abstraction for linked data structures. In *VMCAI 2016: Verification, Model Checking, and Abstract Interpretation*. Lecture Notes in Computer Science, Vol. 9583. Springer, 65–84. DOI: https://doi.org/10.1007/978-3-662-49122-5_3

[3] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. 2001. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. 203–213.

[4] Michael Barnett, Robert DeLine, Manuel Fähndrich, Bart Jacobs, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter. 2005. The Spec# programming system: Challenges and directions. In *VSTTE 2005: Verified Software: Theories, Tools, Experiments*. Lecture Notes in Computer Science, Vol. 4171. Springer, 144–152. DOI: https://doi.org/10.1007/978-3-540-69149-5_16

[5] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2006. What's decidable about arrays? In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*. Springer, 427–442.
[6] Ravi Chugh, David Herman, and Ranjit Jhala. 2012. Dependent types for JavaScript. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. 587–606. DOI: https://doi.org/10.1145/2384616.2384659
[7] Benjamin Cosman and Ranjit Jhala. 2017. Local refinement typing. In *Proceedings of the ACM Conference on Programming Languages*, Vol. 1, Issue ICFP. Article 26, 27 pages. DOI: https://doi.org/10.1145/3110270
[8] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'11)*. 105–118. DOI: https://doi.org/10.1145/1926385.1926399
[9] E. W. Dijkstra. 1976. *A Discipline of Programming*. Prentice Hall.
[10] Matthias Felleisen and Philippa Gardner (Eds.). 2013. *Programming Languages and Systems: 22nd European Symposium on Programming (ESOP'13)*. Lecture Notes in Computer Science, Vol. 7792. Springer.
[11] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3—Where programs meet provers. In *ESOP 2013: Programming Languages and Systems*. Lecture Notes in Computer Science, Vol. 7792. Springer, 125–128.
[12] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*. ACM, New York, NY, 234–245. DOI: https://doi.org/10.1145/512529.512558
[13] Cormac Flanagan and Shaz Qadeer. 2002. Predicate abstraction for software verification. In *Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*. ACM, New York, NY, 191–202.
[14] Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. 2005. A framework for numeric analysis of array operations. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*. 338–350. DOI: https://doi.org/10.1145/1040305.1040333
[15] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. 2008. Lifting abstract interpreters to quantified logical domains. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*. ACM, New York, NY, 235–246.
[16] Rajiv Gupta and Saman P. Amarasinghe (Eds.). 2008. *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI'08)*. ACM, New York, NY.
[17] Peter Habermehl, Radu Iosif, and Tomás Vojnar. 2008. What else is decidable about integer arrays? In *FoSSaCS 2008: Foundations of Software Science and Computational Structures*. Lecture Notes in Computer Science, Vol. 4962. Springer, 474–489. DOI: https://doi.org/10.1007/978-3-540-78499-9-33
[18] Nicolas Halbwachs and Mathias Péron. 2008. Discovering properties about arrays in simple programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. 339–348. DOI: https://doi.org/10.1145/1375581.1375623
[19] Michael Hind and Amer Diwan (Eds.). 2009. *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. ACM, New York, NY.
[20] Ming Kawaguchi, Patrick Maxim Rondon, and Ranjit Jhala. 2009. Type-based data structure verification. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. 304–315.
[21] K. Rustan M. Leino. 2012. Developing verified programs with Dafny. In *Proceedings of the 2012 ACM Conference on High Integrity Language Technology (HILT'12)*. ACM, New York, NY, 9–10.
[22] Manuel Montenegro, Susana Nieva, Ricardo Peña, and Clara Segura. 2017. Liquid types for array invariant synthesis. In *Proceedings of the International Symposium on Automated Technology for Verification and Analysis (ATVA'17)*. 289–306.
[23] Manuel Montenegro, Ricardo Peña, and Jaime Sánchez-Hernández. 2015. A generic intermediate representation for verification condition generation. In *LOPSTR 2015: Logic-Based Program Synthesis and Transformation*. Lecture Notes in Computer Science, Vol. 9527. Springer, 227–243.
[24] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *TACAS 2008: Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, Vol. 4963. Springer, 337–340.
[25] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2009. Generalized, efficient array decision procedures. In *Proceedings of the 2009 Conference on Formal Methods in Computer-Aided Design (FMCAD'09)*. IEEE, Los Alamitos, CA, 45–52.
[26] John C. Reynolds. 1998. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation* 11, 4 (1998), 363–397. DOI: https://doi.org/10.1023/A:1010027404223
[27] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. 159–169.

[28] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2010. Low-level liquid types. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)*. ACM, New York, NY, 131–144.

[29] Saurabh Srivastava and Sumit Gulwani. 2009. Program verification using templates over predicate abstraction. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. 223–234.

[30] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. 2001. A decision procedure for an extensional theory of arrays. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*. IEEE, Los Alamitos, CA, 29–37.

[31] Norihisa Suzuki and David Jefferson. 1980. Verification decidability of Presburger array programs. *Journal of the ACM* 27, 1 (1980), 191–205.

[32] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, et al. 2016. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*. ACM, New York, NY, 256–270.

[33] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the Dijkstra monad. In *Proceedings of the 34th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. 387–398.

[34] Niki Vazou, Alexander Bakst, and Ranjit Jhala. 2015. Bounded refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP'15)*. 48–61.

[35] Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. 2013. Abstract refinement types. In *ESOP 2013: Programming Languages and Systems*. Lecture Notes in Computer Science, Vol. 7792. Springer, 209–228.

[36] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014. LiquidHaskell: Experience with refinement types in the real world. In *Proceedings of the ACM SIGPLAN Symposium on Haskell (Haskell'14)*. 39–51.

[37] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP'14)*. 269–282.

[38] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement types for TypeScript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)*. DOI: https://doi.org/10.1145/2908080.2908110 arXiv:arXiv:1604.02480v1

Received December 2018; revised June 2019; accepted September 2019