Dependent Array Type Inference from Tests

He Zhu[1], Aditya V. Nori[2], and Suresh Jagannathan[1]

[1] Purdue University
[2] Microsoft Research
{zhu,suresh}@cs.purdue.edu, adityan@microsoft.com

Abstract. We present a type-based program analysis capable of inferring expressive invariants over array programs. Our system combines dependent types with two additional key elements. First, we associate dependent types with effects and precisely track effectful array updates, yielding a sound flow-sensitive dependent type system that can capture invariants associated with side-effecting array programs. Second, without imposing an annotation burden for quantified invariants on array indices, we automatically infer useful array invariants by initially guessing very coarse invariant templates, using test suites to exercise the functionality of the program to faithfully instantiate these templates with more precise (likely) invariants. These inferred invariants are subsequently encoded as dependent types for validation. Experimental results demonstrate the utility of our approach, with respect to both expressivity of the invariants inferred, and the time necessary to converge to a result.

1 Introduction

A program invariant describes valid behaviors a program is expected to produce, and can often be derived by a fixpoint construction over an over-approximation of program states [4]. However, applying such a strategy to discover useful properties of values stored in unbounded collections of heap cells is nontrivial.

Dependent type systems [22,17] have been proven to be successful in automated verification of complex invariants for data structures, even when there are an unbounded number of heap locations under consideration [23]. In these systems, decidability is achieved, however, at the loss of flow-sensitivity, i.e., a strong update to a concrete location (e.g. a single array cell) must be subsumed by the whole data structure (e.g. the whole array). As a result, it is not obvious how existing dependent type systems can be extended to verify useful functional properties (e.g. *a sorting procedure will sort only a part of the elements in an input array*) that are beyond the scope of global invariants (e.g. a general memory safety properties).

In this paper, we address these issues by introducing a new dependent type system for array programs that can discharge complex flow-sensitive array invariants naturally characterized in terms of quantifiers on array indices. The dependent type system is effectful because it can tolerate side-effecting array updates. Built on top of a standard type system, our system refines basic type

D. D'Souza et al. (Eds.): VMCAI 2015, LNCS 8931, pp. 412–430, 2015.
© Springer-Verlag Berlin Heidelberg 2015

with a type refinement predicate that captures precise properties of the values defined by the type. Importantly, to verify flow-sensitive invariants, type refinements may be quantified. This is crucial, as strong updates in a procedure may only update a subset of the array.

Rather than requiring users to annotate types with refinements, our approach attempts to learn quantified array invariants. Although significant advances have been made in recent years to allow useful array invariants to be inferred automatically, prior approaches either (a) require a predefined fixed or parameterized partition of array indices [10,14], (b) entail sophisticated reasoning over quantified abstract domains [12], or (c) rely on powerful theorem provers to provide predicates [1,2,15,18,20,24] (as *interpolants*) that may hold on the program in general. We consider the problem from a different perspective, based on the expectation that useful array invariants should be observable from test runs. By summarizing or generalizing the properties that hold in all such runs, we can construct a set of candidates or likely invariants. We then lift these presumed invariants to our dependent type system for validation.

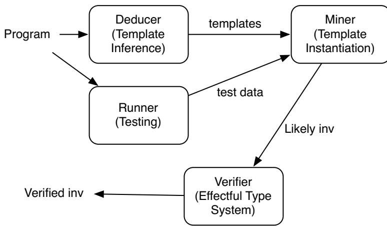

Fig. 1. Framework

The framework of our approach is outlined in Figure 1 : (I) a Deducer initially guesses coarse templates for the invariants; (II) a Runner then runs the subject program through simple random test suites; (III) a Miner generates a constraint system by substituting the variables in the template with concrete values from test runs; (IV) a Verifier validates the likely dependent types derived from the solution of the constraint system.

Our technique is *compositional*—invariants are inferred for each procedure without the need for additional context information about callers and callees. It is *lightweight* because the constraint system from which program invariants are inferred is obtained from concrete program states and not limited by a specific abstract domain construction; experimental results also indicate that the approach allows fast convergence from a *likely* to a *provable* invariant. Our paper thus makes the following contributions:

1. We propose a novel data driven algorithm to infer array invariants that leverages observations from test cases to guide inference.
   - Avoiding the high cost of inferring exact array invariants, our approach initially guesses coarse invariant *templates*, at the expense of precision.
   - We train the template with concrete program states collected from test runs to instantiate it to likely invariants, recovering precision.
2. We integrate our technique within a new effectful dependent type system that can be used to automatically validate the correctness of the likely invariants.

2 Overview

We use the inner loop of the classic *insertion-sort* program shown in Figure 2 to illustrate and motivate our approach. Figure 2 visualizes the execution of

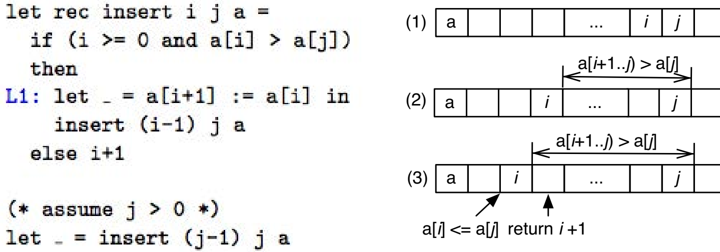

Fig. 2. Inner array insertion-sort program

the recursive function `insert`: (1) initially parameter `i` is set to the index of the left adjacent element of $\mathtt{a[j]}$; (2) function `insert` then accesses the array elements with indices less than `j` iteratively; (3) it terminates when it finds an element that is no greater than $\mathtt{a[j]}$. Our approach can automatically infer a useful dependent type for `insert`, capturing the behaviors described above.

Instead of directly inferring an exact invariant, Deducer (in Figure 1) guesses the invariant's template based on a backward symbolic execution. Assume we infer that the postcondition of `insert` is $\varphi_{\mathtt{post}}$. We focus on the first branch (L1) and unwind the recursion only once, deriving the following precondition,

$$\varphi_{\mathtt{pre}} \equiv \ \exists \mathsf{a}'.\ [\mathsf{a}'/\mathsf{a}]\varphi_{\mathtt{post}} \wedge ((\mathtt{i} >= 0 \wedge \mathsf{a}[\mathtt{i}] > \mathsf{a}[\mathtt{j}] \wedge$$
$$\forall \mathsf{a}_0.((\mathsf{a}_0 = \mathtt{i}+1 \Rightarrow \mathsf{a}'[\mathtt{i}+1] = \mathsf{a}[\mathtt{i}]) \wedge (\mathsf{a}_0 \neq \mathtt{i}+1 \Rightarrow \mathsf{a}'[\mathsf{a}_0] = \mathsf{a}[\mathsf{a}_0])) \wedge ...) \vee ...)$$

where a_0 is a special universal variable and we use a' to refer to a in the state after the update. Note that the precondition provides information about how array elements are manipulated by a procedure. In particular, $\varphi_{\mathtt{pre}}$ reflects the fact that the $(\mathtt{i})^{\mathtt{th}}$ element of `a` is moved to its right position if it is greater than the $(\mathtt{j})^{\mathtt{th}}$ element in `insert`. It is unclear, however, how to generalize this condition, which defines only an under-approximation of the desired invariant.

Nonetheless, it is possible to guess that a general invariant may be in the shape of the predicate $\mathsf{a}[\mathtt{i}] > \mathsf{a}[\mathtt{j}]$ from $\varphi_{\mathtt{pre}}$. Based on the assumption that array invariants are typically universally quantified on array indices, we infer the following form for a valid invariant:

$$\forall \mathsf{a}_0.\ 0 \leq \mathsf{a}_0 < \chi_1(\bar{x}) \Rightarrow \mathsf{a}[\mathsf{a}_0; \chi_2(\bar{x})] > \mathsf{a}[\chi_3(\bar{x})]\}$$

where $\bar{x} = \{\mathtt{i}, \mathtt{j}\}$ is used to denote all the scalar parameters of `insert` and $\chi(\bar{x}) = \bar{c} * \bar{x}$ represents a parameterized linear expression over $\bar{x}$ ($\bar{c}$ are unknown coefficients). This predicate abstracts a relation to describe how `insert` (iterating over `i`) might maintain an invariant over array `a`. In this formula, $\mathsf{a}[\mathsf{a}_0; \chi_2(\bar{x})]$ is universally quantified on the special variable a_0 which is bounded by $\chi_1(\bar{x})$.

Obviously, concrete program states collected from test runs must satisfy the guessed invariant template in terms of `insert`'s preconditions (and postconditions). To generate such states, Runner calls the array *insertion-sort* program with a randomly generated array. We dump the input/output values of `insert` as its concrete pre- and post-states. For each pre-state, we substitutes the variables in the template with their values in this concrete state, deriving some constraints. Thus we obtain a linear constraint system over the unknown coefficients. Using an SMT solver, Miner is able to instantiate the template to the following likely invariant (precondition):

$$\forall \mathsf{a}_0.\ 0 \leq \mathsf{a}_0 < \mathtt{j} - \mathtt{i} - 1 \Rightarrow \mathsf{a}[\mathsf{a}_0 + \mathtt{i} + 1] > \mathsf{a}[\mathtt{j}]$$

Note that this likely invariant is obtained by exploiting the local states of `insert` solely. Not all instantiations are real invariants; spurious instantiations coincide with properties exposed by particular test cases but do not hold in general. Verifier validates whether a likely invariant generalizes by encoding the invariant into a dependent type system (covered in Section 4). Dependent type constraints are solved via an abstract interpretation to yield valid types (whose type refinements are a conjunction of the predicates from the likely invariants) for the program. We delay details of how the above invariant can be validated to Example 2.

Applying all these steps (with a similar inference step for deriving the postcondition), we are able to associate the following non-trivial type to `insert`:

$$\begin{aligned}&\mathtt{i} : \mathtt{int} \rightarrow \mathtt{j} : \mathtt{int} \rightarrow \mathsf{a} : \{\mathsf{array} | \forall \nu_0.\ 0 \leq \nu_0 < \mathtt{j} - \mathtt{i} - 1 \Rightarrow \nu[\nu_0 + \mathtt{i} + 1] > \nu[\mathtt{j}]\} \\ &\rightarrow \mathtt{ret} : \mathsf{int}/[\mathsf{a} : \{\mathsf{array} | \forall \nu_0.\ 0 \leq \nu_0 < \mathtt{j} - \mathtt{ret} \Rightarrow \nu[\nu_0 + \mathtt{ret}] > \nu[\mathtt{j}]\}]\end{aligned}$$

where the special variable ν is used to denote the value of term a in its corresponding type refinement predicate (we ignore the dependent type for $\mathtt{i}$ and $\mathtt{j}$ for simplicity). If ν refers to an array, then ν_0 denotes its first subscript. This type specifies that, in `insert`, the array elements in $\mathsf{a}[\mathtt{i} + 1, \cdots, \mathtt{j} - 1]$ are greater than $\mathsf{a}[\mathtt{j}]$; and produces as a side-effect that, the elements in $\mathsf{a}[\mathtt{ret}, \cdots, \mathtt{j} - 1]$ are greater than $\mathsf{a}[\mathtt{j}]$ where `ret` denotes the return value (in Section 7, we discuss how the predicates over a and a' in $\varphi_{\mathtt{pre}}$ are also exploited to deduce a universally existentially quantified ($\forall\exists$) invariant capable of proving preservation property).

3 Language

In the rest of the paper, we focus on single-dimensional arrays for simplicity; our approach can be naturally extended to handle multi-dimensional cases.

We formalize our ideas in the context of a call-by-value variant of the λ-calculus with support for dependent types. The syntax of the language is shown in Figure 3. Typically a is only bound to arrays and x and y are usually bound to both *scalar* variables, drawn from some non-array base type, and arrays. Predicates (p) are Boolean expressions built from a predefined set ($\mathcal{Q}$) of first-order relational operators (functions); the arguments to these operators are restricted to simple expressions - variables, constants, or array expressions and arithmetic

$$x, y \in \mathsf{Var} \quad a \in \mathsf{Arr} \quad c \in \mathsf{Constant} ::= 0, \ldots, \mathsf{true}, \mathsf{false} \quad B \in \mathsf{Base} ::= \mathsf{int} \mid \mathsf{bool} \mid \mathsf{array}$$
$$\tau \in \mathsf{Monotype} ::= B \mid \tau \to \tau \qquad P \in \mathsf{DepType} ::= \{\nu : B \,|\, r\}/T \mid \{x : P \to P\}$$
$$r \in \mathsf{Refinement} ::= \kappa \mid p \quad T \in \mathsf{EffType} ::= (x : \{\nu : B \mid r\}/[\,]); T \mid [\,]$$
$$p \in \mathsf{Predicate} ::= p \;\mathsf{and}\; p \mid p \;\mathsf{or}\; p \mid \mathcal{Q}(s, \ldots, s) \qquad \mathcal{Q} \in \{>=, >, \cdots\}$$
$$s \in \mathsf{SimpleExp} ::= \nu_0 \mid \nu \mid x \mid a \mid c \mid s \;\mathsf{op}\; s \mid a[s] \qquad \mathsf{op} \in \{+, -, \cdots\}$$
$$e ::= s \mid a[s] := s \mid \lambda\, x.\, e \mid \mathsf{if}\; p \;\mathsf{then}\; e \;\mathsf{else}\; e \mid \mathsf{let}\; x = e \;\mathsf{in}\; e \mid e\; x$$

Fig. 3. Syntax

compositions of such expressions; a type refinement (r) is either a type refinement variable (κ) that represents an unknown type refinement or a concrete predicate (p). Instantiation of the type refinement variables to concrete predicates takes place through the type refinement algorithm described in Section 5.

Our language supports a small set of base types (B), monotypes (τ) and dependent types (P) that include dependent base types and dependent function types. A *dependent base type* is written $\{\nu : B|\, r\}/\ T$. B is a base type, such as int or bool, and r is called a *type refinement* that constrains the value defined by the type. Effect type T is a sequence of dependent types binding to side-effecting arrays, conservatively approximating the side-effects an expression may produce. These bindings have no further effect, i.e., effect types are not nested. In the following, we will often omit the declaration of ν or T if it is empty for simplicity. For example, the expression (let _ $= a[x] := 1$ in 0) where x is an integer, has type $\{\{\mathsf{int} \,|\, \nu = 0\}/T_{ex}\}$ where

$$T_{ex} \equiv \{a : \{\mathsf{array}|\forall \nu_0.\ (\nu_0 = x \Rightarrow \nu\ [\nu_0] = 1) \wedge (\nu_0 \neq x \Rightarrow \nu\ [\nu_0] = a\ [\nu_0])\}\}$$

This type reflects that the expression yields 0, but additionally has a side-effect that updates the $x^{\mathtt{th}}$ element of array a to 1. When this effect is merged with the type environment of this expression (detailed in Section 4), the array a inside the type refinement will be modified to refer to the old array before the update.

A *dependent function type* is written $\{x : P_x \to P\}$[1] where the argument x is constrained by the dependent type P_x, and the result type is specified by P. For instance, $\{a : \mathsf{array} \to x : \{\nu : \mathsf{int} \mid \nu > 0\} \to \{\nu\ : \mathsf{int} \mid \nu > x\}/T_{ex}\}$ specifies the function that given a positive integer returns an integer greater than x, that also raises a side-effect captured by T_{ex}.

Unknown type refinements for array type parameters and return value ($\bar{a}$) of a function are instantiations from a *general template* that is created for each array a_i ($a_i \in \bar{a}$ and the syntactic sugar $\bar{a}[a_{i0}; \bar{x}]$ denotes an arbitrary array expression whose array indices are arithmetic compostions from variables in $[|a_{i0}; \bar{x}|]$):

$$I \equiv \forall a_{i0}.0 \leq a_{i0} < \varphi_i(\bar{x}) \Rightarrow (\bigwedge_{a_j \in \{\bar{a}/a_i\}} \varphi_j(\bar{x}) \leq \psi_j(\bar{x}) < \varphi'_j(\bar{x})) \Rightarrow \mathcal{Q}(\bar{a}[a_{i0}; \bar{x}], \bar{x})$$

[1] Although side-effects can be associated to closures (function typed), we disallow it in the paper to keep simplicity but implement it in our tool (Section 8).

where a_{i0} represents the single subscript of array a_i, $\varphi_i(\bar{x})$ is an arithmetic expression over non-array base type parameters, serving as an upper bound for a_{i0}, and $\mathcal{Q}(\bar{a}[a_{i0};\bar{x}],\bar{x})$ is a predicate (drawn from the language of linear arithmetic and uninterpreted functions) over array expressions $\bar{a}[a_{i0};\bar{x}]$ and all scalars parameters $\bar{x}$ of the function. The second implication condition naturally bounds the array index for arrays other than a_i. To translate an instantiation of the template into a type refinement, we simply perform the substitution $[\nu/a_i][\nu_0/a_{i0}]I$, which can be embedded into the dependent type of a_i (e.g., see the type of `insert` in Section 2).

4 Dependent Type System for Arrays

Figure 4 defines dependent type inference rules; these rules are adapted from [22], generalized to deal with array update effects. Syntactically, $\Gamma \vdash e : P$ states that expression e has dependent type P under type environment Γ, which is a sequence of type bindings $x : P$ and guard predicates p. The former are standard; the latter capture path-sensitivity of program branches, following [22].

As in [22], the built-in units of function such as $+, -$ are encoded as constants which have predefined dependent types that capture their semantics. In this paper we are particularly interested in array updates as side-effects and we encode array update function $a[x] := y$ as *primitive constant.* Its type is given as:

$$\Gamma \vdash (a[x] := y) : \{a : \mathsf{array} \rightarrow x : \mathsf{int} \rightarrow y : \mathsf{int} \rightarrow$$
$$\mathsf{unit}/[a : \{\mathsf{array} | \forall \nu_0.\ (\nu_0 = x \Rightarrow \nu[\nu_0] = y) \wedge (\nu_0 \neq x \Rightarrow \nu[\nu_0] = a[\nu_0])\}]\}$$

Before describing the key components of the type system, we introduce some auxiliary functions. We define $\texttt{mod}\,(\Gamma, e)$ as the function that returns all the arrays bound in Γ that have array updates inside e. Firstly, $\texttt{mod}\,(\Gamma, a[x] := y) = a$. The other cases are simply recursively defined. Additionally, function $\texttt{Eff}\,(P)$ returns the effect of dependent type P (a function definition given as a lambda expression does not produce side-effects). Function $\texttt{Ty}\,(P)$ erases the effects for base dependent types. We use $dom(T)$ to return the keys of the bindings of effect T.

$$\texttt{Eff}\,(\{\nu : B \mid p\}\ /\ T) = T \quad \texttt{Eff}\,(\{x : P \rightarrow P\}) = [\,]$$
$$\texttt{Ty}\,(\{\nu : B \mid p\}\ /\ T) = \{\nu : B \mid p\}\ /\ [\,] \quad \texttt{Ty}\,(\{x : P \rightarrow P\}) = \{x : P \rightarrow P\}$$

Well-Formedness Judgement. These rules are of the form $\Gamma @ \texttt{modset} \vdash P$, and check if dependent type P is well defined under type environment Γ, which is extended with a set of arrays (`modset`) that may be updated by this type's underlying expression. Rule $\mathsf{WF_Base}$ firstly checks that the type refinement p of a dependent base type does not refer to program variables that escape from its type environment Γ, i.e, p is a well defined predicate. Secondly we enforce that all the side-effects raised by an expression must be captured by its type by checking that the keys of the binding in its effect T must contain `modset` and that T

$$\frac{\Gamma;\nu : B \vdash p : \mathsf{bool} \quad \texttt{modset} \subseteq dom(T) \quad \Gamma \vdash T}{\Gamma @ \texttt{modset} \vdash \{\nu : B | p\}/T} \text{ WF_Base}$$

$$\frac{\Gamma; x : P_x @ \texttt{modset} \vdash P}{\Gamma @ \texttt{modset} \vdash x : P_x \to P} \text{ WF_Fun} \qquad \frac{\forall \{a : P\} \in T.\ \Gamma @ [\,] \vdash P}{\Gamma \vdash T} \text{ WF_Eff}$$

$$\frac{\Gamma; x : P_x \vdash e : P_e \quad \Gamma; x : P_x \vdash P_e <: P}{\Gamma \vdash \lambda x.e : \{x : P_x \to P\}} \text{ Fun}$$

$$\frac{\Gamma \vdash e : \{x : P_x \to P\} \quad \Gamma \vdash y : P'_x \quad \Gamma \vdash P'_x <: P_x}{\Gamma \vdash e\ y : [y/x]P} \text{ App}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : P' \quad \Gamma @ \{\texttt{mod}\,(\Gamma, e_1) \cup \texttt{mod}\,(\Gamma, e_2)\} \vdash P \\ \theta = \{[\widetilde{y}/y] \mid y \in dom(\texttt{Eff}\,(P'))\} \\ \theta(\Gamma; x : \texttt{Ty}\,(P'); \texttt{Eff}\,(P'); \forall y \in dom(\texttt{Eff}\,(P')).\widetilde{y} : \Gamma(y)) \vdash e_2 : P\end{array}}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : P} \text{ Let}$$

$$\frac{\Gamma \vdash p : \mathsf{bool} \quad \Gamma; p \vdash e_2 : P \quad \Gamma; \neg p \vdash e_3 : P \quad \Gamma @ \{\texttt{mod}\,(\Gamma, e_1) \cup \texttt{mod}\,(\Gamma, e_2)\} \vdash P}{\Gamma \vdash \mathsf{if}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : P} \text{ If}$$

$$\frac{\langle \Gamma \rangle \wedge \langle \nu : r_1 \rangle \Rightarrow \langle \nu : r_2 \rangle \quad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash \{\nu : B | r_1\}/T_1 <: \{\nu : B | r_2\}/T_2} \text{ Sub_Base}$$

$$\frac{\Gamma \vdash P'_x <: P_x \quad \Gamma; x : P'_x \vdash P <: P'}{\Gamma \vdash \{x : P_x \to P\} <: \{x : P'_x \to P'\}} \text{ Sub_Fun}$$

$$\frac{dom(T_1) \subseteq dom(T_2) \quad \forall \{a : P\} \in T_2.\ \Gamma; T_1 \vdash a : P}{\Gamma \vdash T_1 <: T_2} \text{ Sub_Eff}$$

Fig. 4. Typing rules

is well-formed. Rule WF_Fun and WF_Eff define well-formedness conditions for functions and effects, resp.

Type Judgements. The typing rules state how an expression e can be dependently typed. Rules `Fun` and `App` are standard. As in [22], our approach requires the need for *pending substitutions* because the dependent type of a function application is derived by substituting all the formal argument x in the output by the actual y. It is formally defined as $\theta ::= [y/x]; \theta \mid [\,]$. Pending substitution for base dependent type is defined as $\theta(\{\nu : B \mid p\}\ /\ T) = \{\nu : B \mid \theta p\}\ /\ \theta T$. where $\theta T = \{\theta x : \theta P \mid x : P \in T\}$. Note that we push pending substitution to effects. Pending substitution for functional types are trivially recursively defined.

In rule `Let`, the well-formedness condition checks that the effect of the entire `Let`-expression subsumes all the effects produced by e_1 and e_2. When typing expression e_2, we require that the side-effects of e_1 must refresh e_2's typing environment for soundness ($\texttt{Ty}\,(P')$ resets P''s effects to empty). Note that $\Gamma; T$ means Γ is merged with the effects T: for each binding $x : P \in T$, we substitute the original binding to x in Γ with P. However, the original binding is not simply discarded. If y is an array which could be updated by e_1 (witnessed by its type's effect), its

original dependent type recorded in Γ is re-associated to a $\widetilde{y}$. Intuitively, we use $\widetilde{\mathtt{y}}$ to refer to y in the state before the update. This relieves typing burden because now the array y after the update can refer to its original version $\widetilde{y}$ for the elements that are not changed. To retain soundness, the appearance of y in the type refinement predicates of the dependent types bound to Γ must be substituted with $\widetilde{y}$. This is achieved by performing environment substitution θ as defined in the rule. Formally, $\theta\Gamma = \{x : \theta P | x : P \in \Gamma\}$. Rule `If` is standard except we require that all the effects made by its subexpressions must be subsumed by the effect of the entire `If`-expression.

Subtype Judgement. This class of rules checks at each call site that the actual arguments satisfy the precondition of the called function and verify, at each definition site, that the return value establishes the desired postcondition. Rule `Sub_Base` checks whether a dependent type subtypes another dependent type for based typed expression. The premise check requires the conjunction of environment formula $\langle\Gamma\rangle$ and $\langle\nu : r_1\rangle$ implies $\langle\nu : r_2\rangle$. Our encoding of $\langle\Gamma\rangle$ (or $\langle\nu : r_1\rangle$ and $\langle\nu : r_2\rangle$) as a first order logic formula is inspired by [22]:

$$\bigwedge\{p \mid p \in \Gamma\} \wedge \bigwedge\{[x/\nu][x_0/\nu_0]r \mid x : \{\nu : B \mid r\}/T \in \Gamma\} \qquad (\varrho)$$

For example, consider typing an expression e when it is enclosed in a statement $\mathtt{let}\ _ = A[x] := y\ \mathtt{in}\ e$. According to rule App and Let, the type environment of e is $[\widetilde{A}/A](A : \{\forall\nu_0.\ (\nu_0 = x \Rightarrow \nu[\nu_0] = y) \wedge (\nu_0 \neq x \Rightarrow \nu[\nu_0] = A[\nu_0])\}; \widetilde{A} : \{\Gamma(a)\})$ where $\widetilde{A}$ is a copy of the original array. The encoding for A according to (ϱ) is

$$A : [A/\nu][A_0/\nu_0]\{\mathsf{array}|\forall\nu_0.\ (\nu_0 = x \Rightarrow \nu[\nu_0] = y) \wedge (\nu_0 \neq x \Rightarrow \nu[\nu_0] = \widetilde{A}[\nu_0])\}$$

This illustrates the fact the array update application $A[x] := y$ produces as a side-effect, an update to the $x^{\mathtt{th}}$ element of the array; the other elements of the array are not changed and hence refer to the original array which is remembered by the type system as $\widetilde{A}$. This kind of embedding aims to strengthen the antecedent of the implication and is conservative [22]. Rule `Sub_Fun` is again standard, and rule `Sub_Eff` checks wether two effects are subtyped. Arrays bound in T_1 should be subsumed by that in T_2; subtype checking is reduced to dependent type checking $\Gamma; T_1 \vdash a : P$ for each array a bound in T_2.

Features. Our analysis has several notable characteristics. First, by piggybacking type refinements (which are inferred using the techniques described in Section 5) on top of standard type inference, we can use abstract interpretation (in the form of liquid type inference [22]) to verify array properties. Thus, our technique reduces static analysis for arrays to a Boolean fixpoint computation. Unlike a theorem prover based approach which must generate suitable predicates on the fly, our type system ensures termination. A detailed comparison with related work is summarized in Section 9. Our type system maintains precision in the face of array updates by using case splits on array indices; to avoid case explosion, we exploit the natural function summarization that is expressed by a function type signature.

5 Array Type Refinements Inference

In this section we show how type refinements in dependent types can be automatically inferred from tests. Specifically, we infer a dependent function type for each function by inferring the function's precondition and postcondition. Our type refinement inference is compositional, i.e., we generate likely array invariants for a function, independent of its caller and callee.

5.1 Template Generation

As we have discussed in Section 2, our inference starts from a symbolic analysis analogous to weakest precondition generation wp. We guess invariant templates for each function according to its wp. Our wp algorithm simply pushes postconditions backward, substituting terms for values in the presumed postcondition based on the structure of the predicate used to generate the precondition.

$$
\begin{aligned}
\mathsf{wp}(e, \phi) = \ & \mathsf{case}\ e\ \mathsf{of} \\
& \mid \texttt{if}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightarrow (p \wedge \mathsf{wp}(e_1,\ \phi) \vee (\neg p \wedge \mathsf{wp}(e_2,\ \phi))) \\
& \mid \texttt{let}\ x\ =\ e_1\ \mathsf{in}\ e_2 \rightarrow \mathsf{wp}(e_1,\ [\nu/x]\mathsf{wp}(e_2, \phi))) \\
& \mid (\lambda x.e)\ y \rightarrow [y/x]\mathsf{wp}(e,\ \phi) \\
& \mid e\ y \rightarrow \mathsf{wp}((\lambda x.e')\ y,\ \phi) \quad \text{(where } e \text{ can be deferred to } \lambda x.e'\text{)} \\
& \mid a[s_1] := s_2 \rightarrow \exists a'.\ \phi[a'/a] \wedge \\
& \qquad \{\forall a_0.\ (a_0 = s_1 \Rightarrow a'[s_1]\ \{\geq,\ \leq\}\ s_2) \wedge (a_0 \neq s_1 \Rightarrow a'[a_0] = a[a_0])\} \\
& \mid s \rightarrow [s/\nu]\phi
\end{aligned}
$$

To invoke wp, we initially supply true as the ϕ argument. Notably, during the process, it is refined to capture all the array reads and updates through `if` cases and array update cases of the rules. However this wp definition does not terminate for recursive functions. Since our aim is to guess coarse templates of array invariants, we simply bound the number of times a recursive function call is unrolled (at most 2 in our experiments). When wp has just traversed a function f, our system remembers the immediate result as f's weakest precondition and can later retrieve it using wp(f). As stated in Section 2, wp reflects under-approximative information about how array elements are manipulated by a procedure. Our inference principle is that, while information implied in wp is under-approximate, if encoded into a template, can nonetheless be potentially generalized by running tests for instantiation.

We supply the weakest precondition `wp`(f) of a function f to our template generation algorithm, `guessT` in Figure 5, which outputs a set of invariant templates for f. We define `scalar`(f) as the *scalar* parameters and return value of function f, and `scalar`(p) as the scalar variables used in a predicate p. Similarly, `scalar`(s) returns the scalar variables used in a simple expression s. For readability, we define that notation $s \not\equiv s_i$ is true if and only if `scalar`(s) $\cap$ `scalar`(s_i) $= \emptyset$.

```
let guessT f wp =
 foreach atomic predicate (p as Q(ā[_]; x̄)) in wp
L: foreach a_i[s_i] in p
     let a_i0 = create_var "a_i0" in
     let b = χ(scalar(f)) in
     output "
     ∀a_i0. 0 ≤ a_i0 < b ⇒
       {⋀_{a[s]∈p, a≠a_i ∧ s≢s_i} χ(scalar(f)/scalar(p)) ≤ χ(scalar(f) ∪ scalar(p))
                                   < χ(scalar(f)/scalar(p))  } ⇒
         Q(a_i[a_i0; scalar(f)];  (a[χ(a_i0; scalar(f))] | a[s]∈p∧a≠a_i ∧ ¬(s ≢ s_i));
                                  (a[χ(scalar(f))] | a[s]∈p∧a≠a_i ∧ (s ≢ s_i));  x̄)"
```

Fig. 5. Array Invariant Template Inference

In `guessT`, our algorithm traverses `wp` and generates invariant templates (defined in Section 3) for each of its simple relational predicates $\mathcal{Q}(\bar{a}[_];\bar{x})$), denoted as p, if it ranges over some array expression $\bar{a}[_]$. Inside the loop at location L, for each array expression $a_i[s_i]$, we create a universal variable a_{i0} and its upper bound as $\chi(\texttt{scalar}(f))$ for array a_i. Suppose $\bar{v}$ is a set of scalar variables. $\chi(\bar{v})$ is an arithmetic template over $\bar{v}$: $c_1 * v_1 + \cdots + c_n * v_n + c_0$, with coefficients $c_i (0 \leq i < n)$ as integer variables. Arrays other than a_i are also required to be accordingly bounded (intuitively corresponding to an array index partition) as the algorithm shows (the set minus operation used in these lower- and upper-bounds simply helps avoid considering uninteresting invariants).

Our algorithm then infers appropriate index templates over f's scalar variables for each array expression $a[s]$ in p, while it maintains the main shape of p. If an array expression $a[s]$ is exactly $a_i[s_i]$ or it happens to share some scalar variables with $a_i[s_i]$ in their subscripts, we create its index template, applying χ over the special universal variable a_{i_0} and the scalar parameters defined in $\texttt{scalar}(f)$. Otherwise, s and s_i have disjoint scalar variables; the index of a is transformed to an index template over $\texttt{scalar}(f)$ only.

Example 1. Consider the `insert` procedure in Figure 2. The weakest precondition of `insert`, `wp`(`insert`), defines a simple predicate: $\mathtt{p}_1 \equiv \mathtt{a}[\mathtt{i}] > \mathtt{a}[\mathtt{j}]$. Inside the loop at L, assume array expression $\mathtt{a}[\mathtt{i}]$ is picked. To infer a precondition, the type signature of `insert` reveals that $\texttt{scalar}(\texttt{insert}) = \{\mathtt{i}, \mathtt{j}\}$. So $\mathtt{p}_1$ is parameterized to $\mathtt{a}[\chi_2(\mathtt{a}_0, \mathtt{i}, \mathtt{j})] \leq \mathtt{a}[\chi_3(\mathtt{i}, \mathtt{j})]$ as a template and the universal variable $\mathtt{a}_0$ is accordingly bounded by $\chi_1(\mathtt{i}, \mathtt{j})$. The final template is:

$$\forall \mathtt{a}_0.\ 0 \leq \mathtt{a}_0 < \chi_1(\mathtt{i}, \mathtt{j}) \Rightarrow \{\mathtt{a}[\chi_2(\mathtt{a}_0, \mathtt{i}, \mathtt{j})] \leq \mathtt{a}[\chi_3(\mathtt{i}, \mathtt{j})]\}$$

5.2 Program Sampling

We train the invariant templates of a function using its concrete program states collected from test runs. To this end we instrument the entry and exit of function bodies to dump values of function parameters and returns into a log-file, as pre- and post-states of the corresponding function resp. The format of a concrete program state is as follows.

$$V\ x = \begin{cases} u & \leftarrow \ \mathsf{type(x) = int}\ \mathit{or}\ \mathsf{bool} \\ [|0 : u_0, 1 : u_1, \cdots |] & \leftarrow \ \mathsf{type(x) = Array} \end{cases}$$

If a variable x is scalar, V maps it to the corresponding scalar value, u, sampled in the log file. Otherwise if x is of array type, V maps it to a record where each array element is indexed by its corresponding array subscript. We can use $V\ x\ j$ to retrieve the jth element (u_j) of the array x. The program may be run with multiple tests so we collect a set of pre- or post-states Vs.

5.3 Template Instantiation

With an invariant template and a set of program states Vs, we build a constraint system to find all valid template instantiations that fit the concrete states. For each state $V \in V$s, four constraints are generated. The first one constrains the array content for all array $\bar{a}$, which is encoded as

$$\bigwedge_{0 \le i < |\bar{a}|} \ \bigwedge_{0 \le k < \texttt{Array.length}\,(a_i)} a_i[k] = V\ a_i\ k$$

The second constraint enforces that the requirement that an instantiation should be invariant for all the elements in array a_i:

$$\bigwedge_{0 \le k < \texttt{Array.length}\,(a_i)} [k/a_{i0}][V\ \bar{x}/\bar{x}] \begin{array}{c} (0 \le a_{i0} < \varphi_i(\bar{x})) \Rightarrow (\bigwedge_{a_j \in \{\bar{a}/a_i\}} (\varphi_j(\bar{x}) \le \psi_j(\bar{x}) \\ < \varphi'_j(\bar{x}))) \Rightarrow \mathcal{Q}(\bar{a}[a_{i0}; \bar{x}], \bar{x}) \end{array}$$

In the first substitution, since a_{i0} is bounded by $\texttt{Array.length}(a_i)$ and must be no less than 0, we instantiate it to each possible value $k \in [0, \texttt{Array.length}\ A)$. In the second substitution, we replace scalar variables $\bar{x}$ by $V\ \bar{x}$. As an implication with a `false` premise is always an invariant, albeit useless, the third constraint guarantees the integrity of instantiated invariants:

$$0 \le \varphi_i(\bar{x}) \le \texttt{Array.length}\,(a_i) \wedge \bigwedge_{a_j \in \{\bar{a}/a_i\}} 0 \le \varphi_j(\bar{x}) \le \varphi'_j(\bar{x}) \le \texttt{Array.length}\,(a_j)$$

The fourth constraint aims to rule out array bound exceptions. Index expressions, after instantiation, must respect array length and be positive.

$$\bigwedge_{a[\chi] \in \mathcal{Q}(\bar{a}[a_{i0}; \bar{x}], \bar{x})} 0 \le \chi < \texttt{Array.length}\,(a)$$

These rules help to shrink the search space for likely invariants into a subset of those syntactically restricted by the template. To avoid over-fitting, we further require that all the coefficients must fall into the interval $[-d, d]$ where d is the maximum known constant in the function where the template is inferred. We feed all $4|V\text{s}|$ constraints to a decision procedure to find all the valid assignments for the unknown coefficients.

6 Array Type Refinements Checking

Inferred invariants are not guaranteed to generalize. We lift likely invariants into dependent types, which are subsequently validated through the type system introduced in Section 4. Initially we represent dependent base type as standard base type extended with a type refinement variable κ indicating an unknown type refinement. The dependent type P for an expression e must over-approximate e's side-effects. To generate the effect T for e, for all the arrays $x \in \texttt{mod}\,(\Gamma, e)$ where Γ is the type environment for e, we call an auxiliary function $\texttt{Push}\,(P, x : P_x)$ where P_x is a dependent type for x with unknown type refinement. This function pushes the effect to the right position in P; its definition is given as

$$\texttt{Push}\,(\{\{\nu : B \mid p\} \;/\; T\}, T') = \{\{\nu : B \mid p\} \;/\; T; T'\}$$
$$\texttt{Push}\,(\{x : P_1 \to P_2\}, T') = \{x : P_1 \to \texttt{Push}\,(P_2, T')\}$$

This process is performed before the generation of type constraints.

Type constraints over unknown type refinement variables that capture the subtyping relations between the types of various subexpressions are generated by traversing the program expression in a syntax-directed manner, applying the typing rules in Figure 4. We prove (see [30]) that the generated type constraints are solvable if and only if a valid type derivation exists. In our system, the type refinements for arrays are automatically inferred from test runs and are initially associated to all the unknown type refinement variables for array types. The type checker enumerates all possible solutions following the strategy in [22]. Notably, the type inference is abstracted into an abstract interpretation infrastructure [4]. Specifically, we solve these type constraints by iteratively removing the type refinements for unknown type refinement variables that prevent a type constraint from being satisfied using a decision procedure (an SMT solver) for the implication check in the subtyping rule shown in Figure 4.

Example 2. Consider the `insert` function in Figure 2. Refining `insert`'s standard type, we initially generate its dependent type with unknown type refinement variables: $\{\mathtt{i} : \{\mathtt{int}|\kappa_i\} \to \mathtt{j} : \{\mathtt{int}|\kappa_j\} \to \mathtt{a} : \{\mathtt{array}|\kappa_a\} \to \mathtt{ret} : \{\mathtt{int}|\kappa_{ret}\}/[a : \{\mathtt{array}|\kappa_a^{\mathtt{Eff}}\}]\}$. The variable $\kappa_a^{\mathtt{Eff}}$ represents the effect this function makes; syntactic sugar `ret` represents the return value. According to type checking rule Let, the effect of the `let` expression in `if` branch must be merged with the type environment for the locally-bound subexpressions. Thus, we generate a constraint:

$$\cdots ; \widetilde{\mathtt{a}} : \kappa_a ; i >= 0; \widetilde{\mathtt{a}}[\mathtt{i}] > \widetilde{\mathtt{a}}[\mathtt{j}] \vdash \{\forall \mathtt{a}_0.((\mathtt{a}_0 = \mathtt{i}+1 \Rightarrow \mathtt{a}[\mathtt{i}+1] = \widetilde{\mathtt{a}}[\mathtt{i}])$$
$$\wedge\, (\mathtt{a}_0 \neq \mathtt{i}+1 \Rightarrow \mathtt{a}[\mathtt{a}_0] = \widetilde{\mathtt{a}}[\mathtt{a}_0]))\} <: [\mathtt{i}-1/\mathtt{i}]\kappa_a$$

from the call to `insert` that forces the actual array $\mathtt{a}$ passed in at the callsite to be a subtype of the formal of `insert`, according to rule App. Note that $\widetilde{\mathtt{a}}$ denotes the old array before the update operation, in the type environment. Instantiating κ_a to the likely invariant inferred in Section 2 and executing the implication check in rule $\mathsf{Sub_Base}$ for subtyping would yield a verification condition, whose validity implies the invariant's correctness.

However, such implication checks are quantified formulae which are generally undecidable. The reason is that SMT solvers do not support quantifier instantiation for formulae of arbitrary structure. Our approach provides a heuristic wrapper to SMT solvers, similar to [28]. For a formula given as an universally quantified array invariant, we instantiate its universal variable with all the array accessing indices collected from the program. This mechanism is conservative because all such formulae are quantified over array indices, and is also sound. If a formula is also existentially quantified, we instantiate its existential with a fresh variable which is again matched to other corresponding universally quantified formulae.

7 Extensions

The template (over array and scalar variables) produced from Figure 5 covers a fairly general family of properties and is expressive enough to infer array invariants over an unbounded number of array elements. A natural question to ask is how we might judge the quality or usefulness of the invariants?

To show usefulness, we propose to use the inferred invariants to prove two important classes of program specifications: those that reflect sorting properties, and those that preserve the elements of the input. However, specifying suitable sorting and preservation invariants within a proper array bound requires array-specific domain knowledge. Instead, we equip our system with two built-in very simple templates for capturing sorting and preservation and use tests to instantiate such two specifications.

Array Sorting Invariants. The following template allows our system to infer an array sorting invariant for an arbitrary array a:

$$\forall \mathsf{a}_0.\ \chi(\bar{x}) \leq \mathsf{a}_0 < \chi'(\bar{x}) \Rightarrow \mathsf{a}[\mathsf{a}_0]\ \{\leq, \geq\}\ \mathsf{a}[\mathsf{a}_0 + 1]$$

Array Preservation Invariants. We are also interested in discovering and verifying properties like: "After sorting, the output array a_i preserves all the set of elements from the input array a_j". To this end, the postcondition of a function must be both universally and existentially quantified, and be able to refer to the state of the array a_j at the beginning of the function, which we denote as $\widetilde{a_j}$:

$$\forall a_{j0}.\exists a_{i0}.\ 0 \leq a_{j0} < \chi_j(\bar{x}) \Rightarrow 0 \leq a_{i0} < \chi_i(\bar{x}) \wedge (\widetilde{a_j}[\chi'_j(a_{j0}; \bar{x})] = a_i[\chi'_i(a_{i0}; \bar{x})])$$

where a_{i0} in this case is existentially quantified while another special variable a_{j0} for $\widetilde{a_j}$ (a_j may or may not equal to a_i) is universally quantified. An instantiation of this template yields a preservation invariant: for all the set of array elements (in some scope) in $\widetilde{a_j}$, they are preserved in a_i. Such templates can be created when we detect an array update involving two arrays during the process of generating the weakest precondition.

To deal with this extension, our template instantiation algorithm in Section 5 needs to be sightly extended. A concrete state logged in a file must include both $\bar{a}$ and $\widetilde{\bar{a}}$ (the array at the beginning of the function) when trying to infer a function's post-condition. Specifically, for a $\forall\exists$ template and a set of program

state Vs, for each state $V \in V$s, we again generate four similar constraints.

$$\begin{cases} 1.\ \bigwedge_{0 \le k < \texttt{Array.length}(a_i)} a_i[k] = V\ a_i\ k \wedge \bigwedge_{0 \le k < \texttt{Array.length}(\widetilde{a_j})} \widetilde{a_j}[k] = V\ \widetilde{a_j}\ k \\ 2.\ 0 \le \chi_j(x) \le \texttt{Array.length}(\widetilde{a_j}) \wedge 0 \le \chi_i(x) \le \texttt{Array.length}(a_i) \\ 3.\ 0 \le \chi'_j(a_{j0}; \bar{x}) < \texttt{Array.length}(\widetilde{a_j}) \wedge 0 \le \chi'_i(a_{i0}; \bar{x}) < \texttt{Array.length}(a_i) \\ 4.\ \bigwedge_{0 \le k < \texttt{Array.length}(a_j)} [k/a_{j0}][ex/a_{i0}][V\ \bar{x}/\bar{x}] \\ \quad 0 \le a_{j0} < \chi_j(\bar{x}) \Rightarrow 0 \le a_{i0} < \chi_i(\bar{x}) \wedge (\widetilde{a_j}[\chi'_j(a_{j0}; \bar{x})] = a_i[\chi'_i(a_{i0}; \bar{x})]) \end{cases}$$

The first three constraints are self-explanatory. The fourth constraint enforces that an (instantiated) invariant must hold for all the possible values of the universal variable a_{j0}. It also requires the solver to present a witness for the existential variable a_{i0} for each possible value of a_{j0} (ex is always a fresh variable).

8 Experimental Results

We have implemented our method [2] and evaluated it using benchmarks from recent related work [7,19]. We additionally infer invariants for *binarysearch*, *quicksort-inner* and the complete *mergesort* (see a detailed case study in [30]) and *insertionsort* programs. The results are summarized in Table 1. For the sorting programs, we try to infer and prove the sorted-ness of the result. For each of these benchmarks, our system successfully finds the desired pre- and post-conditions. In the table, we record the number of likely invariants and the time spent in invariant generation as gen_inv and inv_time, resp.; columns inv and vc_time represents the number of validated invariants and the time for validation. Columns tests refers to the number of tests (array input are randomly generated) needed to converge. In the experiment, we keep the size of input arrays to be a small value, either 4 or 5, to refute over-fitting invariants and achieve efficiencies. Notably, we use exactly *the same test suites* for the classic array sorting benchmarks. Compared to [7,19], our primary point of distinction is the use of test runs to infer array invariants and the absence of any requirement to annotate post-conditions, which are now inferred; the overall execution time of our implementation just slightly increases compared to [7], although we require much less annotations. A subset of our benchmarks can be verified via the system presented in [12], that extends abstract interpretation with a quantified abstract domain; like our technique, [12] also does not assume predefined predicates and annotated post-conditions. With the invariants inferred from a small set of tests, our approach can (significantly) more quickly converge to a solution compared to [12], which only relies on static semantics. Subsequent work [28] improves on [12], achieving results similar to ours, but at the cost of requiring programmers to explicitly specify a set of predicates and templates from which invariants are composed.

We also evaluated how increasing tests can affect the performance of our tool by tuning the number of test cases for *insertion-sort-full* in Table 2. It

[2] webpage: `https://www.cs.purdue.edu/homes/zhu103/asolve/index.html`

Table 1. ∀ invariant results

Benchmarks	gen_inv	inv	tests	inv_time	vc_time	total_time
parlindrome[19]	4	4	1	1.22s	0.13s	1.50s
seq-init[19]	2	2	1	0.33s	0.22s	0.71s
max-and-min[19]	4	4	1	0.27s	0.94s	1.78s
first-occur[19]	5	5	2	0.54s	0.59s	1.59s
sum-pair[19]	23	5	2	9.02s	1.63s	11.22s
array-init[7]	7	7	1	0.16s	0.25s	0.61s
array-reverse[7]	4	4	1	0.80s	0.36s	1.40s
array-copy[7]	7	7	1	0.46s	0.36s	1.05s
array-find[7]	2	2	1	0.10s	0.22s	0.45s
array-difference[7]	7	7	2	1.32s	0.76s	2.52s
binarysearch	8	5	2	0.95s	1.00s	2.46s
bubblesort-inner[7]	7	3	3	2.68s	1.40s	4.61s
selection-sort-inner[7]	6	4	3	3.03s	0.84s	4.36s
quick-sort-inner	12	8	2	2.58s	3.86s	7.26s
insert-sort-inner	8	3	2	0.66s	0.78s	1.76s
merge-sort-full	36	32	1	26.66s	22.66s	52.12s

Table 2. Increasing the number of tests for verifying insertion-sort-full

Benchmarks	gen_inv	inv	tests	inv_time	vc_time	total_time
insert-sort-full	20	9	1	3.57s	2.20s	6.43s
insert-sort-full	14	9	2	4.07s	1.83s	6.60s
insert-sort-full	12	9	3	4.34s	1.74s	6.83s
insert-sort-full	≤12	9	≥4	≥6.37s	≤1.79s	≥8.96s

can be seen that although increasing tests could reduce the number of false invariants generated (the verification time reduces correspondingly), the time spent in inference grows. Based on our experience, the number of tests never needs to be greater than a small number (2 or 3 in our experiments). Indeed, our experiments provide evidence to our claim that a simple random test suite suffices to infer very complex array invariants. Finally we show the result of applying our tool to infer preservation (∀∃) properties in Table 3.

Limitations. We briefly comment some limitations of our approach. First, the search space for array invariant is restricted by the shape of the general templates defined in Section 3, and can only discover program invariants that reside within this space. For example, our technique cannot find array invariants that express properties related to non-contiguous partitions of the array. Secondly, invariants discovered from the general template may sometimes be redundant. The reason is that discovered array invariants are all universally quantified. Adjusting the bound for universally quantified variables and the array indices computed from these variables accordingly may generate array invariants with different surface-level descriptions that have the same intension. Our approach bounds the value

Table 3. ∀∃ invariant results

Benchmarks	gen_inv	inv	tests	inv_time	vc_time	total_time
selection-sort	3	3	1	0.27s	1.41s	2.44s
bubble-sort	3	3	1	0.27s	1.68s	2.76s
quick-sort	9	8	1	1.31s	5.89s	9.05s

constants used by the general template to reduce the likelihood of redundant invariants. In future work, we plan to exploit deeper semantic approaches to filter redundant invariants.

9 Related Work and Conclusion

The idea of using a dependent type system to verify data structures is well studied. LiquidType [22] infers sound dependent types whose type refinements are conjunctions over atomic predicates presented from programmers. This approach can prove complex invariants over data structures [16], and has been extended to support abstract type refinements [29], which allows dependent types to be parametrized over type refinements. The ability to infer and verify flow-sensitive properties (for array programs) distinguishes our approach from these efforts.

Abstract interpretation [4] has long been used to infer array invariants. In [10] and its subsequent work [14], invariants are discovered based on an abstract interpretation over abstract values associated with each symbolic array partition. To overcome the problem that array indices can only be quantified over intervals from a fixed partition, [12] introduces quantified abstract domains and infers more general array properties of the form $\forall \mathtt{l}.\varphi(\mathtt{l}) \Rightarrow \psi(\mathtt{a}[\mathtt{l}]\cdots)$. However, abstract interpretation becomes difficult because φ must be under-approximated and it also requires programmers to provide templates for the invariants to be inferred. To overcome these difficulties, a similar but more scalable framework for array programs is presented in [5]. With parameterized bound expressions, arrays are automatically divided and each segment can be uniformly abstracted; such analyses are then combined via a reduced product with existing analyses for scalars. Our approach, in contrast, does not require array divisions and a fixed set of predicates in advance. Another dedicated array program analysis, fluid update [8], also avoids explicit array index partition. It also models array as an abstract location quantified by its index. To avoid the need for explicit array partitions, it retains both over- and under-approximative information of array updates, blurring the boundary of strong and weak updates. In contrast to our approach (dedicated to discovering complex invariants about unbounded array elements), their focus is on unified pointer, scalar and array reasoning.

Theorem provers have also been used for discovering invariants for array programs. Some approaches follow a counterexample guided abstraction refinement paradigm to extract information from spurious error paths about the range of array indices over which a universally quantified property may hold, or derive

array entries that violate an assertion from which predicates that may hold in unbounded intervals are then inferred [1,15,18,20,24]. Similar to our technique, these approaches are flexible because they do not assume a finite set of abstractions fixed in advance but generate suitable assertions on the fly. In contrast, our technique does not rely on program assertions or spurious program paths, and can infer likely program invariants before verification. Other techniques attempt to solve for unknown relations such as loop invariants that occur in verification conditions. This line of work has also been applied to array program in [2] by extending Horn solver to handle quantified predicates. Constraint-based invariant generation [3] is similarly adopted for discovering and verifying universally quantified properties over array variables. For example, a CLP program transformation [11] has been extended to handle array manipulating program in [7]. This work generates a set of verification conditions expressed as CLP (Array) program whose satisfiability implies that the program specification is proved. In [19], by means of Farkas' Lemma, the problem of discovering loop invariants is transformed into a satisfiability problem over the constraints generated from array programs. In contrast to these efforts, our approach builds simple constraints over concrete program states and hence is agnostic to specific program instructions so that it does not rely on the power of specialized theorem provers.

Our approach is inspired by the idea of using tests to improve the precision and efficiency of program analysis. Daikon [9] uses conjunctive template to find invariants, from configurations recorded along test runs. One of its extension in [21] uses equation solving to find array invariant but does not support implication and quantification. In contrast, we search quantified array invariants that allows implication (disjunction). In [6], since invariants are produced from symbolic execution of program paths that the concrete tests satisfy during their executions, the relevance of the generated invariants is increased compared to Daikon. In [13], the information obtained from static abstract interpretation is combined with that from tests to strengthen the ability of invariant generators but it does not consider quantified invariants. We are also inspired by recent interest in using machine learning to infer loop invariants. Compared to learning algorithms that synthesize program invariants in terms of classifiers distinguishing good and bad program samples [27,26,25], we search invariants from a broader program space since the typical learning techniques only search for invariants bounded by annotated assertions; we are unaware of prior learning based approaches capable of handling array programs as complex as the ones we have considered.

Conclusion. This paper presents a compositional and lightweight invariant inference technique that uses test runs to infer quantified array invariants. Our technique builds a constraint system for inferring array invariants on top of concrete program states. All likely flow-sensitive invariants inferred are validated by our dependent type system that allows side-effecting array updates. Experimental results demonstrate the practicality and expressivity of our approach.

References

1. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: SAFARI: SMT-based abstraction for arrays with interpolants. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 679–685. Springer, Heidelberg (2012)
2. Bjørner, N., McMillan, K., Rybalchenko, A.: On solving universally quantified horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 105–125. Springer, Heidelberg (2013)
3. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003)
4. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
5. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: POPL (2011)
6. Csallner, C., Tillmann, N., Smaragdakis, Y.: Dysy: Dynamic symbolic execution for invariant inference. In: ICSE (2008)
7. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Verifying array programs by transforming verification conditions. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 182–202. Springer, Heidelberg (2014)
8. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: Beyond strong vs. Weak updates. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 246–266. Springer, Heidelberg (2010)
9. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The daikon system for dynamic detection of likely invariants. Sci. Comput. Program., 35–45 (December 2007)
10. Gopan, D., Reps, T.W., Sagiv, S.: A framework for numeric analysis of array operations. In: POPL (2005)
11. Grebenshchikov, S., Gupta, A., Lopes, N.P., Popeea, C., Rybalchenko, A.: HSF(C): A software verifier based on horn clauses - (competition contribution). In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 549–551. Springer, Heidelberg (2012)
12. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL (2008)
13. Gupta, A.K., Majumdar, R., Rybalchenko, A.: From tests to proofs. International Journal on Software Tools for Technology Transfer (2013)
14. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: PLDI (2008)
15. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007)
16. Kawaguchi, M., Rondon, P., Jhala, R.: Type-based data structure verification. In: PLDI (2009)
17. Kawaguchi, M., Rondon, P.M., Jhala, R.: Dsolve: Safety verification via liquid types. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 123–126. Springer, Heidelberg (2010)
18. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 470–485. Springer, Heidelberg (2009)

19. Larraz, D., Rodríguez-Carbonell, E., Rubio, A.: SMT-based array invariant generation. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 169–188. Springer, Heidelberg (2013)
20. McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008)
21. Nguyen, T., Kapur, D., Weimer, W., Forrest, S.: Using dynamic analysis to discover polynomial and array invariants. In: ICSE (2012)
22. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: PLDI (2008)
23. Rondon, P.M., Kawaguchi, M., Jhala, R.: Low-level liquid types. In: POPL (2010)
24. Seghir, M.N., Podelski, A., Wies, T.: Abstraction refinement for quantified array assertions. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 3–18. Springer, Heidelberg (2009)
25. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 88–105. Springer, Heidelberg (2014)
26. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Nori, A.V.: Verification as learning geometric concepts. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 388–411. Springer, Heidelberg (2013)
27. Sharma, R., Nori, A.V., Aiken, A.: Interpolants as classifiers. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 71–87. Springer, Heidelberg (2012)
28. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: PLDI (2009)
29. Vazou, N., Rondon, P.M., Jhala, R.: Abstract refinement types. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 209–228. Springer, Heidelberg (2013)
30. Zhu, H., Nori, A.V., Jagannathan, S.: Dependent array type inference from tests. Tech. rep., Purdue Univsersity (2014), https://www.cs.purdue.edu/homes/zhu103/asolve/asolvetech.pdf