Predicate Abstraction for Linked Data Structures

Alexander Bakst(✉) and Ranjit Jhala

University of California, San Diego, USA
{abakst,jhala}@cs.ucsd.edu

Abstract. We present *Alias Refinement Types* (Art), a new approach that uses predicate-abstraction to automate the verification of correctness properties of linked data structures. While there are many techniques for checking that a heap-manipulating program adheres to its specification, they often require that the programmer annotate the behavior of each procedure, for example, in the form of loop invariants and pre- and post-conditions. We introduce a technique that lifts predicate abstraction to the heap by factoring the analysis of data structures into two orthogonal components: (1) Alias Types, which reason about the *physical* shape of heap structures, and (2) Refinement Types, which use simple predicates from an SMT decidable theory to capture the *logical* or semantic properties of the structures. We evaluate Art by implementing a tool that performs type *inference* for an imperative language, and empirically show, using a suite of data-structure benchmarks, that Art requires only 21 % of the annotations needed by other state-of-the-art verification techniques.

1 Introduction

Separation logic (SL) [31] has proven invaluable as a unifying framework for specifying and verifying correctness properties of linked data structures. Paradoxically, the richness of the logic has led to a problem – analyses built upon it are exclusively either expressive *or* automatic. To *automate* verification, we must restrict the logic to decidable fragments, *e.g.* list-segments [4,21], and design custom decision procedures [6,14,16,27,28] or abstract interpretations [7,23,40]. Consequently, we lose expressiveness as the resulting analyses cannot be extended to *user*-defined structures. To *express* properties of user-defined structures, we must fall back upon arbitrary SL predicates. We sacrifice automation as we require programmer assistance to verify entailments over such predicates [10,24]. Even when entailment is automated by specializing proof search, the programmer has the onerous task of providing complex auxiliary inductive invariants [9,30].

We observe that the primary obstacle towards obtaining expressiveness and automation is that in SL, machine state is represented by monolithic assertions that conflate reasoning about heap and data. While SL based tools commonly describe machine state as a conjunction of a pure, heap independent formula,

B. Jobstmann and K.R.M. Leino (Eds.): VMCAI 2016, LNCS 9583, pp. 65–84, 2016.
DOI: 10.1007/978-3-662-49122-5_3

abs :: (int) ⇒ nat[1]	
function abs(x){	x:*int*
if (0 <= x)	(0 ⩽ x); x:*int*
return x;	
var r = 0 - x;	r:{ν = 0 − x}; ¬(0 ⩽ x); x:*int*
return r;	
}	

Fig. 1. Refinement types

absR :: (x:⟨data:int⟩) ⇒ ()/&x ↦ ⟨data:nat[2]⟩	
function absR(x){	$\Gamma_0 \doteq$ x:⟨&x⟩ $\Sigma_0 \doteq$ &x ↦ ⟨data:*int*⟩
var d = x.data;	$\Gamma_1 \doteq$ d:*int*; Γ_0
var t = abs(d);	$\Gamma_2 \doteq$ t:nat^1; Γ_1
x.data = t;	$\Sigma_1 \doteq$ &x ↦ ⟨data:ν = *t*
return;	
}	

Fig. 2. Strongly updating a location

and a∗ combination of heap predicates, the heap predicates themselves conflate reasoning about links (*e.g.* reachability) and correctness properties (*e.g.* sizes or data invariants), which complicates automatic checking and inference.

In this paper, we introduce *Alias Refinement Types* (Art), a subset of separation logic that reconciles expressiveness and automation by *factoring* the representation of machine state along two independent axes: a *"physical"* component describing the basic shape and linkages between *heap cells* and a *"logical"* component describing semantic or relational properties of the *data* contained within them. We connect the two components in order to describe global logical properties and relationships of heap structures, using *heap binders* that name pure "snapshots" of the mutable data stored on the heap at any given point.

The separation between assertions about the heap's structure and heap-oblivious assertions about pure values allow Art to automatically *infer* precise data invariants. First, the program is type-checked with respect to the physical type system. Next, we generate a system of subtyping constraints over the *logical* component of the type system. Because the logical component of each type is heap-oblivious, solving the system of constraints amounts to solving a system of Horn clauses. We use predicate abstraction to solve these constraints, thus yielding precise refinements that summarize unbounded collections of objects.

In summary, this paper makes the following contributions:

- a description of Art and formalization of a constraint generation algorithm for inferring precise invariants of linked data structures;
- a novel soundness argument in which types are interpreted as assertions in separation logic, and thus typing derivations are interpreted as proofs;
- an evaluation of a prototype implementation that demonstrates Art is effective at verifying and, crucially, inferring data structure properties ranging from the sizes and sorted-ness of linked lists to the invariants defining binary search trees and red-black trees. Our experiments demonstrate that Art requires only 21 % of the annotation required by other techniques to verify intermediate functions in these benchmarks.

2 Overview

Refinements Types and Templates. A *basic* refinement type is a basic type, *e.g. int*, refined with a formula from a decidable logic, *e.g.* $nat \doteq \{\nu : int \mid \leqslant \nu\}$

is a refinement type denoting the set of non-negative integers, where *int* is the basic or *physical* part of the type and the refinement $0 \leqslant \nu$ is the *logical* part. A *template* is a refinement type where, instead of concrete formulas we have *variables* κ that denote the unknown to-be-inferred refinements. In the case that the refinement is simply `true`, we omit the refinement (*e.g.* $int = \{\nu : int \mid \texttt{true}\}$). We specify the behaviors of functions using refined function types: $(x_1 : t_1, \ldots, x_n : t_n) \Rightarrow t$. The input refinement types t_i specify the function's *preconditions* and t describes the *postcondition*.

Verification. ART splits verification into two phases: (1) *constraint generation*, which traverses the program to create a set of Horn clause constraints over the κ, and (2) *constraint solving*, which uses an off the shelf predicate abstraction based Horn clause solver [32] that computes a least fixpoint solution that yields refinement types that verify the program. Here, we focus on the novel step (1).

Path Sensitive Environments. To generate constraints ART traverses the code, building up an *environment* of type bindings, mapping program variables to their refinement types (or templates, when the types are unknown.) At each call-site (resp. return), ART generates constraints that the arguments (resp. return value) are a subtype of the input (resp. output) type. Consider `abs` in Fig. 1 which computes the absolute value of the integer input `x`. ART creates a template $(int) \Rightarrow \{\nu : int \mid \kappa_1\}$ where κ_1 denotes the unknown output refinement. (We write nat^1 in the figure to connect the inferred refinement with its κ.) In Fig. 1, the environment after each statement is shown on the right side. The initial environment contains a binder for `x`, which assumes that `x` may be *any int*. In each branch of the `if` statement, the environment is extended with a *guard* predicate reflecting the condition under which the branch is executed. As the type $\{\nu : int \mid \nu = \texttt{x}\}$ is problematic if `x` is mutable, we use SSA renaming to ensure each variable is assigned (statically) at most once.

Subtyping. The returns in the *then* and *else* yield subtyping constraints:

$$\begin{aligned} \texttt{x}{:}int, 0 \leqslant \texttt{x} \;&\vdash\; \&\{\nu : int \mid \nu = \texttt{x}\} \;\preceq\; \{\nu : int \mid \kappa_1\} \\ \texttt{x}{:}int, \neg(0 \leqslant \texttt{x}), \texttt{r}{:}\{\nu : int \mid \nu = 0 - \texttt{x}\} \;&\vdash\; \&\{\nu : int \mid \nu = \texttt{r}\} \;\preceq\; \{\nu : int \mid \kappa_1\} \end{aligned} \tag{1}$$

which respectively reduce to the Horn implications

$$\begin{aligned} (\texttt{true} \wedge 0 \leqslant \texttt{x}) \;&\Rightarrow\; \&(\nu = \texttt{x}) \;\Rightarrow\; \kappa_1 \\ (\texttt{true} \wedge \neg(0 \leqslant \texttt{x}) \wedge \texttt{r} = 0 - \texttt{x}) \;&\Rightarrow\; \&(\nu = \texttt{r}) \;\Rightarrow\; \kappa_1 \end{aligned}$$

By predicate abstraction [32] we find the solution $\kappa_1 \doteq 0 \leqslant \nu$ and hence infer that the returned value is a *nat*, *i.e.* non-negative.

References and Heaps. In Fig. 2, `absR` takes a *reference* to a structure containing an *int* valued `data` field, and updates the `data` field to its `abs`olute value. We use κ_2 for the output refinement; hence the type of `absR` desugars to: $(\texttt{x} : \&\texttt{x})/\&\texttt{x} \mapsto \langle \texttt{data} : int \rangle \Rightarrow ()/\&\texttt{x} \mapsto \langle \texttt{data} : \kappa_2 \rangle$ which states that `absR` *requires* a parameter `x` that is a *reference* to a *location* named `&x`. in an *input heap* where `&x` contains a structure with an *int*-valued `data` field. The function

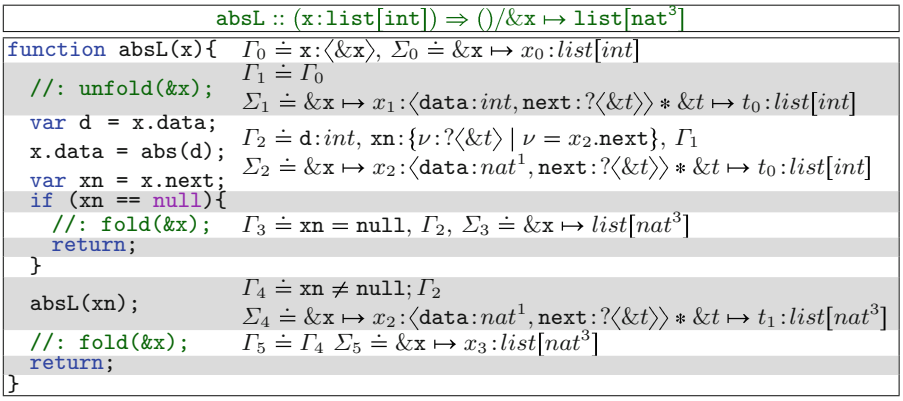

Fig. 3. Strongly updating a collection. The `fold` and `unfold` annotations are automatically inserted by a pre-analysis [3]

returns () (*i.e.* no value) in an *output heap* where the location `&x` is *updated* to a structure with a κ_2-valued `data`-field.

We extend the constraint generation to precisely track updates to locations. In Fig. 2, each statement of the code is followed by the environment Γ and heap Σ that exists after the statement executes. Thus, at the start of the function, `x` refers to a location, `&x`, whose `data` field is an arbitrary *int*. The call `abs(d)` returns a κ_1 that is bound to `t`, which is then used to *strongly update* the `data` field of `&x` from *int* to κ_1. At the `return` we generate a constraint that the return value and heap are sub-types of the function's return type and heap. Here, we get the *heap subtyping* constraint:

$$\texttt{x}{:}\langle \&\texttt{x}\rangle,\ \texttt{d}{:}\mathit{int},\ \texttt{t}{:}\kappa_1\ \vdash\ \&\ \&\texttt{x} \mapsto \langle \texttt{data}{:}\nu = \texttt{t}\rangle \preceq\ \&\texttt{x} \mapsto \langle \texttt{data}{:}\kappa_2\rangle$$

which reduces by field subtyping to the implication: $\kappa_1[\texttt{t}/\nu] \Rightarrow (\nu = \texttt{t}) \Rightarrow \kappa_2$ which (together with the previous constraints) can be solved to $\kappa_2 \doteq 0 \leqslant \nu$ letting us infer that `absR` updates the structure to make `data` non-negative. This is possible because the κ variables denote pure formulas, as reasoning about the heap shape is handled by the alias type system. Next we see how this idea extends to infer strong updates to collections of linked data structures.

Linked Lists. Linked lists can be described as iso-recursive alias types [38]. The definition

$$\texttt{type}\ \mathit{list}[A]\ \doteq\ \exists!\ell \mapsto t{:}\mathit{list}[A].h{:}\langle \texttt{data}{:}A, \texttt{next}?\langle\ell\rangle\rangle$$

says $\mathit{list}[A]$ is a *head* structure with a `data` field of type A, and a `next` field that is either `null` or a reference to the *tail*, denoted by the $?\langle\ell\rangle$ type. The heap $\ell \mapsto t{:}\mathit{list}[A]$ denotes that a *singleton* $\mathit{list}[A]$ is stored at the location denoted by ℓ *if* it is reachable at runtime. The $\exists!$ quantification means that the tail is *distinct* from every other location, ensuring that the list is inductively defined.

Consider `absL` from Fig. 3, which updates each `data` field of a list with its `abs`olute value. As before, we start by creating a κ_3 for the unknown output refinement, so the function gets the template

$$(x : \langle \&x \rangle)/\&x \mapsto x_0 : list[int] \Rightarrow ()/\&x \mapsto x_r : list[\kappa_3]$$

Figure 3 shows the resulting environment and heap after each statement.

The annotations `unfold` and `fold` allow Art to manage updates to collections such as lists. In Art, the user *does not* write `fold` and `unfold` annotations; these may be inferred by a straightforward analysis of the program [3].

Unfold. The location &x that the variable `x` refers to initially contains a $list[int]$ named with a *heap binder* x_0. The binder x_0 may be used in refinements. Suppose that `x` is a reference to a location containing a value of type $list[A]$. We require that before the fields of `x` can be accessed, the list must be *unfolded* into a head cell and a tail list. This is formalized with an `unfold(&x)` operation that unfolds the list at &x from $\&x \mapsto x_0 : list[int]$ to

$$\&x \mapsto x_1 : \langle \texttt{data} : int, \texttt{next}?\langle \&t \rangle \rangle * \&t \mapsto t_0 : list[int],$$

corresponding to *materializing* in shape analysis. The type system guarantees that the head structure and (if `next` is not `null`) the newly unfolded tail structure are unique and distinct. So, after unfolding, the structure at &x can be strongly updated as in `absR`. Hence, the field assignment generates a fresh binder x_2 for the updated structure whose `data` field is a κ_1, the output of `abs`.

Fold. After updating the `data` field of the head, the function tests whether the `next` field assigned to `xn` is `null`, and if so returns. Since the expected output is a list, Art requires that we *fold* the structure back into a $list[\kappa_3]$ – effectively computing a *summary* of the structure rooted at &x. As `xn` is `null` and `xn` : $\{\nu : ?\langle \&t \rangle \mid \nu = x_2.\texttt{next}\}$, `fold(&x)` converts $\&x \mapsto x_2 : \langle \texttt{data} : \kappa_1, \texttt{next} : ?\langle \&t \rangle \rangle$ to $\&x \mapsto list[\kappa_3]$ *after* generating a heap subtyping constraint which forces the "head" structure to be a subtype of the folded list's "head" structure.

$$\Gamma_3 \vdash \&x \mapsto x_2 : \langle \texttt{data} : \kappa_1, \ldots \rangle \preceq \&x \mapsto x_2 : \langle \texttt{data} : \kappa_3, \ldots \rangle \tag{2}$$

If instead, `xn` is non-null, the function updates the tail by recursively invoking `absL(xn)`. In this case, we can inductively assume the specification for `absL` and so in the heap *after* the recursive call, the tail location &t contains a $list[\kappa_3]$. As `xn` and hence the `next` field of x_2 is non-null, the `fold(&x)` transforms

$$\&x \mapsto x_2 : \langle \texttt{data} : \kappa_1, \texttt{next} : ?\langle \&t \rangle \rangle * \&t \mapsto t_1 : list[\kappa_3]$$

into $\&x \mapsto list[\kappa_3]$, as required at the `return`, by generating a heap subtyping constraints for the head and tail:

$$\Gamma_5 \vdash \&x \mapsto x_2 : \langle \texttt{data} : \kappa_1, \ldots \rangle \preceq \&x \mapsto x_2 : \langle \texttt{data} : \kappa_3, \ldots \rangle \tag{3}$$

$$\Gamma_5 \vdash \&\&t \mapsto t_1 : list[\kappa_3] \preceq \&t \mapsto t_1 : list[\kappa_3] \tag{4}$$

The constraints Eqs. (2), (3) and (4) are simplified field-wise into the implications $\kappa_1 \Rightarrow \kappa_3$, $\kappa_1 \Rightarrow \kappa_3$ and $\kappa_3 \Rightarrow \kappa_3$ which, together with the previous constraints (Eq. (1)) solve to: $\kappa_3 \doteq 0 \leqslant \nu$. Plugging this back into the template for `absL` we see that we have automatically inferred that the function *strongly* updates the contents of the input list to make *all* the `data` fields non-negative.

ART *infers* the update the type of the value stored at `&x` at `fold` and `unfold` locations because reasoning about the shape of the updated list is delegated to the alias type system. Prior work in refinement type inference for imperative programs [33] can not type check this simple example as the physical type system is not expressive enough. Increasing the expressiveness of the *physical* type system allows ART to "lift" invariant inference to collections of objects.

Snapshots. So far, our strategy is to factor reasoning about pointers and the heap into a "physical" alias type system, and functional properties (*e.g.* values of the `data` field) into quantifier- and heap-free "logical" refinements that may be inferred by classical predicate abstraction. However, reasoning about recursively defined properties, such as the length of a list, depends on the interaction between the physical and logical systems.

We solve this problem by associating recursively defined properties *not* directly with mutable collections on the heap, but with immutable *snapshot values* that capture the contents of the collection at a particular point in time. These snapshots are related to the sequences of pure values that appear in the definition of predicates such as list in [31]. Consider the heap Σ defined as:

$$\&x_0 \mapsto h : \langle \texttt{data} = 0, \texttt{next} = \&x_1 \rangle * \&x_1 \mapsto t : \langle \texttt{data} = 1, \texttt{next} = \texttt{null} \rangle$$

We say that *snapshot of* $\&x_0$ *in* Σ is the value v_0 defined as:

$$v_0 \doteq (\&x_0, \langle \texttt{data} = 0, \texttt{next} = v_1 \rangle) \quad v_1 \doteq (\&x_1, \langle \texttt{data} = 1, \texttt{next} = \texttt{null} \rangle)$$

Now, the logical system can avoid reasoning about the heap reachable from x_0 – which depends on the heap – and can instead reason about the length of the snapshot v_0 which is independent of the heap.

Heap Binders. We use *heap binders* to name snapshots in the refinement logic. In the desugared signature for `absR` from Fig. 2,

$$(x : \langle \&x \rangle) / \&x \mapsto x_0 : list[int] \Rightarrow () / \&x \mapsto x_r : list[nat]$$

the name x_0 refers to the snapshot of input heap at $\&x$. In ART, no reachable cell of a folded recursive structure (*e.g.* the list rooted at $\&x$) can be *modified* without first *unfolding* the data structure starting at the root: references pointing into the cells of a folded structure may not be dereferenced. Thus we can soundly update heap binders *locally* without updating transitively reachable cells.

Measures. We formalize structural properties like the *length* of a list or the *height* of a tree and so on, with a class of recursive functions called *measures*,

insert :: (A, x:?list[A]) ⇒ {ν:list[A] | (len(ν) = 1 + len(x))[4]}	
function insert(k, x)	$\Gamma_0 = \mathtt{k}:A;\ \mathtt{x}:?\langle\&\mathtt{x}\rangle\ \Sigma_0 = \&\mathtt{x} \mapsto x_0 : list[A]$
if (x == null) {	
var y = {data:k,next:null};	$\Gamma_1 \doteq \mathtt{y}:\langle\&\mathtt{y}\rangle; \mathtt{x} = \mathtt{null}; \Gamma_0$ $\Sigma_1 \doteq \&\mathtt{x} \mapsto x_0 : list[A] * \&\mathtt{y} \mapsto y_0 : \langle \mathtt{data}: A, \mathtt{next}:\mathtt{null}\rangle$
//: fold(&y)	$\Gamma_2 \doteq \mathsf{len}(y_1) = 1; \Gamma_1$ $\Sigma_2 \doteq \&\mathtt{x} \mapsto x_0 : list[A] * \&\mathtt{y} \mapsto y_1 : list[A]$
return y;	
}	
//: unfold(&x)	$\Gamma_3 \doteq \mathsf{len}(x_0) = 1 + \mathsf{len}(t_0); x \neq \mathtt{null}; \Gamma_0$ $\Sigma_3 \doteq \&\mathtt{x} \mapsto x_1 : \langle \mathtt{data}: a, \mathtt{next}:?\langle\&t\rangle\rangle * \&t \mapsto t_0 : list[a]$
if (k <= x.data) {	
var y = {data:k,next:x};	$\Gamma_4 \doteq \mathtt{y}:\langle\&\mathtt{y}\rangle; \Gamma_3$ $\Sigma_4 \doteq \&\mathtt{y} \mapsto y_2 : \langle \mathtt{data}: A, \mathtt{next}:?\langle\&x\rangle\rangle * \Sigma_3$
//: fold(&x)	$\Gamma_5 \doteq \mathsf{len}(x_2) = 1 + \mathsf{len}(t_0); \Gamma_4$ $\Sigma_5 \doteq \&\mathtt{x} \mapsto x_2 : list[A] * \&\mathtt{y} \mapsto y_2 : \langle \mathtt{data}: A, \mathtt{next}:?\langle\&x\rangle\rangle$
//: fold(&y)	$\Gamma_6 \doteq \mathsf{len}(y_3) = 1 + \mathsf{len}(x_2); \Gamma_5\ \Sigma_6 \doteq \&\mathtt{x} \mapsto y_3 : list[A]$
return y;	
}	
var z = x.next;	$\Gamma_7 \doteq u_0 : \kappa_4[t_0/x_0]; \mathtt{u}:\langle\&u\rangle; \mathtt{z}:?\langle\&t\rangle; \Gamma_3$ $\Sigma_7 \doteq \&\mathtt{x} \mapsto x_1 : \langle \mathtt{data}: A, \mathtt{next}:\langle\&u\rangle\rangle * \&\mathtt{u} \mapsto u_0 : list[A]$
var u = insert(k,z);	
x.next = u;	
//: fold(&x)	$\Gamma_8 \doteq \mathsf{len}(x_2) = 1 + \mathsf{len}(u_0); \Gamma_7\ \Sigma_8 \doteq \&\mathtt{x} \mapsto x_2 : list[A]$
return x;	
}	

Fig. 4. Inserting into a collection

which are catamorphisms over (snapshot values of) the recursive type. For example, we specify the length of a list with the measure:

$$\mathsf{len}: : list[A] \Rightarrow int \qquad \mathsf{len}(\mathtt{null}) = 0 \qquad \mathsf{len}(x) = 1 + \mathsf{len}(x.next)$$

We must reason *algorithmically* about these recursively defined functions. The direct approach of encoding measures as *background axioms* is problematic due to the well known limitations and brittleness of quantifier instantiation heuristics [13]. Instead, we encode measures as uninterpreted functions, obeying the congruence axiom, $\forall x, y.x = y \Rightarrow f(x) = f(y)$. Second, we recover the semantics of the function by adding *instantiation constraints* describing the measure's semantics. We add the instantiation constraints at `fold` and `unfold` operations, automating the reasoning about measures while retaining completeness [36].

Consider `insert` in Fig. 4, which adds a key `k` of type `A` into its position in an (ordered) $list[A]$, by traversing the list, and mutating its links to accomodate the new structure containing `k`. We generate a fresh κ_4 for the output type to obtain the function template:

$$(A, \mathtt{x}:?\langle\&\mathtt{x}\rangle)/\&\mathtt{x} \mapsto x_0 : list[A] \Rightarrow \langle\&l\rangle/\&l \mapsto \{\nu : list[A] \mid \kappa_4\}$$

Here, the snapshot of the input list `x` upon entry is named with the heap binder x_0; the output list must satisfy the (as yet unknown) refinement κ_4.

Constraint generation proceeds by additionally instantiating measures at each `fold` and `unfold`. When `x` is null, the `fold(&y)` transforms the binding

$\&\texttt{y} \mapsto y_0 : \langle \texttt{data} : A, \texttt{next:null} \rangle$ into a (singleton) list $\&\texttt{y} \mapsto y_1 : list[A]$ and so we add the instantiation constraint $\mathsf{len}(y_1) = 1$ to the environment. Hence, the subsequent `return` yields a subtyping constraint over the output list that simplifies to the implication:

$$\mathsf{len}(x_0) = 0 \wedge \mathsf{len}(y_1) = 1 \Rightarrow \nu = y_1 \Rightarrow \kappa_4 \tag{5}$$

When `x` is non-null, `unfold(&x)` transforms the binding $\&\texttt{x} \mapsto x_0 : list[A]$ to

$$\&\texttt{x} \mapsto x_1 : \langle \texttt{data} : a, \texttt{next}?\langle \&t \rangle \rangle * \&t \mapsto t_0 : list[A]$$

yielding the instantiation constraint $\mathsf{len}(x_0) = 1 + \mathsf{len}(t_0)$ that relates the length of the list's snapshot with that of its tail's. When `k <= x.data` the subsequent folds create the binders x_2 and y_3 with instantiation constraints relating their sizes. Thus, at the `return` we get the implication:

$$\mathsf{len}(x_0) = 1 + \mathsf{len}(t_0) \wedge \mathsf{len}(x_2) = 1 + \mathsf{len}(t_0) \wedge \mathsf{len}(y_3) = 1 + \mathsf{len}(x_2) \Rightarrow \nu = y_3 \Rightarrow \kappa_4 \tag{6}$$

Finally, in the else branch, after the recursive call to `insert`, and subsequent fold, we get the subtyping implication

$$\mathsf{len}(x_0) = 1 + \mathsf{len}(t_0) \wedge \kappa_4[\nu, x_0/u_0, t_0] \wedge \mathsf{len}(x_2) = 1 + \mathsf{len}(u_0) \Rightarrow \nu = x_2 \Rightarrow \kappa_4 \tag{7}$$

The recursive call that returns u_0 constrains it to satisfy the unknown refinement κ_4 (after substituting t_0 for the input binder x_0). Since the heap is factored out by the type system, the classical predicate abstraction fixpoint computation solves Eqs. (5), (6) and (7) to $\kappa_4 \doteq \mathsf{len}(\nu) = 1 + \mathsf{len}(x_0)$ inferring a signature that states that `insert`'s output has size one more than the input.

Abstract Refinements. Many important invariants of linked structures require us to reason about relationships *between* elements of the structure. Next, we show how our implementation of Art allows us to use *abstract refinements*, developed in the purely functional setting [37], to verify relationships between elements of linked data structures, allowing us to prove that `insertSort` in Fig. 5 returns an ordered list. To this end, we parameterize types with *abstract refinements* that describe relationships between elements of the structure. For example,

$$\texttt{type}\ list[A]\langle p \rangle \ \doteq \exists! l \mapsto t : list[\{\nu : A \mid p(\texttt{data}, \nu)\}]\langle p \rangle.h : \langle \texttt{data} : A, \texttt{next} : ?\langle l \rangle \rangle$$

is the list type as before, but now parameterized by an abstract refinement p which is effectively a relation between two A values. The type definition states that, if the data fields have values $x_1, \ldots, x_n$ where x_i is the i^{th} element of the list, then *for each* $i < j$ we have $p(x_i, x_j)$.

Ordered Lists. We *instantiate* the refinement parameters with concrete refinements to obtain invariants about linked data structures. For example, increasing lists are described by the type $incList[A] \doteq list[A]\langle (\leqslant) \rangle$.

Verification. Properties like sortedness may be *automatically infered* by using liquid typing [32]. Art infers the types:

$$\texttt{insertSort}::(?list[A]) \Rightarrow incList[A] \quad \texttt{insert}::(A, ?incList[A]) \Rightarrow incList[A]$$

```
(x:?list[A]) ⇒ {ν:?incList[A] | len(ν) = len(x)}
function insertSort(x){
  if (x == null) return null;
  //: unfold(&x);
  var y = insertSort(x.next);
  var t = insert(x.data, y);
  //: fold(&t);
  return t;
}
```

Fig. 5. Insertion Sort

i.e. that `insert` and `insertSort` return sorted lists. Thus, alias refinement types, measures, and abstract refinements enable both the specification and automated verification of functional correctness invariants of linked data structures.

3 Type Inference

To explain how Art infers refinement types as outlined in Sect. 2, we first explain the core features of Art's refinement type system. We focus on the more novel features of our type system; a full treatment may be found in [3].

3.1 Type Rules

Type Environments. We describe Art in terms of an imperative language Imp with record types and with the usual call by value semantics, whose syntax is given in Fig. 6. A *function environment* is defined as a mapping, Φ, from functions f to function schemas S. A *type environment* (Γ) is a sequence of *type bindings* $x{:}T$ and *guard expressions* e. A *heap* (Σ) is a finite, partial map from locations (ℓ) to type bindings. We write $\Gamma(x)$ to refer to T where $x{:}T \in \Gamma$, and $\Sigma(\ell)$ to refer to $x{:}T$ where the mapping $\ell \mapsto x{:}T \in \Sigma$.

Type Judgements. The type system of Art defines a judgement $\Phi \vdash f :: S$, which says given the environment Φ, the function f behaves according to its pre- and post-conditions as defined by S. An auxiliary judgement $\Phi, \Gamma, \Sigma \vdash s :: \Gamma'/\Sigma'$ says that, given the input environments Γ and Σ, s produces the output environments Γ' and Σ'. We say that a program p *typechecks* with respect to Φ if, for every function f defined in p, $\Phi \vdash f :: \Phi(f)$.

Well-Formedness. We require that types T be *well formed* in their local environments Γ and heaps Σ, written $\Gamma, \Sigma \vdash T$. A heap Σ must heap be well formed in its local environment Γ, written $\Gamma \vdash \Sigma$. The rules for the judgment [3] capture the intuition that a type may only refer to binders in its environment.

Subtyping. We require a notion of subsumption, *e.g.* so that the integer 2 can be typed either as $\{\nu : int \mid \nu = 2\}$ or simply int. The subtyping relation depends on the environment. For example, $\{\nu : int \mid \nu = x\}$ is a subtype of $\{\nu : int \mid \nu = 2\}$

$$
\begin{array}{lll}
\textbf{\textit{Expressions}} & e ::= & n \mid \texttt{true} \mid \texttt{false} \mid \texttt{null} \mid r_\ell \mid x \mid e \oplus e \\
\textbf{\textit{Statements}} & s ::= & s; s \mid x = e \mid y = x.f \mid x.f = e \mid \texttt{if } e \texttt{ then } s \texttt{ else } s \\
 & & \mid \texttt{return } e \mid x = \texttt{alloc } \{\overline{f : e}\} \mid x = f(\overline{e}) \\
 & & \mid \texttt{unfold}(\ell) \mid \texttt{fold}(\ell) \mid \texttt{concr}(x) \mid \texttt{pad}(\ell) \\
\textbf{\textit{Programs}} & p ::= & \overline{\texttt{function } f(\overline{x})\ \{s\}} \\
\textbf{\textit{Primitive Types}} & b ::= & \mathit{int} \mid \mathit{bool} \mid \alpha \mid \texttt{null} \mid \langle \ell \rangle \mid ?\langle \ell \rangle \\
\textbf{\textit{Types}} & \tau ::= & b \mid C[\overline{T}] \mid \langle \overline{f : T} \rangle \\
\textbf{\textit{Refined Types}} & T ::= & \{\nu : \tau \mid p\} \\
\textbf{\textit{Type Definition}} & C ::= & C[\overline{\alpha}] \doteq \exists!\ \Sigma.\ x : \langle \overline{f : T} \rangle \\
\textbf{\textit{Contexts}} & \Gamma ::= & \varnothing \mid x : T; \Gamma \mid e; \Gamma \\
\textbf{\textit{Heaps}} & \Sigma ::= & \mathsf{emp} \mid \Sigma * \ell \mapsto x : C[\overline{T}] \mid \Sigma * \ell \mapsto x : \langle \overline{f : T} \rangle \\
\textbf{\textit{Function Types}} & S ::= & \forall \overline{\ell}, \alpha.\ (\overline{x : T})/\Sigma \Rightarrow \exists! \overline{\ell'}.\ x' : T'/\Sigma'
\end{array}
$$

$n \in$ Integers, $r_\ell \in$ Reference Constants, $x, y, f \in$ Identifiers, $\oplus \in \{+, -, \ldots\}$

Fig. 6. Syntax of Imp programs and types

if $x : \{\nu : \mathit{int} \mid \nu = 2\}$ holds as well. Subtyping is formalized by the judgment $\Gamma \vdash T_1 \preceq T_2$, of which selected rules are shown in Fig. 7. Subtyping in Imp reduces to the validity of logical implications between refinement predicates. As the refinements are drawn from a decidable logic of Equality, Linear Arithmetic, and Uninterpreted Functions, validity can be automatically checked by SMT solvers [13]. The last two rules convert between non-null and possibly null references ($\langle \ell \rangle$ and $?\langle \ell \rangle$).

Heap Subtyping. The heap subtyping judgment $\Gamma \vdash \Sigma \preceq \Sigma'$ describes when one heap is subsumed by another. Figure 7 summarizes the rules for heap subsumption. Heap subtyping is *covariant*, which is sound because our type system is flow sensitive – types in the heap are updated *after* executing a statement.

Statements. When the condition x, y *fresh* appears in the antecedent of a rule, it means that x and y are distinct names that do not appear in the input environment Γ or heap Σ. We write $[y/x]$ for the capture avoiding substitution that maps x to y. The rules for sequencing, assignment, control-flow joins, and function calls are relatively straightforward extensions from previous work (*e.g.* [33]). Selected rules are given in Fig. 8. The complete set of rules may be found in [3].

Allocation. In T-ALLOC, a record is constructed from a sequence of field name and expression bindings. The rule types each expression e_f as T_f, generates a record type T, and allocates a fresh location ℓ on the heap whose type is T. To connect fields with their containing records, we create a new binder y denoting the record, and use the helper *NameFields* [3] to *strengthen* the type of each field-binding for y from $f : \{\nu_f : \tau \mid p\}$, to $f : \{\nu_f : \tau \mid p \wedge \nu_f = \mathsf{Field}(\nu, f)\}$. Here, Field is an uninterpreted function.

Access. T-RD and T-WR both require that non-*null* pointers are used to access a field in a record stored on the heap. As T-ALLOC strengthens each type with *NameFields*, the type for y in T-RD contains the predicate $\nu_{f_i} = \mathsf{Field}(\nu, f_i)$. Any facts established for y are linked, in the refinement logic, with the original

Subtyping $\boxed{\Gamma \vdash T_1 \preceq T_2,\ \Gamma \vdash \Sigma \preceq \Sigma'}$

$$\frac{\mathit{Valid}(\llbracket \Gamma \rrbracket \Rightarrow \llbracket p \rrbracket \Rightarrow \llbracket p' \rrbracket)}{\Gamma \vdash \{\nu{:}b \mid p\} \preceq \{\nu{:}b \mid p'\}}\ \preceq\text{-B}$$

$$\frac{\mathit{Valid}(\llbracket \Gamma \rrbracket \Rightarrow \llbracket p \rrbracket \Rightarrow \llbracket p' \wedge \nu \neq \texttt{null} \rrbracket)}{\Gamma \vdash \{\nu{:}?\langle\ell\rangle \mid p\} \preceq \{\nu{:}\langle\ell\rangle \mid p'\}}\ \preceq\text{-DOWN}$$

$$\frac{\mathit{Valid}(\llbracket \Gamma \rrbracket \Rightarrow \llbracket p \rrbracket \Rightarrow \llbracket p' \rrbracket)}{\Gamma \vdash \{\nu{:}\langle\ell\rangle \mid p\} \preceq \{\nu{:}?\langle\ell\rangle \mid p'\}}\ \preceq\text{-UP1} \qquad \frac{\mathit{Valid}(\llbracket \Gamma \rrbracket \Rightarrow \llbracket p \rrbracket \Rightarrow \llbracket p' \rrbracket)}{\Gamma \vdash \{\nu{:}\texttt{null} \mid p\} \preceq \{\nu{:}?\langle\ell\rangle \mid p'\}}\ \preceq\text{-UP2}$$

$$\frac{}{\Gamma \vdash \mathsf{emp} \preceq \mathsf{emp}}\ \preceq\text{-EMP} \qquad \frac{\Gamma \vdash \Sigma \preceq \Sigma' \quad \Gamma \vdash T \preceq T'}{\Gamma \vdash \Sigma * \ell \mapsto x{:}T \preceq \Sigma' * \ell \mapsto x{:}T'}\ \preceq\text{-HEAP}$$

Heap Folding $\boxed{\Gamma \vdash x{:}T_1/\Sigma_1 \rhd x{:}T_2/\Sigma_2}$

$$\frac{\mathit{locs}(T_1) \cap \mathit{Dom}(\Sigma_1) = \varnothing \quad \Gamma \vdash T_1 \preceq T_2}{\Gamma \vdash x{:}T_1/\Sigma_1 \rhd x{:}T_2/\Sigma_2}\ \text{F-BASE}$$

$$\frac{\begin{array}{c}\Sigma_1 = \Sigma_1' * \ell \mapsto x{:}T \quad \Sigma_2 = \Sigma_2' * \ell \mapsto x{:}T' \\ \Gamma \vdash \{\nu{:}\langle\ell\rangle \mid p\} \preceq T_2 \quad \Gamma \vdash x{:}T/\Sigma_1' \rhd x{:}T'/\Sigma_2'\end{array}}{\Gamma \vdash y{:}\{\nu{:}\langle\ell\rangle \mid p\}/\Sigma_1 \rhd y{:}T_2/\Sigma_2}\ \text{F-REF}$$

$$\frac{\begin{array}{c}\Gamma \vdash \{\nu{:}?\langle\ell\rangle \mid p\} \preceq T_2 \\ \Sigma_1 = \Sigma_1' * \ell \mapsto y{:}T \quad \Sigma_2 = \Sigma_2' * \ell \mapsto y{:}T' \\ x{:}\{\nu{:}?\langle\ell\rangle \mid p \wedge \nu \neq \mathit{null}\}; \Gamma \vdash y{:}T/\Sigma_1' \rhd y{:}T'/\Sigma_2' \\ x{:}\{\nu{:}?\langle\ell\rangle \mid p \wedge \nu = \mathit{null}\}; \Gamma \vdash y{:}T/\Sigma_1' \rhd y{:}T'/\Sigma_2'\end{array}}{\Gamma \vdash x{:}\{\nu{:}?\langle\ell\rangle \mid p\}/\Sigma_1 \rhd x{:}T_2/\Sigma_2}\ \text{F-?REF}$$

$$\frac{\Gamma \vdash x{:}T_i/\Sigma_1 \rhd x{:}T_i'/\Sigma_2}{\Gamma \vdash y{:}\langle \overline{f_i{:}T_i} \rangle/\Sigma_1 \rhd y{:}\langle \overline{f_i{:}T_i'} \rangle/\Sigma_2}\ \text{F-HEAP}$$

Fig. 7. Selected subtyping, heap subtyping, and heap folding rules

record's field: when a record field is *mutated*, a *new* type binding is created in the heap, and each unmutated field is linked to the old record using Field.

Concretization. As heaps also contain bindings of names to types, it would be tempting to add these bindings to the *local* environment to strengthen the subtyping context. However, due to the presence of possibly null references, adding these bindings would be unsound. Consider the program fragment:

```
function f(){ return null; }
function g(){ var p = f(); assert(false) }
```

One possible type for `f` is $()/\mathsf{emp} \Rightarrow \exists!\ell.\ r{:}?\langle\ell\rangle/\ell \mapsto x{:}\{\nu : \mathit{int} \mid \mathit{false}\}$ because the location ℓ is unreachable. If we added the binding $x{:}\{\nu : \mathit{int} \mid \mathit{false}\}$ to Γ after the call to `f`, then the `assert(false)` in `g` would unsoundly typecheck!

Statement Typing $\boxed{\Phi, \Gamma, \Sigma \vdash s :: \Gamma'/\Sigma'}$

$$\frac{\Gamma \vdash x : \langle \ell \rangle \quad \ell \mapsto z{:}\langle \overline{f_i : T_i} \rangle \in \Sigma}{\Phi, \Gamma, \Sigma \vdash y = x.f_i :: y{:}T_i; \Gamma/\Sigma} \text{ T-RD}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : \langle \ell \rangle \quad \Gamma \vdash e : \{\nu{:}\tau \mid p\} \\ T_r = \mathit{NameFields}(z, \langle f_0{:}T_0, \ldots, f_i{:}\{\nu{:}\tau \mid \nu = e\}, \ldots \rangle) \quad z \textit{ fresh}\end{array}}{\Phi, \Gamma, \ell \mapsto y{:}\langle \overline{f_j : T_j} \rangle * \Sigma \vdash x.f_i = e :: \Gamma/\ell \mapsto z{:}T_r * \Sigma} \text{ T-WR}$$

$$\frac{\text{for each } e_f,\ \Gamma, \Sigma \vdash e_f : T_f \quad T = \mathit{NameFields}(z, \langle \overline{f : T} \rangle) \quad \ell, z \textit{ fresh}}{\Phi, \Gamma, \Sigma \vdash x = \texttt{alloc}\ \{\overline{f : e_f}\} :: x{:}\langle \ell \rangle; \Gamma/\ell \mapsto z{:}T * \Sigma} \text{ T-ALLOC}$$

$$\frac{\Gamma, \Sigma \vdash x : \langle \ell \rangle \quad T_y = \{\nu{:}\tau \mid p\} \quad T_z = \{\nu{:}\tau \mid \nu = y\} \quad z \textit{ fresh}}{\Phi, \Gamma, \ell \mapsto y{:}T_y * \Sigma \vdash \texttt{concr}(x) :: y{:}T_y; \Gamma/\ell \mapsto z{:}T_z * \Sigma} \text{ T-CONCR}$$

$$\frac{\begin{array}{c}\Gamma \vdash C[\overline{\alpha}] = \exists!\ \Sigma_c.\ x_c{:}T_c \quad C[\overline{\alpha}] \vdash_M \mathsf{m}(x) \doteq e_{\mathsf{m}} \\ \Sigma = \ell \mapsto x{:}\{\nu{:}C[\overline{T}] \mid q\} * \Sigma_0 \quad \Sigma' = \ell \mapsto x_c{:}\overline{[T/\alpha]}T_c * \overline{[T/\alpha]}\Sigma_c * \Sigma_0 \\ \Gamma, \Sigma \vdash \overline{T} \quad \Gamma, \Sigma' \vdash \Sigma' \quad \mathit{Dom}(\Sigma_c),\ \mathit{Binders}(\Sigma_c),\ x_c \textit{ fresh}\end{array}}{\Phi, \Gamma, \Sigma \vdash \texttt{unfold}(\ell) :: (\bigwedge_{\mathsf{m}} \mathsf{m}(x) = e_{\mathsf{m}}); \Gamma/\Sigma'} \text{ T-UNFOLD}$$

$$\frac{\begin{array}{c}\Gamma \vdash C[\overline{\alpha}] = \exists!\ \Sigma_c.\ x{:}T_c \\ \Gamma \vdash x{:}T_x/\Sigma_x \rhd x{:}\overline{[T/\alpha]}T_c/\overline{[T/\alpha]}\Sigma_c \quad \Gamma \vdash \ell \mapsto y{:}T_y * \Sigma' \quad \Gamma \vdash \Sigma \preceq \Sigma' \\ C[\overline{\alpha}] \vdash_M \mathsf{m}(x) \doteq e_{\mathsf{m}} \quad T_y = \{\nu{:}C[\overline{T}] \mid \textstyle\bigwedge_{\mathsf{m}} \mathsf{m}(\nu) = e_{\mathsf{m}}\} \quad y \textit{ fresh}\end{array}}{\Phi, \Gamma, \ell \mapsto x{:}T_x * \Sigma_x * \Sigma \vdash \texttt{fold}(\ell) :: \Gamma/\ell \mapsto y{:}T_y * \Sigma'} \text{ T-FOLD}$$

Fig. 8. Selected Statement Typing Rules. We assume that type definitions (and, hence, measures over these definitions) $\Gamma \vdash C[\overline{\alpha}] = \exists!\ \Sigma.\ x{:}T$ are α-convertible.

We thus require that in order to include a heap binder in a local context, Γ, the location must first be made *concrete*, by checking that a reference to it is definitely *not* null. Concretization of a location ℓ is achieved with the *heap annotation* $\texttt{concr}(x)$. Given a non-null reference, T-CONCR transforms the local context Γ and the heap Σ by (1) adding the binding $y{:}T_y$ at the location ℓ to Γ; (2) adding a *fresh* binding $z{:}T_z$ at ℓ that expresses the equality $y = z$.

Unfold. T-UNFOLD describes how a type constructor application $C[\alpha]$ may be unfolded according to its definition. The context is modified to contain the new heap locations corresponding to those mentioned in the type's definition. The rule assumes an α-renaming such that the locations and binders appearing in the definition of C are *fresh*, and then instantiates the formal type variables $\overline{\alpha}$ with the actual $\overline{T}$. The environment is strengthened using the thus-instantiated measure bodies.

Fold. Folding a set of heap bindings *into* a data structure is performed by T-FOLD. Intuitively, to fold a heap into a type application of C, we ensure that it is consistent with the definition of C. Note that the rules assume an appropriate α-renaming of the definition of C. Simply requiring that the heap-to-be-folded

be a subtype of the definition's heap is too restrictive. Consider the first `fold` in `absL` in Fig. 3. As we have reached the end of the list $xn = \texttt{null}$ we need to fold

$$\&\texttt{x} \mapsto x_1 : \langle \texttt{data} : nat, \texttt{next}?\langle \&t \rangle \rangle * \&t \mapsto t_0 : list[int]$$

into $\&\texttt{x} \mapsto x_2 : list[nat]$. An application of heap subtyping, *i.e.* requiring that the heap-to-be-folded is a subtype of the body of the type definition, would require that $\&t \mapsto list[int] \preceq \&t \mapsto list[nat]$, which does not hold! However, the fold *is* safe, as the `next` field is `null`, rendering $\&t$ unreachable. We observe that it is safe to fold a heap into another heap, so long as the sub-heap of the former that is *reachable from a given type* is subsumed by the latter heap.

Our intuition is formalized by the relation $\Gamma, \Sigma \vdash x : T_1/\Sigma_1 \rhd x : T_2/\Sigma_2$, which is read: "given a local context Γ, Σ, the type T_1 and the heap Σ_1 may be folded into the type T_2 and heap Σ_2." F-BASE defines the ordinary case: from the point of view of a type T, any heap Σ_1 may be folded into another heap Σ_2. On the other hand, if T_1 is a reference to a location ℓ, then F-REF additionally requires the folding relation to hold at the type bound at ℓ in Σ_1.

F-?REF splits into two cases, depending on whether the reference is null or not. The relation is checked in two strengthened environments, respectively assuming the reference is in fact null and non-null. This strengthening allows the subtyping judgement to make use reachability. Recall the first fold in `absL` that happens when $\texttt{xn} = \texttt{null}$. To check the `fold(&x)`, the rule requires that the problematic heap subtyping $\Gamma \vdash \&t \mapsto list[int] \preceq \&t \mapsto list[nat]$ only holds when `x.next` is non-null, *i.e.* when Γ is

$$\texttt{xn} : \{\nu :?\langle \&t \rangle \mid \nu = x_2.\texttt{next}\},\ \texttt{xn} = \texttt{null},\ x_2.\texttt{next} \neq \texttt{null}$$

This heap subtyping reduces to checking the validity of the following, which holds as the antecedent is inconsistent:

$$\texttt{xn} = x_2.\texttt{next} \wedge \texttt{xn} = \texttt{null} \wedge x_2.\texttt{next} \neq \texttt{null} \Rightarrow 0 \leqslant \nu.$$

3.2 Refinement Inference

In the definition of the type system we assumed that type refinements were given. In order to *infer* the refinements, we replace each refinement in a program with a unique variable, κ_i, that denotes the unknown refinement. More formally, let $\hat{\Phi}$ denote a function environment as before except each type appearing in $\hat{\Phi}$ is optionally of the form $\{\nu : \tau \mid \kappa_i\}$, *i.e.* its refinement has been omitted and replaced with a unique κ variable. Given a set of function definitions p and a corresponding environment of *unrefined* function signatures $\hat{\Phi}$, to infer the refinements denoted by each κ we extract a system of Horn clause constraints C. The constraints, C, are *satisfiable* if there exists a mapping of K of κ-variables to refinement formulas such each implication in KC, *i.e.* substituting each κ_i with its image in K, is valid. We solve the constraints by abstract interpretation in the predicate abstraction domain generated from user-supplied predicate templates.

$$
\begin{array}{l}
\mathsf{CGen} : \mathsf{FunEnv} \times \mathsf{TypeEnv} \times \mathsf{HeapEnv} \times \mathsf{Stmt} \rightarrow \{\mathsf{Constr}\} \times \mathsf{TypeEnv} \times \mathsf{HeapEnv} \\
\mathsf{CGen}(\Phi,\Gamma,\Sigma,\mathsf{s}) = \textbf{match}\ \mathsf{s}\ \textbf{with} \\
\ldots \\
\quad \mid\ \mathsf{y} = \mathsf{x.f} \rightarrow \textbf{let}\ \ell = \mathsf{loc}(\Gamma(\mathsf{x}))\ \textbf{in}\ (\{\Gamma \vdash \Gamma(\mathsf{x}) \preceq \langle\ell\rangle\},\ \mathsf{y{:}TypeAt}(\Sigma,\ell);\Gamma,\Sigma) \\
\quad \mid\ \mathsf{x.f} = \mathsf{e} \rightarrow \textbf{let}\ (\mathsf{cs},\ \mathsf{t}) \quad\ = \mathsf{CGEx}(\Gamma,\Sigma,\mathsf{e}) \\
\qquad\qquad\qquad\qquad\ \ell \quad\ = \mathsf{Loc}(\mathsf{t}) \\
\qquad\qquad\qquad\qquad\ (\mathsf{y}{:}\mathsf{T_y},\ \mathsf{z}) = (\Sigma(\ell),\ \mathsf{FreshId}()) \\
\qquad\qquad\qquad\qquad\ \mathsf{ht} \quad\ = \mathsf{NameFields}(\mathsf{z},\ \mathsf{T_y}[\mathsf{f} : \mathsf{Shape}(\mathsf{t}) \sqcap (\mathsf{v} = \mathsf{e})]) \\
\qquad\qquad\qquad\ \textbf{in}\ \ (\mathsf{cs} \cup \{\Gamma \vdash \mathsf{t} \preceq \langle\ell\rangle\},\ \Gamma,\ \Sigma[\ell \mapsto \mathsf{z{:}ht}])
\end{array}
$$

Fig. 9. Statement constraint generation

For more details, we refer the reader to [32]. We thus infer the refinements missing from $\hat{\Phi}$ by finding such a solution, if it exists.

Constraint Generation. Constraint generation is carried out by the procedure CGen which takes a function environment (Φ), type environment (Γ), heap environment (Σ), and statement (s) as input, and ouputs (1) a set of Horn constraints over refinement variables κ that appear in Φ, Γ, and Σ; (2) a new type- and heap-environment which correspond to the effect (or post-condition) after running s from the input type and heap environment (pre-condition).

The constraints output by CGen correspond to the well-formedness constraints, $\Gamma, \Sigma \vdash T$, and subtyping constraints, $\Gamma \vdash T \preceq T'$, defined by the type system. *Base* subtyping constraints $\Gamma \vdash \{\nu : b \mid p\} \preceq \{\nu : b \mid q\}$ correspond to the (Horn) Constraint $[\![\, \Gamma \,]\!] \Rightarrow p \Rightarrow q$, where $[\![\, \Gamma \,]\!]$ is the conjunction of all of the refinements appearing in Γ [32]. *Heap* Subtyping constraints $\Gamma \vdash \Sigma \preceq \Sigma'$ are decomposed via classical subtyping rules into base subtyping constraints between the types stored at the corresponding locations in Σ and Σ'. This step crucially allows the predicate abstraction to sidestep reasoning about reachability and the heap, enabling inference.

`CGen` proceeds by pattern matching on the statement to be typed. Each `FreshType()` or `Fresh()` call generates a new κ variable which may then appear in subtyping constraints as described previously. Thus, in a nutshell, `CGen` creates `Fresh` templates for unknown refinements, and then performs a type-based *symbolic execution* to generate constraints over the templates, which are solved to infer precise refinements summarizing functions and linked structures. As an example, the cases of CGen corresponding to T-RD and T-WR are show in Fig. 9.

3.3 Soundness

The constraints output by CGen enjoy the following property. Let (C,Γ',Σ') be the output of $\mathtt{CGen}(\hat{\Phi},\Gamma,\Sigma,\mathrm{s})$. If C is satisfiable, then there exists some solution K such that $K\hat{\Phi}, K\Gamma, K\Sigma \vdash s :: K\Gamma'/K\Sigma'$ [32], that is, there is a type derivation using the refinements from K. Thus K yields the inferred program typing $\Phi \doteq K\hat{\Phi}$, where each unknown refinement has been replaced with its solution, such that $\Phi \vdash f :: \Phi(f)$ for each f defined in the program p.

To prove the soundness of the type system, we translate types, environments and heaps into separation logic *assertions* and hence, typing derivations into *proofs* by using the interpretation function $[\![\cdot]\!]$. We prove [3] the following:

Theorem 1. [Typing Translation]

- $If\ \Phi, \Gamma, \Sigma \vdash s :: \Gamma'/\Sigma'\ then\ [\![\Phi]\!] \vdash \{[\![\Gamma, \Sigma]\!]\}\ s\ \{[\![\Gamma', \Sigma']\!]\}$
- $If\ \Phi \vdash f :: S\ then\ [\![\Phi]\!] \vdash \{Pre(S)\}\ Body(f)\ \{Post(S)\}$

$Pre(S)$, $Post(S)$ and $Body(f)$ are the translations of the input and output types of the function, the function (body) statement. As a corollary of this theorem, our main soundness result follows:

Corollary 1. [Soundness] $If \Phi, \varnothing, \mathsf{emp} \vdash s :: \Gamma/\Sigma, then\ [\![\Phi]\!] \vdash \{true\}\ s\ \{true\}$

If we typecheck a program in the empty environment, we get a valid separation logic proof of the program starting with the pre-condition *true*. We can encode programmer-specified `asserts` as calls to a special function whose type encodes the assertion. Thus, the soundness result says that if a program typechecks then on *all* executions of the program, starting from *any* input state: (1) all memory accesses occur on non-*null* pointers, and (2) all assertions succeed.

4 Experiments

We have implemented alias refinement types in a tool called Art. The user provides (unrefined) function signatures, and Art infers (1) annotations required for alias typing, and (2) refinements that capture correctness invariants. We evaluate Art on two dimensions: the first demonstrates that it is *expressive* enough to verify a variety of sophisticated properties for linked structures; the second that it provides a significant *automation* over the state-of-the-art, represented by the SMT-based VCDryad system. VCDryad has annotations comparable to other recent tools that use specialized decision procedures to discharge Separation Logic VCs [11]. Our benchmarks are available at [1].

Expressiveness. Table 1 summarizes the set of data structures, procedures, and properties we used to evaluate the expressiveness of Art. The user provides the type definitions, functions (with unrefined type signatures), and refined type specifications to be verified for top-level functions, *e.g.* the top-level specification for `insertSort`. **LOC** is lines of code and **T**, the verification time in seconds.

We verified the following properties, where applicable: [Len] the output data structures have the expected length; [Keys] the elements, or "keys" stored in each data structure [Sort] the elements are in sorted order [Order] the ouput elements have been labeled in the correct order (*e.g.* preorder) [Heap] the elements satisfy the max heap property [BST] the structure satisfies the binary search tree property [Red-black] the structure satisfies the red-black tree property.

Table 1. Experimental Results (Expressiveness)

Data Structure	Properties	Procedures	LOC	T
Singly linked list	Len, Keys	`append`, `copy`, `del`, `find`, `insBack`, `insFront`, `rev`	73	2
Doubly linked list	Len, Keys	`append`, `del`, `delMid`, `insBack`, `insMid`, `insFront`	90	16
Cyclic linked list	Len, Keys	`delBack`, `delFront`, `insBack`, `insFront`	49	2
Sorted linked list	Len, Keys, Sort	`rev`, `double`, `pairwiseSum`, `insSort`, `mergeSort`, `quickSort`	135	10
Binary Tree	Order, Keys	`preOrder`, `postOrder`, `inOrder`	31	2
Max heap	Heap, Keys	`heapify`	48	27
Binary search tree	BST, Keys	`ins`, `find`, `del`	105	11
Red-black tree	Red-black, BST, Keys	`ins`, `del`	322	213

Automation. To demonstrate the effectiveness of *inference*, we selected benchmarks from Table 1 that made use of loops and intermediate functions requiring extra proof annotations in the form of pre- and post-conditions in VCDryad, and then used type inference to infer the intermediate pre- and post-conditions. The results of these experiments is shown in Table 2. We omit incomparable benchmarks, and those where the implementations consist of a single top-level function. We compare the number of tokens required to specify type refinements (in the case of Art) and pre- and post-conditions (for VCDryad). The table distinguishes between two types of annotations: (1) those required to specify the desired behavior of the top-level procedure, and (2) additional annotations required (such as intermediate function specifications). Our results suggest that it is possible to verify the correctness of a variety of data-structure manipulating algorithms without requiring many annotations beyond the top-level specification. On the benchmarks we examined, overall annotations required by Art were about 34 % of those required by VCDryad. Focusing on intermediate function specification, Art required about 21 % of the annotation required by VCDryad.

Limitations. Intuitively, Art is limited to "tree-like" ownership structures: while sharing and cycles are allowed (as in double- or cyclic-lists), there is a tree-like *backbone* used for traversal. For example, even with a singly linked list, our system will reject programs that traverse deep into the list, and return a pointer to a cell *unboundedly* deep inside the list. We believe it is possible to exploit the connection made between the SL notion of "magic wands" and the type-theoretic notion of "zippers" [18] identified in [34] to enrich the alias typing discipline to accommodate such access patterns.

Table 2. Experimental results (Inference). For each procedure listed we compare the number of tokens used to specify: **ART** Type refinements for the top-level procedure in Art; **ART** Annot manually-provided predicate templates required to infer the necessary types [32]; **VCDryad** Spec pre- and post-conditions of the corresponding top-level VCDryad procedure; and **VCDryad** Annot loop invariants as well as the specifications required for intermediate functions in VCDryad. Art Annot totals include only *unique* predicate templates across benchmarks.

Data Structure	Procedure	ART		VCDryad	
		Specification	Annotation	Specification	Annotation
Singly Linked List	(definition)	34	-	31	-
	`rev`	5	0	11	15
Sorted Linked List	(definition)	38	-	50	-
	`rev`	11	9	17	15
	`double`	0	4	7	54
	`pairwiseSum`	0	4	13	75
	`insSort`	5	0	20	17
	`mergeSort`	5	18	18	79
	`quickSort`	5	18	11	140
Binary Search Tree	(definition)	58	-	55	-
	`del`	7	32	20	33
	Total	168	63	253	428

5 Related Work

Physical Type Systems. Art infers logical invariants in part by leveraging the technique of alias typing [2,38], in which access to dynamically-allocated memory is factored into references and capabilities. In [8,29], capabilities are used to decouple references from regions, which are collections of values. In these systems, algebraic data types with an ML-like "match" are used to discover spatial properties, rather than `null` pointer tests. `fold` &`unfold` are directly related to roll &unroll in [38]. These operations, which give the program access to quantified heap locations, resemble reasoning about capabilities [29,35]. These systems are primarily restricted to verifying (non-)aliasing properties and finite, non-relational facts about heap cells (*i.e.* "typestates"), instead of functional correctness invariants. A possible avenue of future work would be to use a more sophisticated physical type system to express more data structures with sharing.

Logical Type Systems. Refinement types [20,25,39], encode invariants about recursive algebraic data types using indices or refinements. These approaches are limited to *purely functional* languages, and hence cannot verify properties of linked, mutable structures. Art brings logical types to the imperative setting by using [38] to structure and reason about the interaction with the heap.

Interactive Program Logics. Several groups have built interactive verifiers and used them to verify data structure correctness [12,41]. These verifiers require

the programmer write pre- and postconditions and loop invariants in addition to top-level correctness specifications. The system generates verification conditions (VCs) which are proved with user interaction. [19] uses symbolic execution and SMT solvers together with user-supplied tactics and annotations to prove programs. [10,24] describe separation logic frameworks for Coq and tactics that provide some automation. These are more expressive than Art but require non-trivial user assistance to prove VCs.

Automatic Separation Logics. To automate the proofs of VCs (*i.e.* entailment), one can design decision procedures for various fragments of SL, typically restricted to common structures like linked lists. [4] describes an entailment procedure for linked lists, and [6,14,16] extend the logic to include constraints on list data. [5,21,27,28] describe SMT-based entailment by reducing formulas (from a list-based fragment) to first-order logic, combining reasoning about shape with other SMT theories. The above approaches are not extensible (*i.e.* limited to list-segments); other verifiers support user defined, separation-logic predicates, with various heuristics for entailment [9,11]. Art is related to natural proofs [26,30] and the work of Heule et al. [17], which instantiate recursive predicates using the local footprint of the heap accessed by a procedure, similar to how we insert `fold` and `unfold` heap annotations, enabling generalization and instantiation of structure properties. Finally, heap binders make it possible to use recursive functions (e.g. measures) over ADTs in the imperative setting. While our measure instantiation [20] requires the programmer adhere to a typing discipline, it does not require us to separately prove that the function enjoys special properties [36].

Inference. The above do not deal with the problem of inferring annotations like the inductive invariants (or pre- and post- conditions) needed to generate appropriately strong VCs. To address this problem, there are several abstract interpreters [22] tailored to particular data structures like list-segments [40], lists-with-lengths [23]. Another approach is to combine separate domains for heap and data with widening strategies tailored to particular structures [7,15]. These approaches conflate reasoning about the heap and data using monolithic assertions or abstract domains, sacrificing either automation or expressiveness.

Acknowledgement. This work was supported by NSF grants CCF-1422471, C1223850, CCF-1218344, and a generous gift from Microsoft Research.

References

1. https://github.com/UCSD-PL/nano-js/tree/vmcai_2016/tests/eval
2. Ahmed, A., Fluet, M., Morrisett, G.: L^3: a linear language with locations. Fundam. Inf. **77**(4), 397–449 (2007)
3. Bakst, A., Jhala, R.: Predicate abstraction for linked data structures. http://arxiv.org/abs/1505.02298
4. Berdine, J., Calcagno, C., O'hearn, P.W.: Smallfoot: modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006)

5. Botinčan, M., Parkinson, M., Schulte, W.: Separation logic verification of c programs with an smt solver. ENTCS **254**, 5–23 (2009)
6. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: Accurate invariant checking for programs manipulating lists and arrays with infinite data. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 167–182. Springer, Heidelberg (2012)
7. Chang, B.E., Rival, X.: Relational inductive shape analysis. In: POPL (2008)
8. Charguéraud, A., Pottier, F.: Functional translation of a calculus of capabilities. In: Hook, J., Thiemann, P. (eds.), Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP 2008), pp. 213–224. ACM (2008)
9. Chin, W.-N., David, C., Nguyen, H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. Comput. Program. **77**(9), 1006–1036 (2012)
10. Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic. In: PLDI. ACM (2011)
11. Chu, D.-H., Jaffar, J., Trinh, M.-T.: Automatic induction proofs of data-structures in imperative programs. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15–17, 2015, pP. 457–466 (2015)
12. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
13. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
14. Dudka, K., Peringer, P., Vojnar, T.: Predator: a practical tool for checking manipulation of dynamic data structures using separation logic. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 372–378. Springer, Heidelberg (2011)
15. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL, pp. 235–246 (2008)
16. Haase, C., Ishtiaq, S., Ouaknine, J., Parkinson, M.J.: SeLoger: a tool for graph-based reasoning in separation logic. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 790–795. Springer, Heidelberg (2013)
17. Heule, S., Kassios, I.T., Müller, P., Summers, A.J.: Verification condition generation for permission logics with abstract predicates and abstraction functions
18. Gérard, P.: Huet. The zipper. J. Funct. Program. **7**(5), 549–554 (1997)
19. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011)
20. Kawaguchi, M., Rondon, P., Jhala, R.: Type-based data structure verification. In: PLDI (2009)
21. Lahiri, S.K., Qadeer, S.: Back to the future: revisiting precise program verification using smt solvers. In: POPL (2008)
22. Lev-Ami, T., Sagiv, M.: TVLA: a system for implementing static analyses. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 280–301. Springer, Heidelberg (2000)

23. Magill, S., Tsai, M.-H., Lee, P., Tsay, Y.-K.: THOR: a tool for reasoning about shape and arithmetic. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 428–432. Springer, Heidelberg (2008)
24. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: reasoning with the awkward squad. In: ICFP (2008)
25. Nystrom, N., Saraswat, V., Palsberg, J., Grothoff, C.: Constrained types for object-oriented languages. In: OOPSLA. ACM (2008)
26. Pek, E., Qiu, X., Madhusudan, P.: Natural proofs for data structure manipulation in c using separation logic. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 46. ACM (2014)
27. Navarro Pérez, J.A., Rybalchenko, A.: Separation logic+ superposition calculus = heap theorem prover. In: PLDI (2011)
28. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 773–789. Springer, Heidelberg (2013)
29. Pottier, F., Protzenko, J.: Programming with permissions in mezzo. In: ICFP (2013)
30. Qiu, X., Garg, P., Stefanescu, A., Madhusudan, P.: Natural proofs for structure, data, and separation. In: PLDI (2013)
31. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: LICS (2002)
32. Rondon, P., Kawaguchi, M., Jhala, R.: Liquid types. In: PLDI (2008)
33. Rondon, P., Kawaguchi, M., Jhala, R.: Low-level liquid types. In: POPL (2010)
34. Schwerhoff, M., Summers, A.J.: Lightweight support for magic wands in an automatic verifier. In: 29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5–10, 2015, Prague, Czech Republic, pp. 614–638 (2015)
35. Sunshine, J., Naden, K., Stork, S., Aldrich, J., Tanter, E.: First-class state change in plaid. In: OOPSLA (2011)
36. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. In: POPL (2010)
37. 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)
38. Walker, D., Morrisett, J.G.; Alias types for recursive data structures. In: Types in Compilation (2000)
39. Xi, H., Pfenning, F.: Dependent types in practical programming. In: POPL (1999)
40. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'hearn, P.W.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)
41. Zee, K., Kuncak, V., Rinard, M.C.: Full functional verification of linked data structures. In: PLDI, pp. 349–361 (2008)