Predicate Abstraction
for Linked Data Structures

Alexander Bakst and Ranjit Jhala
University of California, San Diego
{abakst,jhala}@cs.ucsd.edu

Abstract. We present Alias Refinement Types (Art), a new approach to 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. Predicate abstraction would be an attractive abstract domain for performing invariant inference, existing techniques are not able to reason about the heap with enough precision to verify functional properties of data structure manipulating programs. In this paper, we propose 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 prove Art sound by translating types into separation logic assertions, thus translating typing derivations in Art into separation logic proofs. We evaluate Art by implementing a tool that performs type inference for an imperative language. We use the tool to infer functional correctness properties of the implementations of user-defined data structures, such as singly and doubly linked lists, cyclic lists, heaps 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.

1 Introduction

Separation logic (SL) [27] 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 [217], and design custom decision procedures [11,13,23,24] or abstract interpretations [19,34,5]. 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 [20,8]. Even
when entailment is automated by specializing proof search, the programmer has the onerous task of providing complex auxiliary inductive invariants \[7, 26\].

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, 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 sortedness 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 an decidable logic, e.g., *nat* \( \vdash \{ \nu : \text{int} \mid 0 \leq \nu \} \) 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 \( 0 \leq \nu \) is the logical part. A *template* is a refinement type where, instead of concrete formulas we have
variables $\kappa$ that denote the unknown to-be-inferred refinements. In the case that the refinement is simply true, we omit the refinement (e.g. $\text{int} = \{\nu : \text{int} \mid \text{true}\}$). We specify the behaviors of functions using refined function types: $(x_1 : t_1, \ldots, x_n : t_n) \to t$. 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 $\kappa$, and (2) constraint solving, which uses an off the shelf predicate abstraction based Horn clause solver [28] 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 $(\text{int}) \Rightarrow \{\nu : \text{int} \mid \kappa_1\}$ where $\kappa_1$ denotes the unknown output refinement. (We write $\text{nat}$ in the figure to connect the inferred refinement with its $\kappa$.) 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 $\{\nu : \text{int} \mid \nu = x\}$ 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:

\[
x : \text{int}, 0 \leq x \vdash \{\nu : \text{int} \mid \nu = x\} \preceq \{\nu : \text{int} \mid \kappa_1\}
\]

\[
x : \text{int}, (0 \leq x), r : \{\nu : \text{int} \mid \nu = 0 - x\} \vdash \{\nu : \text{int} \mid \nu = r\} \preceq \{\nu : \text{int} \mid \kappa_1\} \tag{1}
\]

which respectively reduce to the Horn implications

\[
(\text{true} \land 0 \leq x) \Rightarrow (\nu = x) \Rightarrow \kappa_1
\]

\[
(\text{true} \land (0 \leq x) \land r = 0 - x) \Rightarrow (\nu = r) \Rightarrow \kappa_1
\]

By predicate abstraction [28] we find the solution $\kappa_1 \equiv 0 \leq \nu$ 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 absolute

---

**Fig. 1** Refinement Types

```plaintext
 abs :: (int) ⇒ nat

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. 2** Strongly Updating a Location

```plaintext
 absR :: (x:(data:int)) ⇒ ()/x⇒ (data:nat)

function absR(x){
    x0 = x:(κx)
    κ0 = κx ⇒ (data:int)
    var d = x.data; I0 = d:int; I0
    var t = abs(d); I1 = t: nat; I1
    x.data = t; Σ1 = κx ⇒ (data:ν = t)
    return;
}
```

---
value. We use $\kappa_2$ for the output refinement; hence the type of $\text{absR}$ desugars to: $(x:\&x)/\&x \rightarrow \langle \text{data: int} \rangle \Rightarrow ()/\&x \rightarrow \langle \text{data: } \kappa_2 \rangle$ which states that $\text{absR}$ 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 returns () (i.e. no value) in an output heap where the location $\&x$ is updated to a structure with a $\kappa_2$-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 $\Gamma$ and heap $\Sigma$ 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 $\text{abs(d)}$ returns a $\kappa_1$ that is bound to $t$, which is then used to strongly update the data field of $\&x$ from int to $\kappa_1$. 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:

$x: \langle \&x, d : \text{int}, t: \kappa_1 \mid \&x \rightarrow \langle \text{data: } \nu = t \rangle \leq \&x \rightarrow \langle \text{data: } \kappa_2 \rangle$

which reduces by field subtyping to the implication: $\kappa_1[t/\nu] \Rightarrow (\nu = t) \Rightarrow \kappa_2$ which (together with the previous constraints) can be solved to $\kappa_2 \vdash 0 \leq \nu$ letting us infer that $\text{absR}$ updates the structure to make data non-negative. This is possible because the $\kappa$ 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** can be described as iso-recursive alias types [82]. The definition

$$\text{type list}[A] \equiv \exists ! \ell \rightarrow t: \text{list}[A]. h: \langle \text{data: } A, \text{next: } ?(\ell) \rangle$$

says 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 $?(\ell)$ type. The heap $\ell \rightarrow t: \text{list}[A]$ denotes that a singleton list$[A]$ is stored at the location denoted
by ℓ if it is reachable at runtime. The ∃! quantification means that the tail is distinct from every other location, ensuring that the list is inductively defined.

Consider absL from Fig. 8 which updates each data field of a list with its absolute value. As before, we start by creating a κ₃ for the unknown output refinement, so the function gets the template

\[(x:⟨&x⟩)/&x \mapsto x₀: list[∩] ⇒ ()/&x \mapsto xᵣ: list[κ₃]\]

Fig. 8 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 (§ E).

**Unfold** The location &x that the variable x refers to initially contains a list[∩] named with a heap binder x₀. The binder x₀ 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(kx) operation that unfolds the list at &x from &x \mapsto x₀: list[∩] to

\[&x \mapsto x₁: ⟨\text{data: int, next: ?⟨&t⟩}⟩ * &t \mapsto t₀: list[∩]\]
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 x₂ for the updated structure whose data field is a κ₁, 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[κ₃] – effectively computing a summary of the structure rooted at &x. As xn is null and xn: {ν: ?⟨&t⟩ | ν = x₂.next}, fold(kx) converts &x \mapsto x₂: ⟨data: κ₁, next: ?⟨&t⟩⟩ to &x \mapsto list[κ₃] after generating a heap subtyping constraint which forces the “head” structure to be a subtype of the folded list’s “head” structure.

\[Γ₃ ⊢ &x \mapsto x₂: ⟨\text{data: κ₁, next: ?⟨&t⟩}⟩ \leq &x \mapsto x₂: ⟨\text{data: κ₃, next: ?⟨&t⟩}⟩\]

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[κ₃]. As xn and hence the next field of x₂ is non-null, the fold(kx) transforms

\[&x \mapsto x₂: ⟨\text{data: κ₁, next: ?⟨&t⟩}⟩ * &t \mapsto t₁: list[κ₃]\]

into &x \mapsto list[κ₃], as required at the return, by generating a heap subtyping constraints for the head and tail:

\[Γ₅ ⊢ &x \mapsto x₂: ⟨\text{data: κ₁, next: ?⟨&t⟩}⟩ \leq &x \mapsto x₂: ⟨\text{data: κ₃, next: ?⟨&t⟩}⟩ \leq &t \mapsto t₁: list[κ₃]\]

The constraints eq. (2), eq. (6) and eq. (4) are simplified field-wise into the implications κ₁ \Rightarrow κ₃, κ₁ \Rightarrow κ₃ and κ₃ \Rightarrow κ₃ which, together with the previous constraints (eq. (1)) solve to: κ₃ = 0 ≤ ν. 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 \cite{29} 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 \cite{27}. Consider the heap $\Sigma$ defined as:

$$\Sigma_{x_0} = \langle \text{data} = 0, \text{next} = \&x_1 \rangle * \&x_1 \rightarrow t: \langle \text{data} = 1, \text{next} = \text{null} \rangle$$

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

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

$$(x:(&x)) & x \rightarrow x_0 : \text{list}[\text{int}] \Rightarrow () & x \rightarrow x_r : \text{list}[\text{nat}]$$

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

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

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 [10]. 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 [31].

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 accommodate the new structure containing $k$. We generate a fresh $\kappa_4$ for the output type to obtain the function template:

$$(A, x : \langle &x \rangle) & x \rightarrow x_0 : \text{list}[A] \Rightarrow \langle &\ell \rangle & \ell \rightarrow \{\nu : \text{list}[A] \mid \kappa_4\}$$

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 $\kappa_4$.

Constraint generation proceeds by additionally instantiating measures at each fold and unfold. When $x$ is null, the fold($\&y$) transforms the binding $&y \rightarrow y_0 : \langle \text{data} : A, \text{next} : \text{null} \rangle$ into a (singleton) list $&y \rightarrow y_1 : \text{list}[A]$ and so we add the instantiation constraint $\text{len}(y_1) = 1$ to the environment. Hence, the subsequent return yields a subtyping constraint over the output list that simplifies to the implication:

$$\text{len}(x_0) = 0 \land \text{len}(y_1) = 1 \Rightarrow \nu = y_1 \Rightarrow \kappa_4$$  \hspace{1cm} (5)

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

$$\&x \rightarrow x_1 : \langle \text{data} : a, \text{next} : \langle &t \rangle \ast &t \rightarrow t_0 : \text{list}[A] \rangle$$

yielding the instantiation constraint $\text{len}(x_0) = 1 + \text{len}(t_0)$ that relates the length of the list’s snapshot with that of its tail’s. When $x \ll x.\text{data}$ 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:

\[
\text{len}(x_0) = 1 + \text{len}(t_0) \land \text{len}(x_2) = 1 + \text{len}(t_0) \land \text{len}(y_3) = 1 + \text{len}(x_2) \Rightarrow \nu = y_3 \Rightarrow \kappa_4
\]

(6)

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

\[
\text{len}(x_0) = 1 + \text{len}(t_0) \land \kappa_4[p, x_0/u_0, t_0] \land \text{len}(x_2) = 1 + \text{len}(u_0) \Rightarrow \nu = x_2 \Rightarrow \kappa_4
\]

(7)

The recursive call that returns \(u_0\) constrains it to satisfy the unknown refinement \(\kappa_4\) (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. (3) to (6) to \(\kappa_4 \hat{=} \text{len}(\nu) = 1 + \text{len}(x_0)\) inferring a signature that states that insert’s output has size one more than the input.

3 Type Inference

Type Environments We describe ART in terms of an in imperative language Imp with record types and with the usual call by value semantics, whose syntax is given in Fig. 5. A function environment \(\Phi\) be defined as a mapping, \(\Phi\), from functions \(f\) to function schemas \(S\). A type environment \((\Gamma)\) is a sequence of type bindings \(x:T\) and guard expressions \(e\). A heap \((\Sigma)\) is a finite, partial map from locations \((\ell)\) 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 \(\Phi\), 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 \(\Gamma\) and \(\Sigma\), \(s\) produces the output environments \(\Gamma'\) and \(\Sigma'\). We say that a program \(p\) typechecks with respect to \(\Phi\) if, for every function \(f\) defined in \(p\), \(\Phi \vdash f :: \Phi(f)\).

Refinement Inference Let \(\hat{\Phi}\) denote a function environment as before except each type appearing in \(\Phi\) is optionally of the form \(\{\nu: \tau \mid \kappa_i\}\), i.e. its refinement has been omitted and replaced with a unique \(\kappa\) variable. To infer the refinements
CGen : FunEnv × TypeEnv × HeapEnv × Stmt → {Constr} × TypeEnv × HeapEnv

CGen(Φ, Γ, Σ, s) = match s with
| x = e → let (cs, t) = CGEx(Γ, Σ, e) in (cs, x:(Shape(t) ⊨ (v = e))):Γ, Σ
| y = x.f → let ℓ = loc(Γ(x)) in ((Γ ⊨ Γ(x) ≤ ℓ), y:TypeAt(Σ, ℓ)):Γ, Σ
| x.f = e → let (cs, t) = CGEx(Γ, Σ, e)
  ℓ = Loc(t)
  (y:T_y, z) = (Σ(ℓ), FreshId())
  ht = NameFields(z, T_y[f : Shape(t) ⊨ (v = e)])
  in (cs ∪ {Γ ⊨ t ≤ ℓ}, Γ, Σ[ℓ ← (t, ht)])
| x = alloc{es} → let (ℓ, z) = (FreshLoc(), FreshId())
  (cs, fts) = Ω{(c,f,t) | (c, t) = CGEx(Γ, Σ, e) ∧ f:e ∈ es}
  in (cs, z:ℓ;Γ, ℓ → z:NameFields(z, {fts});Σ)

Fig. 6: Statement constraint generation

denoted by each κ we extract a system of Horn clause constraints C by performing refinement type checking with respect to Φ for each function defined in the program p. 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_i, is valid. We solve the constraints by abstract interpretation in the predicate abstraction domain generated from user-supplied predicate templates. For more details, we refer the reader to [28]. We thus infer the refinements missing from Φ by finding such a solution, if it exists.

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 outputs (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 are either well-formedness constraints, Γ, Σ ⊢ T, which limit the names that may appear in T to those appearing in Γ and Σ, or subtyping constraints, Γ ⊢ T ≤ T’. Base subtyping constraints Γ ⊢ {ν:b | p} ≤ {ν:b | q} correspond to the (Horn) Constraint [Γ] = p = q, where [Γ] is the conjunction of all of the refinements appearing in Γ [28]. Heap Subtyping constraints Γ ⊢ Σ ≤ Σ’ 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.

Basic Statements yield constraints as shown in Fig. 6. The assignment case (x = e) determines the type of the expression e with the helper CGenExpr. CGen returns constraints generated by CGenExpr with the input environment (Γ) ex-
### Fig. 7: Compound statement and function call constraint generation

```plaintext
let (cs, $\Gamma$, $\Sigma$) = CGen($\Phi, \Gamma, \Sigma, s1$)
\hspace{1em} let (cs, $\Gamma$, $\Sigma$) = CGen($\Phi, \Gamma, \Sigma, s2$)
\hspace{1em} in (cs, $\Gamma$, $\Sigma$)
\hspace{1em} if e then s1 else s2
\hspace{1em} let (csg, $\Gamma$) = CGEx($\Gamma, \Sigma, e$)
\hspace{1em} \((cs_1, \Gamma_1, \Sigma_1), (cs_2, \Gamma_2, \Sigma_2)) = (\text{CGen}(\Phi; e; \Gamma, \Sigma), \text{CGen}(\Phi; \neg e; \Gamma, \Sigma))\)
\hspace{1em} $\Gamma'' = \text{Fresh}(\text{Shape}((x: T_1 | x: T_1 \in \Gamma_1 \land x: T_2 \in \Gamma_2)))$
\hspace{1em} $\Sigma'' = \text{Fresh}(\text{Shape}(\text{FreshBinders}(\Sigma_1)))$
\hspace{1em} $\text{cs} = (\text{cs}, \Gamma, \Sigma) \\ \hspace{1em} \cup \{ \Gamma_1 \vdash \Sigma_1 \leq \Sigma'' \} \\ \hspace{1em} \cup \{ \Gamma_1 \vdash \Sigma_2 \leq \Sigma'' \} \\ \hspace{1em} \cup \{ \Gamma_2 \vdash T \leq \Gamma' \} \\ \hspace{1em} \cup \{ \Gamma_2 \vdash T \leq \Gamma' \}$
\hspace{1em} in (cs, $\Gamma'$, $\Sigma''$)
\hspace{1em} return e
\hspace{1em} let (cs, t) = CGEx($\Gamma, \Sigma, e$)
\hspace{1em} $\Sigma'' = \text{Frame}(\Sigma, \Sigma_1)$
\hspace{1em} $\text{in } \{(\{\{\Gamma \vdash t \leq e/\text{tr}[\text{tr}]\} \cup \{\Gamma \vdash \Sigma'' \leq \Sigma_1\}, \text{false, emp})$\}
\hspace{1em} let (xs, fts, x: T, $\Sigma_{in}, \Sigma_{out}$) = BreakFunSchema(Inst($\Phi(f)$))
\hspace{1em} $\Sigma_{in}, \Sigma_{out} = \text{Frame}(\Sigma, \Sigma_{in})$
\hspace{1em} $\theta$, exs = (MakeSubst(xs $\rightsquigarrow$ es), Combine(es, xs))
\hspace{1em} $\text{cs} = \bigcup \{ \{ \Gamma \vdash T \leq \theta \Sigma_{in} \} \}$
\hspace{1em} in (cs, $x: \theta T, \Gamma$, $\Sigma_{in} \theta \Sigma_{out}$)
```
| \text{pad}(\ell) \rightarrow \text{let} \ (x, T) = (\text{FreshId}(), \text{FreshType}()) \\
| \text{in} \ ((\Gamma, \ell \mapsto x : T \mapsto \Sigma) \mapsto T, \Gamma, \ell \mapsto x : T \mapsto \Sigma) \\
| \text{concr}(x) \rightarrow \text{let} \ (z, \ell) = (\text{FreshId}(), \text{Loc}(\Gamma(x))) \\
| \ (y, T_z) = \Sigma(\ell) \\
| \text{in} \ ((\Gamma \vdash \Gamma(x) \leftarrow z: T \mapsto \Sigma(\ell), y : T_y : \Sigma[\ell \mapsto \text{Shape}(T_y) \cap \{v = y\}]) \\
| \text{unfold}(\ell) \rightarrow \text{let} \ (x_\ell, T_c, \Sigma_c) = \text{FreshIds}(\text{FreshLocs}(\text{Expand}(\Gamma, C[\ell]))) \\
| \ (\varepsilon, [\nu] \text{InstM}(C, x_\ell, \Sigma_c) = \Gamma, \Sigma[\ell \mapsto x_\ell : T_c] \mapsto \Sigma_c) \\
| \text{fold}(\ell) \rightarrow \text{let} \ (x_\ell, T) = \Sigma(\ell) \\
| \ (y, T) = (\text{FreshId}(), \text{FreshType}()) \\
| \ (x_\ell, T_c, \Sigma_c) = [\ell/\alpha] \text{InstNames}(x_\ell, T_c, \Sigma_c, \text{Expand}(\Gamma, C[\ell])) \\
| \ T_\ell = \{(\nu : C[T] \mid \text{InstMeasures}(C, x_\ell, \Sigma_c)) \\
| \ cs = \{(\Gamma, \Sigma \vdash T) \cup \text{FoldHeaps}(\Gamma, (x_\ell, T_c, \Sigma_c), (x_\ell, T_c, \Sigma_c)) \\
| \ (cs, \Gamma, \ell \mapsto x_\ell : T_c \mapsto \Sigma_c)

\textbf{Fig. 8:} Annotation constraint generation

\begin{align*}
\text{FoldHeaps}(\Gamma, (x_\ell, T_c, \Sigma_c), (x_\ell, T_2, \Sigma_2)) &= \\
\{ \Gamma \vdash T_1 \leq T_2 \} \cup \\
\text{if} \quad \text{locs}(T_1) \cap \text{dom}(\Sigma_1) = \emptyset \quad \text{then} \emptyset \quad \text{else} \\
\text{match} \quad (T_1, T_2) \quad \text{with} \\
| \{(v : \ell \mid p \}, \_\}) & \rightarrow \text{let} \ ((y : T_\ell), (y : T_{\ell_2}) = (\Sigma_1(\ell), \Sigma_2(\ell)) \\
| \text{in} \ \text{FoldHeaps}(\Gamma, (y : T_{\ell_1}, \Sigma_1 \setminus \ell), (y : T_{\ell_2}, \Sigma_2 \setminus \ell)) \\
| \{(v : ? : \ell \mid p \}, \_\}) & \rightarrow \\
| \text{let} \ ((y : T_\ell), (y : T_{\ell_2}) = (\Sigma_1(\ell), \Sigma_2(\ell)) \\
| \text{in} \ \text{FoldHeaps}(\text{add}(x_\ell, \ell), (y : T_{\ell_1}, \Sigma_1 \setminus \ell), (y : T_{\ell_2}, \Sigma_2 \setminus \ell)) \\
\cup \ \text{FoldHeaps}(\text{add}(x_\ell, \ell), (y : T_{\ell_1}, \Sigma_1 \setminus \ell), (y : T_{\ell_2}, \Sigma_2 \setminus \ell)) \\
| \{(v : \{fs1\} \mid p_1\}, \{v : \{fs2\} \mid p_2\}) \rightarrow \\
| \text{let} \ ((x : \Sigma) = (\text{FreshId}(), \text{combine}(fs1, fs2)) \text{ in} \\
\bigcup \{ \text{FoldHeaps}(\Gamma, (x : T_{\ell_1}, \Sigma_1), (x : T_{\ell_2}, \Sigma_2)) \mid \{f1 : T_{\ell_1}, f2 : T_{\ell_2}\} \in f_\ell \}
\end{align*}

\textbf{Fig. 9:} Heap folding constraint generation

to be a subtype of the current function’s return type. \texttt{CGen} uses \texttt{Frame} to partition the current heap into a heap $\Sigma'$ that has the same domain as the current function’s output heap ($\Sigma_r$) and a heap $\Sigma''$ containing the remaining locations, and constrains the actually returned $\Sigma'$ to be a subtype of the expected $\Sigma''$. To constrain function calls ($x = f(es)$), \texttt{CGen} uses \texttt{BreakFunSchema} to unpack the type of the function $f$. \texttt{Frame} is again used to partition the current heap. \texttt{CGen} generates subtyping constraints between the actual and formal parameters, and between the actual input heap $\Sigma_m$ and the formal input heap $\Sigma_m$. The output environment is extended with the output heap $\Sigma_{out}$ and the unmodified $\Sigma_u$.

\textbf{Heap Manipulating Annotations} which use type information to \texttt{materialize} and \texttt{generalize} locations (again lifting that burden from the predicate abstraction) are \texttt{automatically inserted} into the program by a straightforward analysis ($\S$ 8). \texttt{CGen} generates constraints for these statements as shown in Fig. 8. A heap location must be \texttt{materialized} (\texttt{concr}(x)) in order to make use of any of
its invariants, as the location may not be reachable at runtime. The argument \( x \) shows the location is reachable, and we materialize the location by constraining \( x \) to be non-null. The output environment is extended with the type binding, \( y: T_y \) stored at \( \ell \) in the heap. \( \ell \) is remapped in the heap to a new type binding; predicate \( v = y \) preserves the invariants described by \( T_y \). We unfold a recursive structure to make its cells accessible (unfold(\( f \))). This does not generate any constraints, but the output environment is extended by instantiating any measures defined on the data type with the freshly unfolded field names and locations. Dually, when we fold a location (fold(\( f \))) back into a collection, effectively indicating a “summarization” point, \( \text{CGen} \) partitions the heap into two heaps, \( \Sigma_x \) and \( \Sigma_o \), which are reachable from the “root” of the folded structure, \( T_x \). \( \text{CGen} \) generates constraints by calling FoldHeaps to ensure that the current heap may safely be folded up as the type \( C[T] \).

**Heap Folding** We observe that it is safe to fold a heap into another heap when the sub-heap of the former that is reachable from a given type is subsumed by the latter heap. FoldHeaps (Fig. 8) captures this intuition. If a type, \( T \), references no locations, any heap \( H \) may be folded into another heap \( \Sigma' \) (with respect to \( T \)). If \( T \) is a reference to a location \( \ell \), then FoldHeaps generates subtyping constraints and recurses on the type bound at \( \ell \) in \( \Sigma \). If \( T \) is a possibly-null reference, then FoldHeaps makes two recursive calls in strengthened environments, respectively assuming the reference is null and non-null. This strengthening allows the subtyping constraint to make use of reachability. Recall the first fold in \( \text{absl} \), that happens when \( x = \text{null} \). The constraint thus requires the heap subtyping \( \Gamma \vdash \& t \rightarrow \text{list}[\text{int}] \leq \& t \rightarrow \text{list}[\text{nat}] \) only holds when \( x.\text{next} \) is non-null, i.e. when \( \Gamma \) is \( x: \{ \nu: ? \& t \mid \nu = x.\text{next} \}, x = \text{null}, x.\text{next} \neq \text{null} \); which is trivially valid as \( \Gamma \) is inconsistent. In the final case, FoldHeaps recurses on the types stored in the record \( T \).

**Soundness** The constraints output by \( \text{CGen} \) enjoy the following property. Let \((C, \Gamma', \Sigma')\) be the output of \( \text{CGen}(\Phi, \Gamma, \Sigma, s) \). If \( C \) is satisfiable, then there exists some solution \( K \) such that \( K\Phi, K\Gamma, K\Sigma \vdash s :: K\Gamma'/K\Sigma' \), that is, there is a type derivation using the refinements from \( K \). Thus \( K \) yields the inferred program typing \( \Phi \vdash K\Phi \), 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 the following theorems:

---

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 \([\Phi] \vdash \{ \text{Pre}(S) \} \text{Body}(f) \{ \text{Post}(S) \}\)

\( \text{Pre}(S), \text{Post}(S) \) and \( \text{Body}(f) \) 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 $\Phi, \emptyset, \text{emp} \vdash s :: \Gamma \Sigma$, then $\models \Phi \vdash \text{true} s \{\text{true}\}$

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.

The soundness proof proceeds by induction on the typing derivation. We show that the typing rules correspond to separation logic proofs using the appropriate proof rule; see §C for details.

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 first to demonstrate that it is expressive enough to verify a variety of sophisticated properties for linked structures, second, that it provides a significant automation over the state of the art.

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 output 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 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 [28]; 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.

**Automation** To demonstrate the effectiveness of inference, we selected benchmarks from Table 1 that made use of loops and intermediate functions, and then used type inference to infer the intermediate pre- and post-conditions. The results of these experiments is shown in Table 2. We compare ART with VCDryad [22], a VC generator for separation logic, omitting 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 (in the case of 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.

5 Related Work

**Physical Type Systems** ART infers logical invariants in part by leveraging the technique of alias typing [32], in which access to dynamically-allocated memory is factored into references and capabilities. In [625], 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 [32]. These operations, which give the program access to quantified heap locations, resemble reasoning about capabilities [30,25]. 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 [33,21,16], 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 [32] to structure and reason about the interaction with the heap.

**Interactive Separation Logics** Several groups have built interactive verifiers that mechanize separation logic (SL) [27], and used them to verify data structure correctness [35,9]. These verifiers require the programmer write pre- and post-conditions and loop invariants in addition to top-level correctness specifications. The system generates verification conditions (VCs) which are proved with user interaction. [15] uses symbolic execution and SMT solvers together with user-supplied tactics and annotations to prove programs. [20,8] 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. [2] describes an entailment procedure for linked lists, and [11,13,4] extend the logic to include constraints on list data. [17,16,24] 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, but use various heuristics for entailment [7]. ART is related to natural proofs [26,22] and the work of Heule et al. [14], 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 [10] requires the programmer adhere to a typing discipline, it does not require us to separately prove that the function enjoys special properties [31].

**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 [18] tailored to particular data structures like list-segments [34], lists-with-lengths [19]. Another approach is to combine separate domains for heap and data with widening strategies tailored to particular structures [12,5]. These approaches conflate reasoning about the heap and data using monolithic assertions or abstract domains, sacrificing either automation or expressiveness.
References

1. Amal Ahmed, Matthew Fluet, and Greg Morrisett. L$^3$: a linear language with locations. *Fundamenta Informaticae*, 77(4):397–449, June 2007.
2. J. Berdine, C. Calcagno, and P.W. Ohearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137. Springer, 2006.
3. M. Botinčan, M. Parkinson, and W. Schulte. Separation logic verification of c programs with an smt solver. *ENTCS*, 254:5–23, 2009.
4. A. Bouajjani, C. Drăgoi, C. Enea, and M. Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In *ATVA*, 2012.
5. B. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, 2008.
6. Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming (ICFP’08)*, pages 213–224. ACM, 2008.
7. W-N. Chin, C. David, H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9), 2012.
8. A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*. ACM, 2011.
9. E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Snten, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent c. In *TPHOLs*, 2009.
10. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3), 2005.
11. K. Dudka, P. Peringer, and T. Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In *CAV*, 2011.
12. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246, 2008.
13. C. Haase, S. Ishtiaq, J. Ouaknine, and M. J. Parkinson. Seloger: A tool for graph-based reasoning in separation logic. In *CAV*, 2013.
14. Stefan Heule, Ioannis T Kassios, Peter Müller, and Alexander J Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions.
15. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NASA Formal Methods*, pages 41–55. 2011.
16. M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI*, 2009.
17. S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using smt solvers. In *POPL*, 2008.
18. T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *SAS*. 2000.
19. S. Magill, M-H. Tsai, P. Lee, and Y-K. Tsay. Thor: A tool for reasoning about shape and arithmetic. In *CAV*, 2008.
20. A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *ICFP*, 2008.
21. N. Nystrom, V. Saraswat, J. Palsberg, and C. Grothoff. Constrained types for object-oriented languages. In *OOPSLA*. ACM, 2008.
22. Edgar Pek, Xiaokang Qiu, and P Madhusudan. 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, page 46. ACM, 2014.

23. J.A. Navarro Pérez and A. Rybalchenko. Separation logic+ superposition calculus= heap theorem prover. In PLDI, 2011.

24. R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using SMT. In CAV, 2013.

25. F. Pottier and J. Protzenko. Programming with permissions in mezzo. In ICFP, 2013.

26. X. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In PLDI, 2013.

27. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In LICS, 2002.

28. P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In PLDI, 2008.

29. P. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In POPL, 2010.

30. J. Sunshine, K. Naden, S. Stork, J. Aldrich, and E. Tanter. First-class state change in plaid. In OOPSLA, 2011.

31. Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision procedures for algebraic data types with abstractions. In POPL, 2010.

32. D. Walker and J.G. Morrisett. Alias types for recursive data structures. In Types in Compilation, 2000.

33. H. Xi and F. Pfenning. Dependent types in practical programming. In POPL, 1999.

34. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O’Hearn. Scalable shape analysis for systems code. In CAV, pages 385–398, 2008.

35. K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In PLDI, pages 349–361, 2008.
A  Imp with Annotations

A.1 Annotation Elaboration

In § we state that our soundness proof relies on a semantics-preserving elaboration step that inserts assignments to ghost variables. More formally, we can take a typing derivation for a program in Imp as presented in § and produce an elaborated source with an almost identical derivation. These elaborations amount to making assignments to ghost variables to correspond to locations and heap binders, and are thus semantics preserving because the ghost variables do not appear in the original program. The type judgements for the elaborated code are used in the soundness proof. The syntax of Imp with ghost variable annotations is given in Fig. 10. Statement typing judgements with ghost variable annotations are given in Fig. 12.

Expressions \( e ::= n \mid true \mid false \mid null \mid \ell \mid x \oplus e \)

Statements \( s ::= s ; s \mid x = e \mid y = x. f \mid x. f = e \mid \text{if } e \text{ then } s \text{ else } s \)

| return \( e \) | \( x = \text{alloc} (\ell : e) \) | \( x = f(\ell) \) |
|----------|----------------|----------------|
| unfold(\( \ell \)) | fold(\( \ell \)) | concr(x) |
| pad(\( \ell \)) |

Programs \( p ::= \text{function } f(\ell) \{ s \} \)

Primitive Types \( b ::= \text{int} \mid \text{bool} \mid \alpha \mid \text{null} \mid \ell \mid ?(\ell) \)

Types \( \tau ::= b \mid C[\ell] \mid [f:T] \)

Refined Types \( T ::= \nu \cdot \tau \mid p \)

Type Definition \( C ::= C[\alpha] = \exists ! \Sigma. x : [f:T] \)

Heaps \( \Sigma ::= \text{emp} \mid \Sigma * \ell \mapsto x : C[\ell] \mid \Sigma * \ell \mapsto x : [f:T] \)

Function Types \( S ::= \forall \ell, \alpha. (x:T)/\Sigma \Rightarrow \exists \ell'. x':T'/\Sigma' \)

Fig. 10: Syntax of Imp programs and types

B  Well Formedness and Expression Typing

Well-formedness rules are given in this section. Expression typing judgements are given in Fig. 12.
Subtyping

\[
\frac{\text{Valid}(\Gamma) \Rightarrow [p] \Rightarrow [p']) \quad \text{Valid}(\Gamma) \Rightarrow [p] \Rightarrow [p']}{\Gamma \vdash \nu:p \leq \nu:p'} \leq_{\text{B}} \quad \frac{\text{Valid}(\Gamma) \Rightarrow [p] \Rightarrow [p']}{\Gamma \vdash \{\nu:C[p]\} \leq \{\nu:C[p']\}} \leq_{\text{APP}} \quad \frac{\Gamma \vdash \nu:f:T \Rightarrow [p] \leq \nu:f:T' \Rightarrow [p']}{\Gamma \vdash \{\nu:?(f)\} \leq \{\nu:?(f')\}} \leq_{\text{REC}} \quad \frac{\text{Valid}(\Gamma) \Rightarrow [p] \Rightarrow [p']}{\Gamma \vdash \{\nu:\text{null}\} \leq \{\nu:\text{null}\}} \leq_{\text{DOWN}} \frac{\text{Valid}(\Gamma) \Rightarrow [p] \Rightarrow [p']}{\Gamma \vdash \nu:p \leq \nu:p'} \leq_{\text{UP1}} \frac{\text{Valid}(\Gamma) \Rightarrow [p] \Rightarrow [p']}{\Gamma \vdash \{\nu:\text{null}\} \leq \{\nu:?(f)\} \Rightarrow [p']}} \leq_{\text{UP2}} \frac{\Gamma \vdash \nu:p \leq \nu:p'}{\Gamma \vdash \text{emp} \leq \text{emp}} \leq_{\text{EMP}} \frac{\Gamma \vdash \Sigma \leq \Sigma' \quad \Gamma \vdash T \leq T'}{\Gamma \vdash \text{loc}(T_1) \cap \text{Dom}(\Sigma_1) = \emptyset 
\quad \Gamma \vdash T_1 \leq T_2} \leq_{\text{F-BASE}}
\]

Heap Folding

\[
\frac{\text{loc}(T_1) \cap \text{Dom}(\Sigma_1) = \emptyset \quad \Gamma \vdash T_1 \leq T_2}{\Gamma \vdash x:T_1/\Sigma_1 \triangleright x:T_2/\Sigma_2} \quad \frac{\Sigma_1 = \Sigma_1 \triangleright x:T \quad \Sigma_2 = \Sigma_2 \triangleright x:T'}{\Gamma \vdash \nu:?(f) \Rightarrow [p] \leq \Sigma_1 \triangleright x:T_2/\Sigma_2} \quad \frac{\Sigma_1 = \Sigma_1 \triangleright y:T \quad \Sigma_2 = \Sigma_2 \triangleright y:T'}{\Gamma \vdash y:?(f) \Rightarrow [p] \triangleright \Sigma_1 \triangleright y:T_2/\Sigma_2} \quad \Gamma \vdash y:?(f) \Rightarrow [p] \triangleright \Sigma_1 \triangleright y:T_2/\Sigma_2}
\]

Fig. 11: Subtyping, Heap Subtyping, and Heap Folding Rules
Statement Typing

\[ \Phi, \Gamma \vdash s :: \Gamma' / \Sigma' \]

\[ \Phi, \Gamma, \Sigma \vdash e :: \nu : \tau \mid p \]

T-ASSGN

\[ \Phi, \Gamma, \Sigma \vdash x = e :: x : \nu : \tau \mid \nu = e ; \Gamma / \Sigma \]

T-SEQ

\[ \Phi, \Gamma, \Sigma ; s_1 :: \Gamma' / \Sigma' \]

T-PAD

\[ \Phi, \Gamma, \Sigma \vdash \alpha :: \Gamma' / \Sigma' \]

T-IF

\[ \Phi, \Gamma, \Sigma \vdash c :: \text{bool} \]

T-WR

\[ \Phi, \Gamma, \Sigma \vdash \alpha :: \Gamma' / \Sigma' \]

for each \( \alpha \)

\[ \Phi, \Gamma, \Sigma \vdash c :: \text{ann} :: f : \ell \]

T-ALLOC

\[ \Phi, \Gamma, \Sigma \vdash x = z :: \ell \]

T-CONCR

\[ \Phi, \Gamma, \Sigma \vdash \text{unfold} \ell, x, \Sigma \]

T-UNFOLD

\[ \Phi, \Gamma, \Sigma \vdash e :: \ell \]

T-FOLD

\[ \Phi, \Gamma, \Sigma \vdash e :: \ell \]

T-RET

\[ \Phi, \Gamma, \Sigma \vdash f :: S \]

T-CALL

Fig. 12: Annotated Statement Typing Rules. In T-CALL, \( S \) is \( \alpha \)-renamable. Note that the well-formedness check requires all output binders to be fresh in the calling context. We assume that type definitions (and, hence, measures over these definitions) \( \Gamma \vdash \Phi, \Gamma ; \Sigma \) are \( \alpha \)-convertible.
Well Formed Types

\[ \Gamma, \Sigma \vdash T \]

- \( \text{WF-PRIM} \)
  - \( \text{FV}(p) \subseteq \text{FV}(\nu:b; \Gamma) \cup \text{FV}(\Sigma) \quad p \text{ is well-sorted in } \nu:b; \Gamma \text{ and } \Sigma \)
  - \( \Gamma, \Sigma \vdash [\nu:b \mid p] \)

- \( \text{WF-APP} \)
  - \( \text{length}(\mathtt{t}) = \text{length}([\nu:b \mid p_i]) \quad \Gamma, \Sigma \vdash [\nu:b \mid p_i] \quad q \)
  - \( \Gamma, \Sigma \vdash [\nu:C([\nu:b \mid p_i]) \mid q] \)

- \( \text{WF-REC} \)
  - \( \Gamma, \Sigma \vdash [\nu:b \mid p_i] \quad \Gamma, \Sigma \vdash [\nu:f_i;[\nu:b \mid p_i] \mid q] \quad f_i = f_i' \Rightarrow i = i' \)

**Fig. 13:** Well Formed Types

Well Formed Type Definitions

- \( \text{WF-DEF} \)
  - \( x \notin \text{FV}(\Sigma) \)
  - \( \Sigma \vdash T \quad x: \langle f: T \rangle \mid p, \Sigma \vdash \Sigma \)
  - \( \text{TypeVars}(\Sigma) \cup \text{TypeVars}([\nu:f; T \mid p]) = \pi \)
  - \( \vdash C[\pi] = \exists! \Sigma. x: \langle f; T \rangle \mid p \)

**Fig. 14:** Well Formed Type Definitions

Well Formed Heaps and Worlds

- \( \text{WF-EMP} \)
  - \( \ell \notin \text{Dom}(\Sigma') \quad x \notin \text{Bound}(\Sigma') \cup \text{Bound}(\Gamma) \)
  - \( \Gamma, \Sigma \vdash \Gamma, \Sigma \vdash \Sigma' \)
  - \( \Gamma \vdash \ell \Rightarrow x: T \star \Sigma' \)

- \( \text{WF-BIND} \)
  - \( \Gamma, \Sigma \vdash \Sigma \)
  - \( \Gamma, \Sigma \vdash \Gamma, \Sigma \vdash \Sigma' \)
  - \( \Gamma \vdash x: T / \Sigma' \)

**Fig. 15:** Well Formed Heaps and Worlds

Well Formed Schemas

- \( \text{WF-FUN} \)
  - \( \forall j. x_j \notin \text{FV}(\Gamma) \cup \text{FV}(\Sigma) \quad x_j: T_j; \Gamma, \Sigma \vdash T_j \)
  - \( x_j: T_j; \Gamma, \Sigma \vdash \Sigma_i \quad x_j: T_j; \Gamma \vdash x_o: T_o / \Sigma_o \quad x_o \notin \text{FV}(\Gamma) \cup \text{FV}(\Sigma) \)
  - \( \text{FV}(\Sigma_o) \subseteq \text{FV}(\Gamma) \cup \text{FV}(\Sigma) \)
  - \( \Gamma, \Sigma \vdash \forall \ell, \alpha. (x_j: T_j) / \Sigma_i \Rightarrow \exists \ell, \alpha. x_o: T_o / \Sigma_o \)

**Fig. 16:** Well Formed Schemas
Well Formed Measures

\[ C[\alpha] \vdash_M m(x) = e \]

\[ \frac{e \in T^*}{\Gamma \vdash_M e : T^*} \quad \frac{x : \langle \ldots f : T_f \ldots \rangle \in \Gamma}{\Gamma \vdash_M x.f : T_f} \quad \frac{f : T \Rightarrow T'' \in \Gamma}{\Gamma \vdash_M f(e) : T''} \]

\[ \frac{\oplus : T \times T' \Rightarrow T'' \in \Gamma \quad \Gamma \vdash_M e : T \quad \Gamma \vdash_M e' : T'}{\Gamma \vdash_M e \oplus e' : T''} \]

\[ \frac{\Gamma \vdash_M e : \text{bool} \quad \Gamma \vdash_M e' : T \quad \Gamma \vdash_M e'' : T}{\Gamma \vdash_M \text{if } e \text{ then } e' \text{ else } e'' : T} \]

\[ \vdash C[s] \equiv \exists \Sigma. x : T \quad T_i = \text{SnapType}(T, \Sigma) \]

\[ m : T_o \Rightarrow T \in \Gamma \quad x : T_v \vdash_M e : T \quad x : \text{null} \vdash_M e : T \]

\[ C[\alpha] \vdash_M m(x) = e \]

Fig. 17: Well Formed Measures

Expression Typing

\[ \Gamma, \Sigma \vdash e : T \]

\[ \frac{\Gamma, \Sigma \vdash n : \{ \nu : \text{int} \mid \nu = n \}}{\text{T-INT}} \quad \frac{x : \{ \nu : \tau \mid \nu = x \} \in \Gamma}{\text{T-VAR}} \]

\[ \frac{\Gamma, \Sigma \vdash \text{null} : \{ \nu : \text{null} \mid \nu = \text{null} \}}{\text{T-NULL}} \quad \frac{\Gamma, \Sigma \vdash e_1 : \text{int} \quad \Gamma, \Sigma \vdash e_2 : \text{int}}{\text{T-OP}} \]

\[ \frac{\Gamma, \Sigma \vdash \text{null} : \{ \nu : \text{null} \mid \nu = \text{null} \}}{\text{T-NULL}} \quad \frac{\Gamma, \Sigma \vdash e_1 \oplus e_2 : \{ \nu : \text{int} \mid \nu = e_1 \oplus e_2 \}}{\text{T-OP}} \]

\[ \frac{\Gamma, \Sigma \vdash e : \{ \nu : \langle \ell \rangle \mid p \}}{\text{T-REF-INV}} \]

\[ \frac{\Gamma, \Sigma \vdash e : T_1 \quad \Gamma \vdash T_1 < T_2 \quad \Gamma, \Sigma \vdash T_2}{\text{T-SUB}} \]

Fig. 18: Expression Typing Rules

Function Typing

\[ \Phi \vdash f :: S \]

\[ \ell = \text{dom}(\Sigma_i) \quad \ell_o = \text{dom}(\Sigma_o) \cap \text{dom}(\Sigma_i) \]

\[ \emptyset, \text{emp} \vdash S = \forall \ell. \forall \pi. (x : T) / \Sigma_1 \Rightarrow \exists \ell_o, x_o : T_o / \Sigma_o \]

\[ (\ell, \Sigma) = \text{AddHeap}(x : T, \Sigma_i) \]

\[ \Phi, R = x_o : T_o / \Sigma_o ; \Gamma, \Sigma \vdash s : \emptyset \]

\[ \Phi \vdash f :: S \]

Fig. 19: Function Typing Rules
Substitution of Heap Binders

\[ \text{Subst}(\Sigma, \Sigma') = \theta \Sigma' \quad \text{where} \quad \theta_x = \begin{cases} x' & \ell \mapsto x : T \in \Sigma \text{ and } \ell \mapsto x' : T' \in \Sigma' \\ x & \text{otherwise} \end{cases} \]

Fresh Renaming of Worlds

\[ \text{FreshSubst}(x : T / \Sigma) = \text{FreshSubst'}(x, T, \Sigma, \text{emp}) \]
\[ \text{FreshSubst'}(x, T, \text{emp}, \theta) = [x'/x] \cdot \theta \quad x \text{ fresh} \]
\[ \text{FreshSubst'}(x, T, \ell \mapsto y : T_y * \Sigma, \Sigma') = \text{FreshSubst'}(x, T, \Sigma, [\ell'/\ell] \cdot [y'/y] \cdot \theta) \quad \ell', y' \text{ fresh} \]

Strengthening Field Names

\[ \text{NameFields}(\nu : \langle f : \{ \nu_f : \tau_f \mid p_f \} \mid p \rangle) = \{ \nu : \langle f : \{ \nu_f : \tau_f \mid p_f \wedge \nu_f = \text{field}(\nu, f) \} \rangle \mid p \} \]

Function Instantiation

\[ \text{Inst}(\forall \ell. S, \ell : T, \ell, T) = \text{Inst}([\ell'/\ell]S, L, T) \]
\[ \text{Inst}(\forall \alpha. S, \epsilon, T, \ell, T) = \text{Inst}([T/\alpha]S, \epsilon, T) \]
\[ \text{Inst}(S, \epsilon, \epsilon) = S \]

Local Context Strenghtening

\[ \text{AddHeap}(\Gamma, \Sigma) = \begin{cases} (\Gamma, \Sigma) & \Sigma = \text{emp} \\ (y : T; \Gamma' ; \ell \mapsto z : \{ \nu : \tau \mid \nu = y \} \ast \Sigma') & \Sigma = \ell \mapsto y : \{ \nu : \tau \mid p \} \ast \Sigma_0, \\ (\Gamma', \Sigma') & x : \ell \in \Gamma, \text{ and } z \text{ fresh} \\ \end{cases} \]
\[ \text{where} \quad (\Gamma', \Sigma') = \text{AddHeap}(\Gamma, \Sigma_0) \]

Fig. 20: Helper Functions
C  Soundness of ART

D  Soundness

We translate types, environments and heaps into separation logic assertions and hence, typing derivations into separation logic proofs. We illustrate the translation by example and present the key theorems; see §C for details.

D.1 Translation

We define a translation $\cdot$ from types, contexts, and heaps to assertions in separation logic. We translate a typing context $\Gamma, \Sigma$ by conjoining the translations of $\Gamma$ and $\Sigma$. Next, we illustrate the translation of local contexts $\Gamma$ and heaps $\Sigma$ using the `insert` function from Fig. 4. When we describe e.g. a $\Gamma$ at a program point, we are referring to the $\Gamma$ before that program point’s execution.

Local Contexts We translate contexts $\Gamma$ into a conjunction of pure, i.e. heap independent, assertions. The context $\Gamma_0$ before $A$ contains two assumptions: (1) $x$ is a possibly null reference to $\&x$ and (2) $x$ is definitely null. Note that $\&x$ does not denote the application of a function called $\&$; it is a distinct location named $\&x$. We translate $\Gamma_0$ by conjoining the translations of these contents. Note that translation states that if $x$ is not null, then $x$ refers to $\&x$.

$\Gamma_0 \equiv x:?'(\&x), x = \text{null} \quad [\Gamma_0] \equiv (x \neq \text{null} \Rightarrow x = \&x) \wedge (x = \text{null})$

Heaps We interpret heaps as (separated) conjunctions of impure (i.e., heap-dependent assertions) and pure assertions corresponding to the types of the values stored in each location. The heap before $A$ states that if the location exists on the heap, then it refers a valid list, described by $\text{list}(A, \&x, x_0)$.

$\Sigma_0 \equiv \&x \mapsto x_0: \text{list}[A] \quad [\Sigma_0] \equiv (\&x \neq \text{null}) \Rightarrow \text{list}(A, \&x, x_0)$ (8)

Type Predicates We translate the definition of the $\text{list}[A]$ type into a separation logic assertion that characterizes the part of the heap where a $\text{list}[A]$ is stored. For example, for the list type from Fig. 4 we define the type predicate as:

$\text{list}(A, \ell, x) \equiv \text{Snapshot}(\ell, x) \wedge \exists h, t, \ell_i. \text{hd}(A, \ell, h, \ell_i) * ((\ell_i \neq \text{null}) \Rightarrow \text{list}(A, \ell_i, t))$

$\text{hd}(A, \ell, h, \ell_i) \equiv [\ell \mapsto h; \text{data}:A, \text{next}?:\ell_i]$

Intuitively, the conjuncts of the assertion $\text{list}(A, \ell, x)$ say: (1) the snapshot $x$ is a pure value comprising all the records (data and references) that are transitively reachable on the heap starting at $\ell$, and (2) the heap satisfies the “shape” invariants defined by the recursive alias type $\text{list}[A]$. We prove that each well-formed type definition has a translation analogous to the above, allowing us translate heap-locations mapped to recursive types in a manner analogous to eq. 8.

Records The other case is for heap-locations that are mapped to plain records. At $B$, `insert` has allocated a record to which $y$ refers. There, we have

$\Gamma_1 \equiv \Gamma_0, y: \&y \quad [\Gamma_1] \equiv [\Gamma_0] \wedge (y \neq \text{null} \Rightarrow y = \&y) \wedge (y \neq \text{null})$ (10)
i.e. the binding for $y$ is translated to the assertion for a maybe-null pointer plus the fact that $y$ is not-null. The heap before executing $B$ is translated:

\[
\llbracket \Sigma_1 \rrbracket \models \llbracket [\Sigma_0] \ast (\&y \mapsto y_0 \land y_0 = \langle \text{data:<k, next:NULL} \rangle) \rrbracket
\]

### D.2 Soundness Theorem

We prove soundness by showing that a valid typing of a statement $s$ under given contexts can be translated into a valid separation logic proof of $s$ under pre- and post-conditions obtained by translating the contexts. Note that there an (semantics-preserving) elaboration step that, for clarity, we defer to §C which inserts ghost variables to name locations and heap binders.

**Theorem 1. [Typing Translation]**

- If $\Phi, \Gamma, \Sigma \vdash s :: \Gamma' / \Sigma'$ then $\llbracket \Phi \rrbracket \models \{\{ \Gamma, \Sigma \}\} \ s \ \{\{ \Gamma', \Sigma' \}\}$
- If $\Phi \vdash f :: S$ then $\llbracket \Phi \rrbracket \models \{\text{Pre}(S)\} \ \text{Body}(f) \ \{\text{Post}(S)\}$

$\text{Pre}(S), \text{Post}(S)$ and $\text{Body}(f)$ are the translations of the input and output types of the function, the function (body) statement. By virtue of how contexts are translated, the result of the translation is not a vacuous separation logic proof. As a corollary of this theorem, our main soundness result follows:

**Corollary 1. [Soundness]** If $\Phi, \emptyset, \text{emp} \vdash s :: \Gamma / \Sigma$, then $\llbracket \Phi \rrbracket \models \{\text{true}\} \ s \ \{\text{true}\}$

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.

The soundness proof proceeds by induction on the typing derivation. We show that the typing rules correspond to separation logic proofs using the appropriate proof rule. The critical parts of the proof are lemmas that transcribe the rules for subtyping and folding recursive type definitions into separation logic entailments:

**Lemma 1. [Folding]** If $\Gamma \vdash x :: T_1 / \Sigma_1 \triangleright x :: T_2 / \Sigma_2$ then

\[
\llbracket \Gamma \rrbracket \Rightarrow [\ell \mapsto x :: T_1 * \Sigma_1] \Rightarrow \exists FV(\Sigma_2). [\ell \mapsto x :: T_2 * \Sigma_2] \land \exists y. \ \text{Snapshot}(\ell, y)
\]

**Lemma 2. [Subtyping]**

- If $\Gamma \vdash T_1 \leq T_2$ then $\llbracket \Gamma \rrbracket \Rightarrow \llbracket T_1 \rrbracket \Rightarrow \llbracket T_2 \rrbracket$
- If $\Gamma \vdash \Sigma_1 \leq \Sigma_2$ then $\llbracket \Gamma \rrbracket \Rightarrow \llbracket \Sigma_1 \rrbracket \Rightarrow \llbracket \Sigma_2 \rrbracket$

These lemmas effectively translate fold, unfold, and subtyping (Fig. 11) into a combination of “skip” statements and the consequence rule of separation logic.

For example, at $B$, insert is about to return the list that $y$ points to (at location $\&y$). At this point the context is given by the $\Gamma_1$ and $\Sigma_1$ from eqs. (9)
and \(^\text{[11]}\). ART checks that the context satisfies the output type of \texttt{insert} i.e. that \&\(y\) is the root of a valid list whose length equals one greater than (the snapshot of) the input \(x\). It uses subtyping and \texttt{T-FOLD} to obtain at \(C\):

\[
\Gamma_2 = \Gamma_1, \text{len}(y_1) = 1 + \text{len}(	exttt{null}) \quad \Sigma_2 = \Sigma_0 \ast \&y \Rightarrow y_1:\texttt{list}[A]
\]

Using lemma \(^\text{[1]}\) and lemma \(^\text{[2]}\) we have: \([\Gamma_1, \Sigma_1] \Rightarrow [\Gamma_2, \Sigma_2]\). In particular, note that \(y_1\) names the snapshot value corresponding to the (single-celled) list referred to by \(y\), and the measure instantiation due to \texttt{T-FOLD} states that the length of the snapshot is 1 greater than the length of the tail \texttt{null}. The implication follows from the definitions of the length measure, the snapshot predicate and the type predicate. We then use the implication in an instance of the consequence rule.

Soundness of ART follows from the following: (1) data type definitions correspond to an inductively defined set of \textit{pure} snapshot values; (2) heap binders correspond to snapshot values; (3) no data structure is modified without first \textit{unfolding} it, ensuring that each “snapshot” (or “version”) receives a fresh name.

\section*{D.3 Assertion language}

Values in \texttt{Imp} are records, integers (which are also used as addresses), the constant \texttt{null}, or products of integers and records. We assume an intuitionistic interpretation of assertions (and thus \(p \ast \texttt{true} \iff p\)).

\begin{align*}
\textbf{Value} = & \ Z \cup \bigcup \{\texttt{null}\} \\
\bigcup \bigcup \{\texttt{null}\} = & \ \text{Records} \\
\{\texttt{null}\} = & \ \text{Integers (and addresses)} \\
\{\texttt{null}\} = & \ \text{S} = Z \times \emptyset
\end{align*}

\textbf{Expressions} \(E::=\cdots\)

\begin{align*}
| & \langle f:E \rangle \text{ record constant} \\
| & f(E) \text{ UIF application}
\end{align*}

\textbf{Assertions} \(P::=\cdots\)

\begin{align*}
| & \texttt{emp} \text{ empty heap} \\
| & E \Rightarrow E \text{ singleton heap} \\
| & P \ast P \text{ separating conjunction}
\end{align*}

With the following axiom:

\[
(y = \langle \ldots f : e \ldots \rangle) \Rightarrow (P \iff \texttt{field}(y, f)/e\{e\}P)
\]

\section*{D.4 Imp Proof Rules}

In addition to the standard frame rule and consequence rule, we assume the following axioms:

\begin{align*}
\textbf{Proof Rules} \quad F \vdash \{P\} s \{Q\}
\end{align*}

\begin{align*}
\textbf{Concretization} \quad z \notin \texttt{FV}(e) \quad x, z \text{ distinct} \\
\vdash \{x \mapsto e\} \texttt{conc}(x, z) \{x \mapsto e \land z = e\}
\end{align*}
Unfolding

\[ \vdash \{x: P\} \text{unfold}(\ell, \bar{\ell}) \{P\} \]

Folding

\[ x \text{ does not appear free in } P \]

\[ \vdash \{\exists x: P\} \text{fold}(\ell, x) \{P\} \]

Padding

\[ \ell, x \text{ distinct} \]

\[ \vdash \{\text{emp}\} \text{pad}(\ell, x) \{\text{emp} \land \ell = \text{null} \land x = \text{null}\} \]

Assignment

\[ x \notin \text{FV}(e) \]

\[ \vdash \{\text{emp}\} x = e \{x = e\} \]

Conditional

\[ F \vdash \{B \land P\} s_1 \{Q\} \quad F \vdash \{-B \land P\} s_2 \{Q\} \]

\[ F \vdash \{P\} \text{if } B \text{ then } s_1 \text{ else } s_2 \{Q\} \]

Allocation

\[ x, z \notin \text{FV}(e) \]

\[ \vdash \{\text{emp}\} x = _z \text{alloc} \{\text{f}: e\} \{x = \ell \land \ell \neq \text{null} \land x \mapsto z \land z = \langle f: e\rangle\} \]

Access

\[ a, v \text{ distinct vars} \]

\[ \vdash \{a \mapsto \langle f: e\rangle\} v = a.f_1 \{a \mapsto \langle f: e\rangle \land v = e_1\} \]

Mutation

\[ z \notin \text{FV}(v) \]

\[ \vdash \{a \mapsto \langle \ldots f_{i}\ldots\rangle\} a.f_i = _z v \{a \mapsto z \land z = \langle \ldots f_{i}\ldots v\ldots\rangle\} \]

Sequence

\[ F \vdash \{P\} s \{R\} \quad F \vdash \{R\} s' \{Q\} \]

\[ F \vdash \{P\} s; s' \{Q\} \]

Procedure Call

\[ g(x_1 \ldots x_m) \Rightarrow s; \text{return } x_o \]

\[ \{P\} g(x_1 \ldots x_m) \{Q\}, F \vdash \{P\} s; \text{return } x_o \{Q\} \]

\[ F \vdash \{P\} x = g(x_1 \ldots x_m) \{x = x_o \land Q\} \]

Return

\[ \vdash \{[e/x_o]P\} \text{return } e \{P\} \]

D.5 Definitions

Definition 1 (Base type translation).

- \( \llbracket x: \text{int} \rrbracket = \text{int}(x) \)
- \( \llbracket x: \text{null} \rrbracket = (x = \text{null}) \)
- \( \llbracket x: \langle \ell \rangle \rrbracket = ((x = \ell) \land (x \neq \text{null})) \)
- \( \llbracket x: \langle \ell \rangle \rrbracket = ([x: \langle \ell \rangle] \lor [x: \text{null}]) \)
- \( \llbracket x: \langle f_i: T_i \rangle \rrbracket = (x = \langle f_i: \text{field}(x, f_i) \rangle) \)
Definition 2 (Local type binding translation).
\[\llbracket x : \{\nu : b \mid p\} \rrbracket = [x/\nu]p \land [x : b]\]
\[\llbracket x : \{\nu : C[T] \mid p\} \rrbracket = [x/\nu]p\]
\[\llbracket x : \{\nu : \langle f_i : T_i \rangle \mid p\} \rrbracket = [x/\nu]p \land [x : \langle f_i : T_i \rangle] \land \bigwedge_{f_i \in \tau} \llbracket \text{field}(x, f_i) : T_i \rrbracket\]

Definition 3 (Heap type binding translation).
\[\llbracket \ell \mapsto x : \{\nu : C[T] \mid p\} \rrbracket = ((\ell \neq \text{null}) \Rightarrow (\llbracket x : \{\nu : C[T] \mid p\} \rrbracket) \ast c(T, \ell, x)) \land ((\ell = \text{null}) \Rightarrow (x = \text{null}))\]
\[\llbracket \ell \mapsto x : \{\nu : \langle f_i : T_i \rangle \mid p\} \rrbracket = ((\ell \neq \text{null}) \Rightarrow (\llbracket x : \{\nu : \tau \mid p\} \rrbracket) \ast \ell \mapsto x) \land ((\ell = \text{null}) \Rightarrow (x = \text{null}))\]

Definition 4 (Free Variables of $\Gamma, \Sigma$).
\[\text{FV}(\emptyset) = \emptyset\]
\[\text{FV}(x:?\ell; \Gamma) = \{\ell\} \cup \text{FV}(\Gamma)\]
\[\text{FV}(x:\ell; \Gamma) = \{\ell\} \cup \text{FV}(\Gamma)\]
\[\text{FV}(\text{emp}) = \emptyset\]
\[\text{FV}(\ell \mapsto x : T \ast \Sigma) = \{\ell, x\} \cup \text{FV}(\Sigma)\]

Definition 5 (Pure Value Types). In order to show measure well formedness, we extend the language of types with products ($T \times T'$) and unions with null ($T + \text{null}$), and define:
\[\text{SnapType} : \text{Type} \times \text{Heap} \rightarrow \text{Type}\]
\[\text{SnapType}(\langle \ell, \Sigma \rangle) = \begin{cases} \langle \ell \rangle \times \text{SnapType}(T, \Sigma) & \ell \mapsto x : T \in \Sigma \\ \langle \ell \rangle & \ell \notin \text{Dom}(\Sigma) \end{cases}\]
\[\text{SnapType}(?\langle \ell, \Sigma \rangle) = \text{SnapType}(\langle \ell, \Sigma \rangle) + \text{SnapType}(\text{null}, \Sigma)\]
\[\text{SnapType}(T, \Sigma) = T\]

Definition 6 (Pure Values from Type Definitions). For each type $T$ we denote the pure values associated with that type as $T^*$, which we define inductively:
\[\begin{array}{c|c|c|c}
\text{null} & \in & \text{null}^* & e \in \text{Value} \\
\hline
\text{null} & \in & \text{int}^* & e \in \alpha^* \\
\hline
\text{null} & \in & \langle \ell \rangle^* & e \in \langle \ell \rangle^* \\
\hline
\text{null} & \in & ?\langle \ell \rangle^* & e \in ?\langle \ell \rangle^* \\
\hline
\text{null} & \in & \langle f_i : T_i \rangle^* & e \in T_i^* \\
\hline
\langle f_i : e_i \rangle & \in & \langle f_i : T_i \rangle^* & (e_1, e_2) \in (T_1 \times T_2)^* \\
\hline
e & \in & (\text{SnapType}(\langle T_\alpha \rangle) \cup \text{null})^* & e \in (T_1 + T_2)^* \\
\hline
C[\alpha] & \in & \exists! \Sigma. x : T & e \in C[T_{\alpha}]^* \\
\hline
\end{array}\]
Definition 7 (Constraints from Pure Values). We consider elements of $C^*$ as “snapshots” of some heap. We “restore” these snapshots by mapping them to assertions with the functions

\[ \text{Snapshot} : \mathbb{Z} \times \text{Value} \rightarrow \text{Prop} \]
\[ \text{Walk} : \text{Value} \rightarrow \text{Value} \times \mathcal{P}(\mathbb{Z} \times \text{Value}) \]

\[ \text{Snapshot}(\ell, x) \doteq ((\ell \neq \text{null}) \Rightarrow (\ell \mapsto e \otimes (e', e''\rightarrow e'))) \]

where \((e, h) = \text{Walk}(x)\)

and where \text{Walk} is defined as follows:

\[ \text{Walk}(\ell, x) = (\ell : e, \bigcup \Sigma) \quad \text{where} \quad (\ell', h) = \text{Walk}(e) \]
\[ \text{Walk}((e_1, e_2)) = (e_1, \{e_1, e_3\} \cup h) \quad \text{where} \quad (e_3, h) = \text{Walk}(e_2) \]
\[ \text{Walk}(e) = (e, \emptyset) \]

Definition 8 ( Assertions from Type Definitions). Given the definition $C[\alpha] \doteq \exists \Sigma. \text{true}$ define the assertion $c(T_{\alpha}, \ell, x) \doteq (\exists e_{\alpha}, \text{FV}(\Sigma). (\ell \mapsto x_{\alpha} \land [\text{type } e_{\alpha} : [T_{\alpha}/\alpha]T]) \land \text{Snapshot}(\ell, x)$

Definition 9 (Interpretation of Measures). Let $m(x:C[\alpha]) \doteq e$ and $s$ be a mapping from variables to values. Define $\llbracket m(x) \rrbracket_{\text{exp}} \doteq \llbracket e \rrbracket_{\text{exp}}$ where

\[ \llbracket v \rrbracket_{\text{exp}}(s) = v \]
\[ \llbracket x.f \rrbracket_{\text{exp}}(s) = s(x).f \]
\[ \llbracket f(e) \rrbracket_{\text{exp}}(s) = f(\llbracket e \rrbracket_{\text{exp}}(s)) \]
\[ \llbracket e_1 \oplus e_2 \rrbracket_{\text{exp}}(s) = \llbracket e_1 \rrbracket_{\text{exp}}(s) \oplus \llbracket e_2 \rrbracket_{\text{exp}}(s) \]
\[ \llbracket \text{if } e \text{ then } e_1 \text{ else } e_2 \rrbracket_{\text{exp}}(s) = \begin{cases} \llbracket e_1 \rrbracket_{\text{exp}}(s) & \text{if } \llbracket e \rrbracket_{\text{exp}}(s) = \text{true} \\ \llbracket e_2 \rrbracket_{\text{exp}}(s) & \text{otherwise} \end{cases} \]

Definition 10 (Interpretation of type contexts/worlds).

\[ \llbracket \Gamma \rrbracket = \bigwedge_{e \in \Gamma} \bigwedge_{x : e \in \Gamma} \bigwedge_{\text{type } t = T} \]
\[ \llbracket \Sigma \rrbracket = \bigwedge_{\ell \mapsto x : e \in \Sigma} \bigwedge_{\ell \mapsto T} \]
\[ \llbracket \Gamma, \Sigma \rrbracket = \llbracket \Gamma \rrbracket \land \llbracket \Sigma \rrbracket \]

Definition 11 (Interpretation of procedure declarations). Assume that $f(x) \doteq s$. Given $S = \forall \ell. \forall x. (x:T) / \Sigma_i \Rightarrow \exists x_o : T_o / \Sigma_o$.

\[ \text{Pre}(S) = \llbracket x : T \rrbracket \land \llbracket \Sigma_i \rrbracket \]
\[ \text{Post}(S) = \llbracket x_o : T_o \rrbracket \land \llbracket \Sigma_o \rrbracket \]
\[ \text{Body}(f) = s \]

and the variables appearing in Post(S) and not Pre(S) are considered to be modified by f.
Definition 12 (Interpretation of procedure contexts).

\[
\llbracket f : S; \Phi \rrbracket = \{ \text{Pre}(S) \} \ \text{Body}(f) \ \{ \text{Post}(S) \}; \ \llbracket \Phi \rrbracket
\]

D.6 Type Soundness

We assume the following:

1. The set of program variables \((x, y, \text{etc.})\) is disjoint from the set of symbols used to denote locations \((\ell, \ell', \text{etc.})\).
2. All programs are in single static assignment form. The only variables which have more than one static assignment are “phi” variables which are assigned once in each branch of an “if” statement.

Lemma 3. If for every \(x : T \in \Gamma\), \(\Gamma \vdash T\), then the only symbols that appear in \(\llbracket \Gamma \rrbracket\) are variables \((x, y)\), arithmetic and equality symbols and uninterpreted functions, and no variable appears bound twice.

Proof. By induction on the type well-formedness judgement.

Corollary 2. The location \(\ell\) only appears in \(\llbracket \Gamma \rrbracket\) if there exists \(x : T \in \Gamma\) and \(T = \{ \nu : \langle \ell \rangle | p \}\) or \(T = \{ \nu : ?\langle \ell \rangle | p \}\).

Lemma 4. If \(\Gamma \vdash \Sigma\), then for each \(\ell \mapsto x : T \in \Sigma\):

1. No binding \(x : T'\) appears in \(\Gamma\).
2. \(\llbracket \Sigma \rrbracket\) contains exactly one sub-assertion of the form \(\ell \mapsto x\).

Proof. By definition of \(\llbracket \Sigma \rrbracket\) and induction on the well-formedness judgement.

Lemma 5. If \(\Gamma, \Sigma \vdash S\), then the formal arguments or free variables of the output world of \(S\) do not appear free in \(\Gamma\) or \(\Sigma\).

Proof. By the assumptions of \(\text{WF-FUN}\).

Corollary 3. The location \(\ell\) only appears in \(\llbracket \Sigma \rrbracket\) if there exists \(\ell' \mapsto x : T \in \Sigma\) and \(\ell' = \ell\) or \(T = \{ \nu : \langle \ell \rangle | p \}\) or \(T = \{ \nu : ?\langle \ell \rangle | p \}\).

Lemma 6. If \(\vdash_M m(x : C(\pi)) = e\), and \(v \in C^*(A)\), then \(\llbracket [v/x]e \rrbracket_{\text{exp}} \in \text{Values}\).

Proof. By induction on the \(\Gamma \vdash_M e : T\) judgement.

Lemma 7. [Subtyping]

- If \(\Gamma \vdash T_1 \leq T_2\) then \(\llbracket \Gamma \rrbracket \Rightarrow \llbracket T_1 \rrbracket \Rightarrow \llbracket T_2 \rrbracket\)
- If \(\Gamma \vdash \Sigma_1 \leq \Sigma_2\) then \(\llbracket \Gamma \rrbracket \Rightarrow \llbracket \Sigma_1 \rrbracket \Rightarrow \llbracket \Sigma_2 \rrbracket\)

Proof. By induction on the subtyping derivation.

Case \(\Gamma \vdash \{ \nu : \tau | p \} \leq \{ \nu : \tau | p' \}\)

By assumption.
Case $\Gamma \vdash \{\nu;C[T]\mid p_C\} \leq \{\nu;C[T']\mid p'\}$

By assumption,

$$[\Gamma] \land p_C \Rightarrow p'_C$$

and

$$[\Gamma] \land [\nu:T] \Rightarrow [\nu:T']$$

for each $T$, $T'$. By hypothesis and the definition of $[\cdot]$, and by the form of each $T$, each $p$ only occurs positively in $c([\nu;\tau\mid p], \ell, x)$. Destructing $T$ as $\Gamma^\nu$ and $T_1$ as $\Gamma_1^\nu$,

$$[\nu;\{\nu;C[T]\mid p_C\}] = p_C \land \exists \ell. c(T, \ell, \nu)$$

$$\Rightarrow p'_C \land \exists \ell. c(T', \ell, \nu)$$

which is equivalent to $t_\nu \downarrow x \nu: T_1 y | p'_{1\nu}$

Case $\Gamma \vdash \{\nu;f:T\mid p\} \leq \{\nu;f:T\mid p'\}$

Unfolding the definition of $[\cdot]$,

$$[\nu;\{\nu;\langle f:T\rangle\mid p\}] = p \land \nu = \langle f:\text{field}(f, \nu)\rangle$$

By hypothesis,

$$[\Gamma] \Rightarrow [\nu:T] \Rightarrow [\nu:T']$$

for each $T$, $T'$, and applying eq. 12 gives us

$$[\Gamma] \Rightarrow [\text{field}(\nu, f):T] \Rightarrow [\text{field}(\nu, f):T']$$

Now,

$$[\nu:T_1] = p \land \langle f:\text{field}(\nu, f)\rangle \land \bigwedge_{\text{field}(\nu, f):T}$$

so, combined with the assumption that $[\Gamma] \Rightarrow p \Rightarrow p'$,

$$[\Gamma] \Rightarrow p \land \langle f:\text{field}(\nu, f)\rangle \land \bigwedge_{\text{field}(\nu, f):T'}$$

which is equivalent to $\{\nu;\langle f:T\rangle\mid p'\}$

Case $\Gamma \vdash \{\nu;?\langle l\rangle\mid p\} \leq \{\nu;\langle l\rangle\mid p'\}$

By assumption,

$$[\Gamma] \Rightarrow p \Rightarrow (p' \land \nu \neq \text{null})$$

By definition,

$$[\nu;\{\nu;?\langle l\rangle\mid p\}] = p \land (\nu = l \land \nu \neq \text{null}) \lor (\nu = \text{null})$$

and thus, combined with the assumption,

$$[\Gamma] \Rightarrow p \Rightarrow (p' \land \nu = l \land \nu \neq \text{null})$$

which implies

$$[\nu;\{\nu;\langle l\rangle\mid p'\}] = p' \land \nu = l \land \nu \neq \text{null}$$
Case $\Gamma \vdash \{\nu:\langle l \rangle \mid p\} \preceq \{\nu:\langle l \rangle \mid p'\}$

By assumption,

\[
\llbracket \Gamma \rrbracket \Rightarrow p \Rightarrow (p' \land \nu \neq \text{null} \land \nu = l)
\]

and thus

\[
\llbracket \Gamma \rrbracket \Rightarrow p \Rightarrow (p' \land ((\nu \neq \text{null} \land \nu = l) \lor \nu = \text{null}))
\]

which is equivalent to $\llbracket \nu:\langle l \rangle \mid p' \rrbracket$.

---

Case $\Gamma \vdash \{\nu:\text{null} \mid p\} \preceq \{\nu:\langle l \rangle \mid p'\}$

By assumption,

\[
\llbracket \Gamma \rrbracket \Rightarrow p \Rightarrow (p' \land \nu = \text{null})
\]

and thus

\[
\llbracket \Gamma \rrbracket \Rightarrow p \Rightarrow (p' \land ((\nu \neq \text{null} \land \nu = l) \lor \nu = \text{null}))
\]

which is equivalent to $\llbracket \nu:\langle l \rangle \mid p' \rrbracket$.

---

Case $\Gamma \vdash \text{emp} \preceq \text{emp}$

The heaps are equivalent and their domains are empty, so the conclusion is trivially true.

---

Case $\Gamma \vdash \Sigma \ast \ell \mapsto x:T \preceq \Sigma' \ast \ell \mapsto x:T'$

By the inductive hypothesis,

\[
\llbracket \Gamma' \rrbracket \Rightarrow \llbracket \Sigma \rrbracket \Rightarrow \llbracket \Sigma' \rrbracket
\]

and by assumption and the definition of $\llbracket \cdot \rrbracket$,  

\[
\llbracket \Gamma \rrbracket \Rightarrow \llbracket \ell \mapsto x:T \rrbracket \Rightarrow \llbracket \ell \mapsto x:T' \rrbracket
\]

hence,

\[
\llbracket \Gamma' \rrbracket \Rightarrow \llbracket \ell \mapsto x:T \ast \Sigma \rrbracket \Rightarrow \llbracket \ell \mapsto x:T' \ast \Sigma' \rrbracket
\]

and thus,

\[
\llbracket \Gamma' \rrbracket \Rightarrow \llbracket \ell \mapsto x:T \ast \Sigma \rrbracket \Rightarrow \llbracket \ell \mapsto x:T' \ast \Sigma' \rrbracket
\]

As a consequence of our interpretation of typing contexts, the following lemma immediately follows:

**Lemma 8.** Assume that $\Phi$, $\Gamma$, and $\Sigma$ are (respectively) well formed global, local, and heap contexts. We may then write the denotation of these typing contexts in the following way:

\[
\llbracket \Gamma, \Sigma \rrbracket = G \ast H
\]

where the following hold:

1. $G$ is pure.
2. \( G \) is a conjunction of the assertions

\[
G_x \doteq \llbracket x: T \rrbracket
\]

for each \( x: T \in \Gamma \). \( G \) may thus be written \( \cdots \land G_x \land \cdots \) for each such binding in \( \Gamma \).

3. \( H \) is a conjunction (via \( \ast \)) of the assertions

\[
H_\ell \doteq (\ell = \text{nil} \lor \llbracket \ell \mapsto x: T \rrbracket)
\]

for each \( \ell \mapsto x: T \in \Sigma \). \( G \) may thus be written \( \cdots \ast H_\ell \ast \cdots \) for each such binding in \( \Sigma \).

Proof. Follows immediately from the definition of \( \llbracket - \rrbracket \).

Lemma 9. 

If \( \Gamma, \Sigma \vdash e : T \)

Then \( \llbracket \Gamma, \Sigma \rrbracket = G \ast H \)

such that

1. \( G \) is pure.
2. \( G \Rightarrow \llbracket e: T \rrbracket \)

Proof. By induction on the expression typing judgment. In particular, we make use of the fact that the typing and subtyping rules for pure expressions only depend on \( \Gamma \). Subsumption requires application of lemma 7.

Lemma 10. Suppose that \( \Gamma, \Sigma \vdash x : \langle \ell \rangle \) and that \( \Gamma \vdash \Sigma = \Sigma_0 \ast \ell \mapsto y: T \). It then follows that \( \llbracket \Gamma, \Sigma \rrbracket = G \ast H \ast H_\ell \) where \( G \) is pure and:

(i) \( G \Rightarrow (x = \ell \land x \neq \text{null}) \)

(ii) \( H = H' \ast H_\ell \ast H'' \) such that

\[
G \ast H_\ell \Rightarrow \llbracket x \mapsto y: T \rrbracket
\]

and furthermore, if \( T \) is a record type,

\[
G \ast \llbracket x \mapsto y: T \rrbracket \ast H_\ell \Rightarrow x \Rightarrow y
\]

equivalently:

\[
G \ast \llbracket x \mapsto y: T \rrbracket \ast H_\ell \Rightarrow x \Rightarrow \langle f_1: \text{field}(y, f_1) \rangle
\]

Proof. Follows from lemma 9 and the definition of \( \llbracket - \rrbracket \).

Lemma 11. [Folding] If \( \Gamma \vdash x: T_1/\Sigma_1 \triangleright x: T_2/\Sigma_2 \) then

\[
\llbracket \Gamma \rrbracket \Rightarrow \llbracket \ell \mapsto x: T_1 \ast \Sigma_1 \rrbracket \Rightarrow \exists \text{FV}(\Sigma_2), \llbracket \ell \mapsto x: T_2 \ast \Sigma_2 \rrbracket \land \exists y. \text{Snapshot}(\ell, y)
\]

Proof. By induction on the judgement derivation.

Case F-BASE

By the subtyping hypothesis,

\[
\llbracket \Gamma \rrbracket \land \llbracket \ell \mapsto x: T_a \rrbracket \Rightarrow \llbracket \ell \mapsto x: T_b \rrbracket
\]
And thus, by letting every other $\ell \in \text{Dom}(\Sigma_2) = \text{null}$,
\[
\exists \text{FV}(\Sigma_2). \; \llbracket \ell \mapsto x:T_2 * \Sigma_2 \rrbracket.
\]
Since $T_1$ contains no pointers, and since we are assuming the satisfiability of $\llbracket x:T_1 \rrbracket$, we can construct $y$ from the base type of $T_1$ and its refinement.

Case F-ref
By the subtyping hypothesis, where $T_1 = \{ \nu: \langle \ell' \mid p \rangle \}$,
\[
\llbracket \Gamma \rrbracket \Rightarrow \llbracket x:T_1 \rrbracket \Rightarrow \llbracket x:T_2 \rrbracket
\]
And by the inductive hypothesis,
\[
\llbracket \Gamma \rrbracket \Rightarrow \llbracket \ell' \mapsto x:T * \Sigma'_1 \rrbracket \Rightarrow \exists \text{FV}(\Sigma'_2). \; \llbracket \ell' \mapsto x:T' * \Sigma'_2 \rrbracket \land \exists s. \; \text{Snapshot}(\ell', e)
\]
Combining the two gives us
\[
\exists \text{FV}(\Sigma_2). \; \llbracket \ell \mapsto y:T_2 * \Sigma_2 \rrbracket.
\]
We construct $y = (\ell', s)$ so that $y \in \langle \ell' \rangle^*$ to yield
\[
\text{Snapshot}(\ell, y)
\]

Case F-?ref
By hypothesis,
\[
((x \neq \text{null}) \land \llbracket \Gamma, \ell \mapsto x:T * \Sigma'_1 \rrbracket) \Rightarrow
\]
\[
(\exists \text{FV}(\Sigma'_2). \; \llbracket \ell \mapsto x:T' * \Sigma'_2 \rrbracket \land \exists y'. \; \text{Snapshot}(\ell, y'))
\]
\[
((x = \text{null}) \land \llbracket \Gamma, \ell \mapsto x:T * \Sigma'_1 \rrbracket) \Rightarrow
\]
\[
(\exists \text{FV}(\Sigma'_2). \; \llbracket \ell \mapsto x:T' * \Sigma'_2 \rrbracket \land \exists y'. \; \text{Snapshot}(\ell, y'))
\]
In the first case, $(x \neq \text{null})$ we let
\[
 y = (\ell, y').
\]
In the second case,
\[
y = \text{null}.
\]
We combine both hypotheses and $y$ constructions to conclude:
\[
\exists \text{FV}(\Sigma_2). \; \llbracket \ell \mapsto x:T_2 * \Sigma_2 \rrbracket.
\]
and, by definition, \text{Snapshot}(\ell, y),

Case F-heap
By hypothesis,
\[
\llbracket \Gamma, \ell \mapsto x:T * \Sigma_1 \rrbracket \Rightarrow
\]
\[
(\exists \text{FV}(\Sigma_2). \; \llbracket \ell \mapsto x:T * \Sigma_2 \rrbracket \land \exists x_i. \; \text{Snapshot}(\ell, x_i))
\]
for each field $(x, f_i)$ in $T_1$. If we substitute field $(x, f_i)$ for $x$ in each conclusion, we conclude
\[
\llbracket \ell \mapsto x:T_2 * \Sigma_2 \rrbracket
\]
and we construct $y = \langle f_i : x_i \rangle$. 
Lemma 12. Given the definition
\[ C[\alpha] \triangleq \exists! \Sigma. x:T, \]
and the set of corresponding pure values, \( C^*(A) \):
\[ \text{Snapshot}(\ell, x_f) \land \llbracket \ell \mapsto x:T \ast \Sigma \rrbracket \Rightarrow x_f \in C^*(A) \]

Proof. By contrapositive. Assume \( x_f \notin C(A)^* \). Then
\[ x_f \notin \text{Pure}_C(A, C^*, T) \Sigma. \]
The proof is by induction on the derivation of \( x_f \notin \text{Pure}_C(A, C^*, T) \Sigma \). The base case starts with the root of \( \text{Snapshot}(\ell, x_f) \)
\[ \text{Snapshot}(\ell, x_f) = \ell \mapsto e \cdots \]
If \( e \notin \text{Pure}_C(A, C^*, T) \Sigma \) then either \( e \) is not a record value, trivially showing \( \neg \llbracket \ell \mapsto x:T \ast \Sigma \rrbracket \), or \( e \) contains some field \( f \) with value \( e_f \) such that.

(i) \( f : T_f \in T \);
(ii) and \( e_f \notin \text{Pure}_C(A, C^*, T_f) \Sigma \)
Assuming \( e_f \notin (\ell', e') \), if \( e_f \notin \text{Pure}_C(A, C^*, T_f) \Sigma \), then the assertion in \( \text{Snapshot}(\ell, x_f) \), \( \ell \mapsto (\cdots f: e_j \cdots) \), must not satisfy the corresponding \( \llbracket \ell \mapsto x:T \ast \Sigma \rrbracket \) (by the definition of \( \text{Pure}_C(A, C^*, T) \Sigma \)). If \( e_f = (\ell', e') \) for some \( \ell', e' \), then \( e' \notin \text{Pure}_C(A, C^*, T_f) \Sigma \) However, by the same reasoning as above, \( \text{Snapshot}(\ell', e') \) must not be equisatisfiable with \( \llbracket \ell \mapsto x:T \ast \Sigma \rrbracket \)

Theorem 2. [Statement Typing] If \( \Phi, \Gamma, \Sigma \vdash s :: \Gamma'/\Sigma' \) then \( \llbracket \Phi \rrbracket \vdash \{ \llbracket \Gamma, \Sigma \rrbracket \} s \{ \llbracket \Gamma', \Sigma' \rrbracket \} \)

Proof. By induction on the typing derivation of \( s \).

Case \( x = e \):
By assumption, \( x \) does not appear in \( \Gamma \). Since \( e \) is well-typed, \( x \notin \text{FV}(e) \). By the frame rule, since \( x \notin \Gamma \) and \( x \notin \Sigma \), we derive the following rule:
\[ \llbracket \Phi \rrbracket \vdash \{ \llbracket \Gamma, \Sigma \rrbracket \} x = e \{ \llbracket \Gamma, \Sigma \rrbracket \ast x = e \} \]
We then perform the deduction by the definition of \( \llbracket : : \rrbracket \):
\[ \{ \llbracket \Gamma, \Sigma \rrbracket \land (x = e) \} \]
\[ \{ \llbracket \Gamma, \Sigma \rrbracket \land \{ x : \nu : \tau \mid \nu = e \} \} \]
\[ \{ x : \nu : \tau \mid \nu = e \} ; \Gamma, \Sigma \} \]

Case \( s; s' \)
By hypothesis,
\[ \Phi \vdash \{ \llbracket \Gamma, \Sigma \rrbracket \} s_1 \{ \llbracket \Gamma', \Sigma' \rrbracket \} \]
By straightforward application of the sequence rule, we immediately conclude
\[ \Phi \vdash \{ [ \Gamma, \Sigma ] \} s_1; s_2 \{ [ \Gamma''', \Sigma''' ] \} \]

**Case if e then s₁ else s₂:**

By hypothesis:
\[ \Phi \vdash \{ e \land [ \Gamma, \Sigma ] \} s_1 \{ [ \Gamma_1; \Gamma, \Sigma_1 \ast \Sigma'_1 ] \} \]

and
\[ \Phi \vdash \{ \neg e \land [ \Gamma, \Sigma ] \} s_2 \{ [ \Gamma_2; \Gamma, \Sigma_2 \ast \Sigma'_2 ] \} \]

For each \( x_i \), by lemma 12
\[ [ \Gamma_1; \Gamma ] \Rightarrow [ x_i : T' ] \quad \text{and} \quad [ \Gamma_2; \Gamma ] \Rightarrow [ x_i : T' ] \]

and by lemma 13
\[ [ \Gamma_1; \Gamma ] \Rightarrow [ \Sigma_1 ] \Rightarrow [ \Sigma' ] \quad \text{and} \quad [ \Gamma_2; \Gamma ] \Rightarrow [ \Sigma_2 ] \Rightarrow [ \Sigma' ] \]

Combining these facts allows us to conclude,
\[ [ \Gamma_1; \Gamma, \Sigma_1 ] \Rightarrow [ x_i : T_i : \Gamma, \Sigma' ] \quad \text{and} \quad [ \Gamma_2; \Gamma, \Sigma_2 ] \Rightarrow [ x_i : T_i : \Gamma, \Sigma' ] \]

and thus, by consequence,
\[ \Phi \vdash \{ e \land [ \Gamma, \Sigma ] \} s_1 \{ [ x_i : T_i : \Gamma, \Sigma' ] \} \]

and
\[ \Phi \vdash \{ \neg e \land [ \Gamma, \Sigma ] \} s_2 \{ [ x_i : T_i : \Gamma, \Sigma' ] \} \]

hence, by the conditional rule,
\[ \Phi \vdash \{ [ \Gamma, \Sigma ] \} \text{if e then s₁ else s₂} \{ [ x_i : T_i : \Gamma, \Sigma' ] \} \]

**Case concr (x,z):**

Letting \[ [ \Gamma, \Sigma ] = G \ast H \] by lemma 14, by assumption
\[ \{ G \ast H \} \]

By hypothesis and lemma 15
\[ \{ G \ast H \ast (x = \ell \land \ell \neq \text{null}) \ast \ell \rightarrow [ y : T_y ] \} \]

By consequence, assuming without loss of generality that \( T_y \) is a record type, substitution on the rule for concr, and applying the frame rule,
\[ \{ G \ast H \ast (x = \ell \land \ell \neq \text{null}) \ast x \rightarrow [ y : T_y ] \} \]

concr\( (x, z) \)
\[ \{ G \ast H \ast (x = \ell \land \ell \neq \text{null}) \ast x \rightarrow [ y : T_y ] \land z = y \} \]
since $z$ is a fresh variable. By definition of $[\cdot]$ and $T_{y'}$, since $z$ does not appear free in $G, H$ or $T_y$, 
\[
\{ \; y:T_y; \Gamma, \ell \rightarrow z:T_z * \Sigma' \; \} \]

---

**Case $\text{pad}(\ell, x)$**

Beginning with the proof rule for $\text{pad}(\ell, x)$,
\[
\vdash \{ \text{emp} \} \text{pad}(\ell, x) \{ \text{emp} \land \ell = \text{null} \land x = \text{null} \}
\]

By hypothesis, neither $\ell$ nor $x$ appear in $\Gamma$ or $\Sigma$, and thus do not appear in $[\Gamma, \Sigma]$, so we can apply the frame rule:
\[
\vdash \{ [\Gamma, \Sigma] \} \text{pad}(\ell, x) \{ [\Gamma, \Sigma] * (\ell = \text{null} \land x = \text{null}) \}
\]

By consequence,
\[
\vdash \{ [\Gamma, \Sigma] \} \text{pad}(\ell, x) \{ [\Gamma, \Sigma] * [\ell \rightarrow x:T] \}
\]

And thus,
\[
\vdash \{ [\Gamma, \Sigma] \} \text{pad}(\ell, x) \{ [\Gamma, \Sigma * \ell \rightarrow x:T] \}
\]

---

**Case $\text{unfold}(\ell, \bar{x})$**

Letting $[\Gamma, \Sigma] = G * H * [\ell \rightarrow x:C[\bar{T}]]$ by lemma 8 and by assumption,
\[
\{ \quad G * H * ((\ell \neq \text{null}) \Rightarrow c(T, \ell, x)) \land ((\ell = \text{null}) \Rightarrow (x = \text{null})) \quad \}
\]

First, assume $\ell = \text{null}$. Then,
\[
\{ \quad G * H * (\ell = \text{null} \land x = \text{null}) \quad \}
\]

And by lemma 6,
\[
\{ \quad G * H * (x = \text{null}) \land \bigwedge_m \{ m(\text{null}) \}_\text{exp} = e_m \quad \}
\]

Quantifying over $x$ allows us to complete the proof via consequence and the frame rule:
\[
\{ \quad [\Gamma, \Sigma] \quad \}
\]
\[
\{ \quad \exists \text{FV}(\bar{\ldots}). \quad G * H * (\ell = \text{null} \land x = \text{null}) \land \bigwedge_m \{ m(\text{null}) \}_\text{exp} = e_m \quad \}
\]

\text{unfold}(\ell, y \cdot \bar{y})
\[
\{ \quad G * H * (\ell = \text{null} \land x = \text{null}) \land \bigwedge_m \{ m(\text{null}) \}_\text{exp} = e_m \quad \}
\]

\[
\{ \quad [\bigwedge_m \{ m(y) \}_\text{exp} = e_m; \Gamma, \Sigma'] \quad \}
\]

Next, we assume $\ell \neq \text{null}$. By definition,
\[
\{ \quad G * H * \exists x_c. \quad \exists \text{FV}(\Sigma_c). \quad [\ell \rightarrow x_c:T_c * \Sigma] \land \text{Snapshot}(\ell, x) \quad \}
\]
By hypothesis, \( x_c \) and the free variables of \( \llbracket \Sigma_c \rrbracket \) have been \( \alpha \)-renamed and do not appear in \( \llbracket \Gamma \rrbracket \) or \( \llbracket \Sigma \rrbracket \). By the unfold axiom and the frame rule,
\[
\begin{align*}
& \{ \, G \cdot H \cdot \exists x_c . \exists \text{FV}(\Sigma_c) . \llbracket \ell \mapsto x_c ; [T/\alpha]T_c \cdot [T/\alpha] \Sigma_c \rrbracket \land \text{Snapshot}(\ell, x) \, \} \\
& \text{un\text{d}f\text{o}l}(\ell, x_c \cdot \text{Dom}(\Sigma_c) \cdot \text{Binders}(\Sigma_c)) \\
& \{ \, G \cdot H \cdot \llbracket \ell \mapsto x_c ; [T/\alpha]T_c \cdot [T/\alpha] \Sigma_c \rrbracket \land \text{Snapshot}(\ell, x) \, \}
\end{align*}
\]
where \( x \in C^* \). Therefore, by lemma 8, each well-formed measure on \( C[\alpha] \), \( m \), is defined on \( x \) and we strengthen:
\[
\{ \, G \cdot H \cdot \llbracket \ell \mapsto x_c ; [T/\alpha]T_c \cdot [T/\alpha] \Sigma_c \rrbracket \land \text{Snapshot}(\ell, x) \land (m(x) ||_{\exp} e_m) \, \}
\]
Which is, by definition
\[
\{ \, \llbracket (m(x) ||_{\exp}) = e_m : \Gamma \rrbracket \land \llbracket \Sigma' \rrbracket \, \}
\]

**Case \( x = z . \text{alloc} \{ f : e \} \)**

By assumption, \( x \) does not appear bound in \( \Gamma \) and \( \ell \) does not appear in \( \llbracket \Gamma, \Sigma \rrbracket \). Applying the frame rule to the rule for \text{alloc} gives us the precondition
\[
\{ \llbracket \Gamma, \Sigma \rrbracket \}
\]
and the postcondition
\[
\{ \llbracket \Gamma, \Sigma \rrbracket \cdot * x = \ell \land \ell \neq \text{null} \land x \mapsto z \land z = \langle f : e \rangle \}
\]
Rearranging and by consequence:
\[
\{ \llbracket \Gamma, \Sigma \rrbracket \cdot * x = \ell \land \ell \neq \text{null} \land (\ell \neq \text{null} \Rightarrow \ell \mapsto z \land z = \langle f : e \rangle) \}
\]
By the typing hypotheses and lemma 9,
\[
\llbracket \Gamma \rrbracket \Rightarrow [e : T_f]
\]
\[
\{ \llbracket \Gamma, \Sigma \rrbracket \cdot * x = \ell \land \ell \neq \text{null} \land (\ell \neq \text{null} \Rightarrow \ell \mapsto z \land z = \langle f : e \rangle \land \llbracket e : T_f \rrbracket) \}
\]
so eq. 12 and the definition of \( \llbracket \cdot \rrbracket \) imply,
\[
\{ \llbracket \Gamma, \Sigma * \ell \mapsto z : T \rrbracket \}
\]

**Case \( y = x . f_i \)**

By hypothesis, lemma 8 and lemma 10, we let
\[
\llbracket \Gamma, \Sigma \rrbracket = G \cdot H \cdot \llbracket \ell \mapsto z ; \langle \ldots f_i ; \text{field}(z, f_i) \ldots \rangle \rrbracket
\]
and by consequence,
\[
\{ \, G \cdot H \cdot \llbracket z : \langle f : T_f \rangle \rrbracket \cdot * x \mapsto \langle \ldots f_i ; \text{field}(z, f_i) \ldots \rangle \, \}
\]
By the frame rule:
\[
\{ \, G \cdot H \cdot \llbracket z : \langle f : T_f \rangle \rrbracket \cdot * x \mapsto \langle \ldots f_i ; \text{field}(z, f_i) \ldots \rangle \, \}
\]
\( y = x . f_i \)
\[
\{ \, G \cdot H \cdot \llbracket z : \langle f : T_f \rangle \rrbracket \cdot * x \mapsto \langle \ldots f_i ; \text{field}(z, f_i) \ldots \rangle \land (y = \text{field}(z, f_i)) \, \}
\]
Unfolding the definition of $\llbracket \cdot \rrbracket$ and by consequence (using the equality $y = \text{field}(z, f_i)$),

$$
\{ G \ast H \ast \llbracket z: \langle f:T_i \rangle \rrbracket \ast x \mapsto \langle \ldots f_i: \text{field}(z, f_i) \ldots \rangle \land \llbracket T_i: y \rrbracket \}
$$

And hence, by definition and consequence (to weaken the mapping of $x$ and applying the equality $x = \ell$),

$$
\{ \llbracket y: T_i; \Gamma, \Sigma \rrbracket \}
$$

\underline{Case $x.f_i = z$ e:}

Starting from $\{ \llbracket \Gamma, \Sigma \rrbracket \}$, by lemmas 8 and 10, we deduce by consequence,

$$
\{ (G \ast (x = \ell \land \ell \neq \text{null}) \ast H \ast \llbracket y: \langle f_j: \nu: \tau_j \mid \rho_j \rangle \rrbracket \ast x \mapsto \langle f_j: \text{field}(y, f_j) \rangle) \}
$$

Thus, by the frame rule,

$$
\begin{align*}
\{ & (G \ast (x = \ell \land \ell \neq \text{null}) \ast H \ast \llbracket y: \langle f_j: \nu: \tau_j \mid \rho_j \rangle \rrbracket \ast x \mapsto \langle f_j: \text{field}(y, f_j) \rangle) \\
& x. f_i = z \ e \\
\{ & (G \ast (x = \ell \land \ell \neq \text{null}) \ast H \ast \llbracket y: \langle f_j: \nu: \tau_j \mid \rho_j \rangle \rrbracket \ast x \mapsto z \land z = \langle \ldots f_i: e \ldots \rangle) \}
\end{align*}
$$

By consequence and lemma 4,

$$
\begin{align*}
\{ & (G \ast \llbracket e: \nu: \tau \mid \rho \rrbracket) \ast (x = \ell \land \ell \neq \text{null}) \ast H \\
& \ast \llbracket y: \langle f_j: \nu: \tau_j \mid \rho_j \rangle \rrbracket \ast x \mapsto z \land z = \langle \ldots f_i: e \ldots \rangle) \}
\end{align*}
$$

Unfolding the definition of $\llbracket \cdot \rrbracket$, this expands to

$$
\begin{align*}
\{ & (G \ast \llbracket e: \nu: \tau \mid \rho \rrbracket) \ast (x = \ell \land \ell \neq \text{null}) \ast H \\
& \ast y = \langle f_j: \text{field}(y, f_j) \rangle \land \bigwedge_{f \in \mathcal{F}_j} \llbracket \text{field}(y, f_j); T_j \rrbracket \\
& \ast \llbracket y: \langle f_j: \nu: \tau_j \mid \rho_j \rangle \rrbracket \ast x \mapsto z \land z = \langle \ldots f_i: e \ldots \rangle) \}
\end{align*}
$$

By eq. (12),

$$
\begin{align*}
\{ & (G \ast \llbracket e: \nu: \tau \mid \rho \rrbracket) \ast (x = \ell \land \ell \neq \text{null}) \ast H \\
& \ast y = \langle f_j: \text{field}(y, f_j) \rangle \land \bigwedge_{f \in \mathcal{F}_j} \llbracket \text{field}(y, f_j); T_j \rrbracket \\
& \ast \llbracket y: \langle f_j: \nu: \tau_j \mid \rho_j \rangle \rrbracket \ast x \mapsto z \land z = \langle f_0: \text{field}(y, f_0) \ldots f_i: \text{field}(z, f_i) \ldots \rangle \\
& \ast (\text{field}(z, f_i) = e) \} \\
\end{align*}
$$

By definition of $\llbracket \cdot \rrbracket$,

$$
\begin{align*}
\{ & (G \ast \llbracket e: \nu: \tau \mid \rho \rrbracket) \ast (x = \ell \land \ell \neq \text{null}) \ast H \\
& \ast y = \langle f_j: \text{field}(y, f_j) \rangle \land \bigwedge_{f \in \mathcal{F}_j} \llbracket \text{field}(y, f_j); T_j \rrbracket \\
& \ast \llbracket y: \langle f_j: \nu: \tau_j \mid \rho_j \rangle \rrbracket \ast \llbracket \ell \mapsto z; T_r \rrbracket \} \\
\end{align*}
$$
and thus, by definition,
\[
\{ \llbracket \Gamma, \Sigma \rrbracket \}
\]
\[
x.f_i = e
\]
\[
\{ \llbracket \Gamma, \ell \mapsto z : T_x \ast \Sigma \rrbracket \}
\]

Case fold(ℓ, z)

Given \( C[\alpha] \models \exists ! \Sigma. x : T \),
\[
c(T_\alpha, \ell, y) \models (\exists x, \text{FV}(\Sigma). [\ell \mapsto x : T_\alpha / \alpha] \ast [T_\alpha / \alpha] \ast \Sigma) \land \text{Snapshot}(\ell, y)
\]
By lemma \([\nabla]\) applied to the type judgement folding assumption we deduce
\[
\exists \Sigma_\ell. [\ell \mapsto x : \theta T_\ell \ast \theta \Sigma_\ell] \land \exists y. \text{Snapshot}(\ell, y) \quad \text{where } \theta = [T/\alpha]
\]
and, by the definition of \( c(T, \ell, x) \):
\[
\exists y. c(T, \ell, y)
\]
Applied to the rule for fold(ℓ, y), since by assumption y is a fresh variable,
\[
\vdash \{ \exists y. c(T, \ell, y) \} \text{fold}(\ell, y) \{ c(T, \ell, y) \}
\]
By lemma \([\nabla]\), \( x \in C[T]^* \), so by lemma \([\nabla]\), each well-formed measure on \( C[\alpha] \), \( m \), is defined on \( y \). By the form of \( T_\ell \) and the definition of \( \text{Snapshot} \), for any field of \( x \) \( \text{field}(y, f) = \text{field}(x, f) \) and we thus strengthen the postcondition:
\[
\{ c(T, \ell, y) \land \bigwedge_m m(y) = e_m \}.
\]
Applying the frame rule and consequence with the definition of \( T_y \) and lemma \([\nabla]\) applied to \( \Sigma' \),
\[
\Phi \vdash \{ \llbracket \Gamma, \ell \mapsto x : T_x \ast \Sigma_x \ast \Sigma \rrbracket \} \text{fold}(\ell, y) \{ \llbracket \Gamma, \ell \mapsto y : T_y \ast \Sigma' \rrbracket \}
\]

Case \( x_r = f(T) \)

Assuming \( f : S \) and
\[
S = \forall \ell. \forall x. (x) / \Sigma_i \ast \Sigma' \Rightarrow \exists \ell_0, x_0 : T_0 / \Sigma_0
\]
let \( P = \text{Pre}(S) \) and \( Q = \text{Post}(S) \). We apply the substitution \( \theta = [e_j/x_j] \) to obtain:
\[
\theta P = [e_j : \theta T_j] \land [\theta \Sigma_i]
\]
\[
\theta Q = [x_0 : \theta T_0] \land [\theta \Sigma_0]
\]
By lemmas \([\nabla]\) and \([\nabla]\),
\[
[\Gamma] \Rightarrow [e_j : \theta T_j]
\]
\[
[\Gamma] \Rightarrow [\Sigma_m] \Rightarrow [\theta \Sigma_i]
\]
and thus
\[
\llbracket \Gamma \rrbracket \Rightarrow \llbracket \Sigma_m \ast \Sigma_u \rrbracket \Rightarrow \llbracket \Gamma \rrbracket \land \theta P \ast \Sigma_u
\]
Which gives us, by consequence, and framing of \( \Gamma \) and \( \Sigma_u \), and unfolding the definition of \( Q \),
\[
\Phi \vdash \{ \llbracket \Gamma, \Sigma_u \ast \Sigma_m \rrbracket \} x = f(T) \{ \llbracket \Gamma, \Sigma_u \ast \theta \Sigma_o \rrbracket \}
\]
Theorem 3. [Procedure Typing]
If $\Phi \vdash f :: S$ then $\llbracket \Phi \rrbracket$ $\llbracket \text{Body}(f) \rrbracket$ $\llbracket \text{Post}(S) \rrbracket$.

Proof. Let

\[
P = \text{Pre}(S) = [\overrightarrow{x:T}] \land [\Sigma_i]
\]
\[
Q = \text{Post}(S) = [x_o:T_o] \land [\Sigma_o]
\]

By inversion on the judgement, we must have typed a statement $s' = \text{return } e$ for some expression, or a sequence of statements ending in a return. By the statement typing theorem,

$\Phi \vdash \{\text{Pre}(S)\} s \{[\Gamma_s, \Sigma_s] \}$

Applying lemma 8 and lemma 7 to the hypotheses of the return typing rule, we determine

\[
[\Gamma_s, \Sigma_s] \Rightarrow [e/x_o][x_o:T_o]
\]
\[
[\Gamma_s, \Sigma_s] \Rightarrow [e/x_o][\Sigma_o]
\]

and thus

$\Phi \vdash \{[\Gamma_s, \Sigma_s]\} \text{return } e \{\text{Post}(S)\}$.

By the sequence rule,

$\Phi \vdash \{\text{Pre}(S)\} s; \text{return } e \{\text{Post}(S)\}$.

\[\square\]

Corollary 4. [Soundness] If $\Phi, \emptyset, \text{emp} \vdash s :: \Gamma/\Sigma$, then $\llbracket \Phi \rrbracket \vdash \{\text{true}\} s \{\text{true}\}$

E Heap Annotation Inference

Up until this point, our presentation has assumed that the annotations \texttt{concr}, \texttt{unfold}, and \texttt{fold} have already been inserted into the source file. To alleviate this burden, ART automatically inserts these annotations at critical locations in the input program. To be sure, the soundness of the type system does not depend on any particular algorithm for inferring these locations. In fact, while the algorithm we will present was sufficient for the benchmarks we used to test ART, it would be entirely feasible to swap our algorithm for another. In this section, we will describe the simple source elaboration that preceeds refinement type checking and inference.

Annotation inference. To formalize our method of inferring annotation locations, we define a function $\mathcal{F}$ with type:

$\mathcal{F} : \text{Statement} \times S \rightarrow \text{Statement} \times \mathcal{P}(\mathcal{U})$

$\mathcal{S} \equiv \text{Type Bindings} \times \text{Heap} \times \text{Function} \times \mathcal{U}$

$\mathcal{U} \equiv \text{Location} \times \text{Type Constructor}$
Thus, $F(s, (\Gamma, \Sigma, f, U))$ transforms the statement $s$ yielding a new (possibly compound, i.e. comprising several sequenced statements) statement and a new set of unfolded locations. The input to $F$ is $s$, the statement to be transformed; $\Gamma$, containing physical type information for local variables; $\Sigma$, containing physical heap type information; $f$, the physical specification of the function containing $s$; and $U$, a set of $(\ell, C)$ pairs that denote unfolded locations and the type constructor that was unfolded.

Because this step precedes refinement type checking, the only type information that is available is base type information – i.e. all types are of the form $\{\nu: \tau | \text{true}\}$. Definitions for $F$ on the statements that may possibly generate annotations are given in Fig. 21 with some helper functions given in Fig. 22.

**Function Declarations.** $F$ transforms programs function by function. Thus it simply calls $F$ recursively on the function body, querying the type system to determine $\Gamma$ and $\Sigma$ with $\text{Env}(f)$ and $\text{HeapIn}(f)$.

**Field Access and Mutation.** On a field read or write, ART computes the set of locations that must be unfolded, using the $\text{UnfoldList}$ helper. $\text{UnfoldList}$ computes this list by (1) consulting $\Gamma$ and heap to determine if $x$ points to a location that is folded up; and (2) subtracting from this list all locations that are already unfolded. ART inserts $\text{unfold}$ calls for these locations before the field access.

ART also optimistically inserts a $\text{concr}$ before a field access. The $\text{concr}$ will type check exactly when the field access type checks.

**Function Return.** At function returns, ART must take care to ensure that $\Sigma$ agrees with the current function’s specified heap with respect to which locations folded. If a location in the function’s schema is unfolded, it must first be folded up. At a return statement in a function, $f$, ART computes these locations by comparing the current heap, $\Sigma$ with $f$’s specified heap. Given a “current” heap $\Sigma$ and a “target” heap $\Sigma'$, $\text{FoldList}$ computes the set of locations in $\Sigma$ that must be folded up. These locations may depend on each other, so $\text{FoldList}$ orders these $\text{fold}$ calls using the $\text{FoldOrder}$ function.

**Function Calls.** With respect to folds, function calls behave exactly like function returns.

**Example.** Revisiting `abslist`, consider the following:

```javascript
var l1 = { data:0, next: null };
var l2 = { data:1, next: l1 }; //: fold(&l1);
absList(l2);
```

$\text{FoldList}$ determines that the locations (and their associated types) that must be folded up before calling $\text{absList}$ are ($&l2, \text{List[number]}$) and ($&l1, \text{List[number]}$). However, $&l2$ depends on $&l1$. $\text{FoldOrder}$ realizes this dependency with the correct ordering $\text{fold}(&l1); \text{fold}(&l1);$. The code is annotated with $\text{fold}$ and $\text{concr}$ calls are as follows:

```javascript
var l1 = { data:0, next: null }; 
var l2 = { data:1, next: l1 }; 
//: fold(&l1);
```
If Statements. To annotate if statements, ART calls $F$ on the statements in the “then” and “else” branches. To determine which statements must be folded, $F$ calculates three sets. $L_1$ and $L_2$ are the sets of unfolded locations that were previously folded before the if statement. Additionally, $F$ queries the type checker to get heaps $\Sigma_1$ and $\Sigma_2$ that are returned from type checking $s_1$ and $s_2$, respectively. These heaps are used to determine $L_{alias}$, by calling Alias. This procedure determines if a location $\ell$ would cause any references to become aliased (and thus eventually fail to type check). Using the same ordering as in function returns and function calls, ART calls FoldOrder to determine the sequence of folds that need to occur in both the “then” and “else” branches.

Example. In the following, suppose $x$ is a pointer to a list. The code

```plaintext
var d = x.data;
if (d > 0) {
    x.next = { data: 1, next: null };
} else {
    x.next = { data: -1, next: null };
}
```

would thus be annotated

```plaintext
//: unfold(&x)
var d = x.data;
if (d > 0) {
    x.next = { data: 1, next: null };
    //: fold(&x)
} else {
    x.next = { data: -1, next: null };
    //: fold(&x)
}
```

because the next fields of $x.next$ at the end of either branch point to different locations. Folding $&x$ ensures a consistent view of the heap after the control flow join.

Padding Whenever a heap subtyping occurs, it is possible that the sub-heap has fewer locations than the super-heap. However, the heap subtyping judgements requires the sub-heap and super-heap to have equivalent domains. We insert pad statements at these locations when the physical type checker determines that the sub-heap’s domain is too small.
\[ \mathcal{F}(\text{function } f(\overline{x}) \{ s \}, \mathcal{S}) = (\text{function } f(\overline{x}) \{ s' \}, \emptyset) \]

where \( (s', \emptyset) = \mathcal{F}(s, (\text{Env}(f), \text{HeapIn}(f), \emptyset)) \)

\[ \mathcal{F}(y = x, f, (\Gamma, \Sigma, \mathcal{U})) = (\text{concr}(x); u; y = x, f, \mathcal{U}') \]

where \( (u, \mathcal{U}') = \text{UnfoldList}(x, \Gamma, \Sigma, \mathcal{U}) \)

\[ \mathcal{F}(x, f = e, (\Gamma, \Sigma, \mathcal{U})) = (\text{concr}(x); u; x = e; \text{concr}(x), \mathcal{U}') \]

where \( (u, \mathcal{U}') = \text{UnfoldList}(x, \Gamma, \Sigma, \mathcal{U}) \)

\[ \mathcal{F}(x = g(e), (\Gamma, \Sigma, \mathcal{U})) = (w; p; x = g(e), \mathcal{U}') \]

where \( p = \text{PadLocs}(\Sigma, \text{HeapOut}(g)) \)

\( (w, \mathcal{U}') = \text{FoldList}(\Sigma, \text{HeapOut}(g), \mathcal{U}) \)

\[ \mathcal{F}(\text{return } e, (\Gamma, \Sigma, \mathcal{U})) = (w; p; \text{return } e, \emptyset) \]

where \( p = \text{PadLocs}(\Sigma, \text{HeapOut}(f)) \)

\( (w, \mathcal{U}') = \text{FoldList}(\Sigma, \text{HeapOut}(f), \mathcal{U}) \)

\[ \mathcal{F}(\text{if } e \text{ then } s_1 \text{ else } s_2, (\Gamma, \Sigma, f, \mathcal{U}) \text{ as } \mathcal{S}) = (\text{if } e \text{ then } s'_1; w_1; p_1 \text{ else } s'_2; w_2; p_2, \mathcal{U}\backslash \text{Lalias}) \]

where \( p_1 = \text{PadLocs}(\Sigma_1, \Sigma_2) \)

\( p_2 = \text{PadLocs}(\Sigma_2, \Sigma_1) \)

\( (s'_1, \mathcal{U}_1) = \mathcal{F}(s_1, \mathcal{S}) \)

\( (s'_2, \mathcal{U}_2) = \mathcal{F}(s_2, \mathcal{S}) \)

\( (\Sigma_1, \Sigma_2) = (\text{HeapAfter}(s_1), \text{HeapAfter}(s_2)) \)

\( \text{Lalias} = \{ (\ell, C) \mid \text{Alias}(\ell, \Sigma_1, \Sigma_2) \land (\ell, C) \in \mathcal{U} \} \)

\( (L_1, L_2) = (\mathcal{U}\backslash \mathcal{U}, \mathcal{U}\backslash \mathcal{U}) \)

\( w_1 = \text{FoldOrder}([\text{fold}(\ell) \mid (\ell, C) \in L_1 \cup \text{Lalias}], \Sigma) \)

\( w_2 = \text{FoldOrder}([\text{fold}(\ell) \mid (\ell, C) \in L_2 \cup \text{Lalias}], \Sigma) \)

---

Fig. 21: Statement annotation insertion
\text{UnfoldList}(x, \Gamma, \Sigma, U) = (U, L \cup U)
\text{where } U = \{\text{unfold}(\ell) \mid (\ell, C) \in L \setminus U\}
\text{and } L = \{(\ell, C) \mid \ell \in \text{TypeLocs}(\Gamma, x) \land (\ell \mapsto y:C[\varphi]) \in \Sigma\}

\text{FoldList}(\Sigma, \Sigma', U) = (W, L \cap U)
\text{where } W = \text{FoldOrder}(\{\text{fold}(\ell) \mid (\ell, C) \in L' \setminus L, \Sigma\})
\text{and } (L, L') = (\text{WoundLocs}(\Sigma), \text{WoundLocs}(\Sigma'))

\text{Alias}(\ell, \Sigma_1, \Sigma_2) = \ell \mapsto x_1{:}f{:}\ldots{:}x_2{:}\ldots \in \Sigma_1 \land \ell \mapsto x_2{:}f{:}\ldots \in \Sigma_2
\land |\text{locs}(\tau_1) \cup \text{locs}(\tau_2)| > 1

\text{TypeLocs}(\Gamma, x) = \{\text{locs}(\tau) \mid \Gamma \vdash x : \tau\}

\text{WoundLocs}(\Sigma) = \{(\ell, C) \mid \ell \mapsto x:C[-] \in \Sigma\}
\text{PadLocs}(\Sigma, \Sigma') = \{\text{pad}(\ell) \mod \ell \in \text{Dom}(\Sigma') \setminus \text{Dom}(\Sigma)\}

\text{Fig. 22: Annotation insertion helper functions}
0.1 Assertion language

Values in Imp are records, integers (which are also used as addresses), the constant null, or products of integers and records. We assume an intensional interpretation of assertions (and thus $p \land \textit{true} \iff p$).

\[
\begin{align*}
\text{Value} & = \mathbb{Z} \cup \text{null} \\
\text{null} & = \text{Records} \\
\mathbb{Z} & = \text{Integers (and addresses)} \\
\text{S} & = \mathbb{Z} \times \text{null}
\end{align*}
\]

Expressions \( E \) ::=
  \begin{align*}
  & \ldots \\
  & [\ell; E] \quad \text{record constant} \\
  & f(E) \quad \text{UIF application}
  \end{align*}

Assertions \( P \) ::=
  \begin{align*}
  & \ldots \\
  & \text{emp} \quad \text{empty heap} \\
  & E \Rightarrow E \quad \text{singleton heap} \\
  & P \land P \quad \text{separating conjunction}
  \end{align*}

With the following axiom:

\[
(y = \langle \ldots f:e \ldots \rangle) \Rightarrow (P \iff [\text{field}(y, f)/e]P)
\]

0.2 Imp Proof Rules

In addition to the standard frame rule and consequence rule, we assume the following axioms:

\[
F \vdash \{P\} \ s \ {\{Q\}}
\]

Allocation

\[
\text{pad}(\ell, x) \ {\{\text{emp} \land \ell \neq \text{null} \land x = z = \langle f; e \rangle\}}
\]

Access

\[
\text{a, v distinct vars} \quad \text{a, v distinct vars} \quad \text{a, v distinct vars}
\]

\[
\text{z} \neq \text{FV}(v)
\]

\[
\text{a, v distinct vars} \quad \text{a, v distinct vars} \quad \text{a, v distinct vars}
\]

\[
\text{z} \neq \text{FV}(v)
\]

\[
\text{a, v distinct vars} \quad \text{a, v distinct vars} \quad \text{a, v distinct vars}
\]

\[
\text{z} \neq \text{FV}(v)
\]

Sequence

\[
F \vdash \{P\} \ s \ {\{R\}} \quad F \vdash \{R\} \ s' \ {\{Q\}}
\]

\[
F \vdash \{P\} \ x = g(x_1 \ldots x_m) \ {\{Q\}} \quad F \vdash \{P\} \ s; \text{return} \ x_o \ {\{Q\}}
\]

Return

\[
\vdash \{t/x_o\} P \text{return} \ c \ {\{P\}}
\]

1. Soundness of ART

1.1 Definitions

Definition (Base type translation).

\[
\begin{align*}
\text{Int} & : \text{Int} \\
\text{null} & : \text{null} \\
\text{Int} & : \text{null} \\
\text{Int} & : \text{null} \\
\text{Int} & : \text{null} \\
\text{Int} & : \text{null}
\end{align*}
\]

Definition (Local type binding translation).

\[
\begin{align*}
\text{Int} & : \text{Int} \\
\text{null} & : \text{null} \\
\text{Int} & : \text{null} \\
\text{Int} & : \text{null} \\
\text{Int} & : \text{null} \\
\text{Int} & : \text{null}
\end{align*}
\]

Definition (Heap type binding translation).

\[
\begin{align*}
\text{Int} & : \text{Int} \\
\text{null} & : \text{null} \\
\text{Int} & : \text{null} \\
\text{Int} & : \text{null} \\
\text{Int} & : \text{null} \\
\text{Int} & : \text{null}
\end{align*}
\]
Definition (Free Variables of $\Gamma, \Sigma$).
\[
\begin{align*}
\text{FV}(\emptyset) &= \emptyset \\
\text{FV}(x:T(\ell); \Gamma) &= \{ \ell \} \cup \text{FV}(\Gamma) \\
\text{FV}(\ell \mapsto x:T * \Sigma) &= \{ \ell, x \} \cup \text{FV}(\Sigma)
\end{align*}
\]

Definition (Pure Value Types). In order to show measure well-formedness, we extend the language of types with products ($T \times T'$) and unions with null ($T + \text{null}$), and define:
\[
\text{SnapType} : \text{Type} \times \text{Heap} \rightarrow \text{Type}
\]
\[
\begin{align*}
\text{SnapType}(\emptyset, \Sigma) &= \{ \langle \ell \rangle \times \text{SnapType}(T, \Sigma) \mid \ell \mapsto x:T \in \Sigma \} \\
\text{SnapType}(\exists \ell, \Sigma) &= \text{SnapType}(\emptyset, \Sigma) \cup \text{SnapType}(\text{null}, \Sigma) \\
\text{SnapType}(T, \Sigma) &= T
\end{align*}
\]

Definition (Pure Values from Type Definitions). For each type $T$ we denote the pure values associated with that type as $T^*$, which we define inductively:
\[
\begin{align*}
\text{null} \in \text{null}^* \\
e \in \mathbb{Z} \\
e \in \text{ind}^* \\
e \in \alpha^*
\end{align*}
\]

\[
\begin{align*}
e \in \{ f_{T_1} \}^* \\
e \in T_1^* \\
e \in T_2^* \\
e \in T_1^* \lor \ne \in T_1^* \\
e \in (T_1 \times T_2)^* \\
e \in (T_1 + T_2)^*
\end{align*}
\]

Definition (Constraints from Pure Values). We consider elements of $C^*$ as “snapshots” of some heap. We “restore” these snapshots by mapping them to assertions with the functions:
\[
\begin{align*}
\text{Snapshot} : \mathbb{Z} \times \text{Value} \rightarrow \text{Prop} \\
\text{Walk} : \text{Value} \rightarrow \text{Value} \times \mathcal{P}(\mathbb{Z} \times \text{Value})
\end{align*}
\]

\[
\begin{align*}
\text{Snapshot}(\ell, x) &= ((\ell \neq \text{null}) \Rightarrow (\ell \mapsto e \oplus (x, e') \in h \Rightarrow e')) \\
\text{Walk}(x) &= \text{Walk}(x)
\end{align*}
\]

Definition (Assertions from Type Definitions). Given the definition $C[\alpha] \doteq \exists! \Sigma. x:T$, define the assertion:
\[
c(T_\alpha, x, f, e)(\Sigma) \doteq \exists x, e \in \text{FV}(\Sigma). (\ell \mapsto x \in \text{FV}(T_\alpha / \alpha[T]) \lor e \in \text{FV}(T_\alpha / \alpha[\Sigma]))
\]

Definition (Interpretation of Measures). Let $m(x:C[\alpha]) \doteq e$ and $s$ be a mapping from variables to values. Define $\text{exp} \subseteq \{ e \}$:
\[
\begin{align*}
[v]_{\text{exp}} &= v \\
[x.f]_{\text{exp}} &= s(x).f \\
[f(e)]_{\text{exp}} &= f([e]_{\text{exp}}) \\
[e_1 \oplus e_2]_{\text{exp}} &= [e_1]_{\text{exp}} \oplus [e_2]_{\text{exp}} \\
\text{otherwise} &= false
\end{align*}
\]

Definition (Interpretation of type contexts/worlds).
\[
\begin{align*}
\llbracket \Gamma \rrbracket &= \bigwedge e \in \text{ind} \Gamma \in \text{type } e \in \Gamma \\
\llbracket \Sigma \rrbracket &= \bigwedge \ell \mapsto x:T \in \Sigma \\
\llbracket \Gamma, \Sigma \rrbracket &= [\llbracket \Gamma \rrbracket] \land [\llbracket \Sigma \rrbracket]
\end{align*}
\]

Definition (Interpretation of procedure declarations). Assume that $f(x) = s$. Given $S = \forall x. \Sigma \in (x:T)/\Sigma \Rightarrow \exists x_0. x_0:T_\alpha/\Sigma_\alpha$, $\text{Pre}(S) = \llbracket x:T \rrbracket \land [\llbracket \Sigma \rrbracket]$

\[
\begin{align*}
\text{Post}(S) &= \llbracket x_0:T_\alpha \rrbracket \land [\llbracket \Sigma_\alpha \rrbracket] \\
\text{Body}(f) &= s
\end{align*}
\]

and the variables appearing in $\text{Post}(S)$ and not $\text{Pre}(S)$ are considered to be modified by $f$.

Definition (Interpretation of procedure contexts).
\[
\llbracket f:S; \Phi \rrbracket = \{ \text{Pre}(S) \} \text{ Body}(f) \} \{ \text{Post}(S) \}; \llbracket \Phi \rrbracket
\]

Type Soundness

We assume the following:

1. The set of program variables $(x, y, etc.)$ is disjoint from the set of symbols used to denote locations $(\ell, l', etc.)$.
2. All programs are in single static assignment form. The only variables which have more than one static assignment are “phi” variables which are assigned once in each branch of an “if” statement.

Lemma 1. If for every $x:T \in \Sigma$, $\Gamma \vdash T$, then the only symbols that appear in $[\llbracket \Gamma \rrbracket]$ are variables $(x, y)$, arithmetic and equality symbols, and uninterpreted functions, and no variable appears bound twice.

Proof. By induction on the type well-formedness judgement.

Corollary. The location $\ell$ only appears in $[\llbracket \Gamma \rrbracket]$ if there exists $x:T \in \Gamma$ and $T = \{ v: \ell \mid p \}$ or $T = \{ v: \ell \mid p \}$.

Lemma 2. If $\Gamma \vdash \Sigma$, then for each $x:T \in \Sigma$:

1. No binding $x:T'$ appears in $\Gamma$.
2. $[\llbracket \Sigma \rrbracket]$ contains exactly one sub-assertion of the form $\ell \mapsto x$.

Proof. By definition of $[\llbracket \Sigma \rrbracket]$ and induction on the well-formedness judgement.

Lemma 3. If $\Gamma, \Sigma \vdash S$, then the formal arguments or free variables of the output world of $S$ do not appear free in $\Gamma$ or $\Sigma$.

Proof. By assumption of WF-FUN.

Corollary. The location $\ell$ only appears in $[\llbracket \Sigma \rrbracket]$ if there exists $\ell' \mapsto x:T \in \Sigma$ and $\ell' = \ell$ or $T = \{ v: \ell \mid p \}$ or $T = \{ v: \ell \mid p \}$.  

2 2015/8/12
Lemma 4. If $\Gamma \vdash M : C[\tau]$ and $v \in C^*(A)$, then $\llbracket [v/x]e \rrbracket_{exp} \in \mathcal{V}$. 

Proof. By induction on the $\Gamma \vdash M : T$ judgement.

Lemma 5. [Subtyping]
1. If $\Gamma \vdash T_1 \leq T_2$ then $\llbracket \Gamma \rrbracket \Rightarrow \llbracket T_1 \rrbracket \Rightarrow \llbracket T_2 \rrbracket$
2. If $\Gamma \vdash \Sigma_1 \leq \Sigma_2$ then $\llbracket \Gamma \rrbracket \Rightarrow \llbracket \Sigma_1 \rrbracket \Rightarrow \llbracket \Sigma_2 \rrbracket$

Proof. By induction on the subtyping derivation.

Case $\Gamma \vdash \{\nu : \tau \mid p\} \leq \{\nu : \tau \mid p'\}$
By assumption,

Unfolding the definition of $\llbracket \cdot \rrbracket$, and by the form of each $T_i$, each $p_i$ only occurs positively in $c([\nu : \tau \mid p], \ell, x)$.

Destructing $T$ as $[\nu : \tau \mid p]$ and $\overline{T}$ as $[\nu : \tau \mid p']$:

$\llbracket \nu : \{C[T_i] \mid p_i\} \rrbracket = pC \land \exists \epsilon. c(T, \ell, \nu)$

which implies

$\llbracket \nu : \{\ell \mid p'\} \rrbracket = p' \land \nu = l \land \nu \neq \text{null}$

Case $\Gamma \vdash \{\nu : \# \mid p\} \leq \{\nu : \# \mid p'\}$
By assumption,

$\llbracket \Gamma \rrbracket \Rightarrow p \Rightarrow (p' \land \nu = \text{null})$

and thus

$\llbracket \Gamma \rrbracket \Rightarrow p \Rightarrow (p' \land (\nu \neq \text{null} \land \nu = l) \lor \nu = \text{null})$

which is equivalent to $\llbracket \nu : \{\nu : \# \mid p'\} \rrbracket$.

Case $\Gamma \vdash \{\nu : \# \mid p\} \leq \{\nu : \# \mid p'\}$
By assumption,

$\llbracket \Gamma \rrbracket \Rightarrow p \Rightarrow (p' \land \nu = \text{null})$

and thus

$\llbracket \Gamma \rrbracket \Rightarrow p \Rightarrow (p' \land (\nu \neq \text{null} \land \nu = l) \lor \nu = \text{null})$

which is equivalent to $\llbracket \nu : \{\nu : \# \mid p'\} \rrbracket$.

Case $\Gamma \vdash \text{emp} \leq \text{emp}$
The heaps are equivalent and their domains are empty, so the conclusion is trivially true.

Case $\Gamma \vdash \Sigma \ast \ell \mapsto x : T \leq \Sigma' \ast \ell \mapsto x : T'$
By the inductive hypothesis,

$\llbracket \Gamma \rrbracket \Rightarrow \llbracket \Sigma \rrbracket \Rightarrow \llbracket \Sigma' \rrbracket$

and by assumption and the definition of $\llbracket \cdot \rrbracket$,

$\llbracket \Gamma \rrbracket \Rightarrow \llbracket \ell \mapsto x : T \rrbracket \Rightarrow \llbracket \ell \mapsto x : T' \rrbracket$

hence,

$\llbracket \Gamma \rrbracket \Rightarrow \llbracket \ell \mapsto x : T \ast \Sigma \rrbracket \Rightarrow \llbracket \ell \mapsto x : T' \ast \Sigma' \rrbracket$

and thus,

$\llbracket \Gamma \rrbracket \Rightarrow \llbracket \ell \mapsto x : T \ast \Sigma \ast \ell \mapsto x : T' \ast \Sigma' \rrbracket$

As a consequence of our interpretation of typing contexts, the following lemma immediately follows:

Lemma 6. Assume that $\Phi$, $\Gamma$, and $\Sigma$ are (respectively) well formed global, local, and heap contexts. We may then write the denotation of these typing contexts in the following way:

$\llbracket \Gamma, \Sigma \rrbracket = G \ast H$

where the following hold:

1. $G$ is pure.
2. $G$ is a conjunction of the assertions

$G_x \equiv [x : T]$

for each $x : T \in \Gamma$. $G$ may thus be written $\cdots \land G_x \land \cdots$ for each such binding in $\Gamma$.
3. $H$ is a conjunction (via $\ast$) of the assertions

$H_\ell \equiv (\ell = \text{nil} \lor [\ell \mapsto x : T])$

for each $\ell \mapsto x : T \in \Sigma$. $G$ may thus be written $\cdots H_\ell \cdots$ for each such binding in $\Sigma$. 

2018/3/12
Proof. Follows immediately from the definition of $\llbracket - \rrbracket$.

Lemma 7.
If $\Gamma, \Sigma \vdash e : T$
Then $\llbracket \Gamma, \Sigma \rrbracket = G * H$
such that
1. $G$ is pure.
2. $G \Rightarrow [ e : T ]$

Proof. By induction on the expression typing judgment. In particular, we make use of the fact that the typing and subtyping rules for pure expressions only depend on $\Gamma$. Subsumption requires application of ??.

Lemma 8. Suppose that $\Gamma, \Sigma \vdash x : \langle \ell \rangle$ and that $\Gamma \vdash \Sigma = \Sigma_0 \ast \ell \ast y : T$. It then follows that $\llbracket \Gamma, \Sigma \rrbracket = G * H * H_\ell$ where $G$ is pure and:
(i) $G \Rightarrow \{ x \mapsto y : T \}$
(ii) $H = H' * H_\ell * H''$ such that
$G * H_\ell \Rightarrow \{ x \mapsto y : T \}$
and furthermore, if $T$ is a record type,
$G * \{ x \mapsto y : T \} * H_\ell \Rightarrow x \mapsto y$
equivalently:
$G * \{ x \mapsto y : T \} * H_\ell \Rightarrow x \mapsto \langle f_1 : \text{field}(y, f_1) \rangle$

Proof. Follows from ?? and the definition of $\llbracket - \rrbracket$.

Lemma 9. [Folding] If $\Gamma \vdash x : T_1 / \Sigma_1 \triangleright x : T_2 / \Sigma_2$ then
$\llbracket \Gamma \rrbracket \Rightarrow [ \ell \mapsto x : T_1 * \Sigma_1 ] \Rightarrow \exists FV(\Sigma_2). \{ \ell \mapsto x : T_2 * \Sigma_2 \} \land \exists y$. Snapshot$(\ell, y)$

Proof. By induction on the judgement derivation.

Case F-BASE
By the subtyping hypothesis,
$\llbracket \Gamma \rrbracket \land \{ \ell \mapsto x : T_2 \} \Rightarrow \{ \ell \mapsto x : T_2 \}$
And thus, by letting every other $\ell \in \text{Dom}(\Sigma_2) = \text{null}$,
$\exists FV(\Sigma_2). \{ \ell \mapsto x : T_2 * \Sigma_2 \} \land \exists y$

Since $T_1$ contains no pointers, and since we are assuming the satisfiability of $\{ x : T_1 \}$, we can construct $y$ from the base type of $T_1$ and its refinement.

Case F-REF
By the subtyping hypothesis, where $T_1 = \{ \nu : \langle \ell' \rangle | p \}$,
$\llbracket \Gamma \rrbracket \Rightarrow [ x : T_1 ] \Rightarrow [ x : T_2 ]$

And by the inductive hypothesis,
$\llbracket \Gamma \rrbracket \Rightarrow [ \ell' \mapsto x : T' * \Sigma'_2 ] \Rightarrow \exists FV(\Sigma_2). \{ \ell' \mapsto x : T' * \Sigma'_2 \} \land \exists s$. Snapshot$(\ell', s)$

Combining the two gives us
$\exists FV(\Sigma_2). \{ \ell \mapsto y : T_2 * \Sigma_2 \}$

We construct $y = (\ell', s)$ so that $y \in (\ell')^*$ to yield
Snapshot$(\ell, y)$

Case F-REF
By hypothesis,
$((x \neq \text{null}) \land [ \Gamma, \ell \mapsto x : T * \Sigma'_1 ] \Rightarrow \exists FV(\Sigma'_2), \{ \ell \mapsto x : T' * \Sigma'_2 \} \land \exists y')$. Snapshot$(\ell, y')$

In the first case, $(x \neq \text{null})$ we let
$y = (\ell, y')$

In the second case, $y = \text{null}$.

We combine both hypotheses and $y$ constructions to conclude:
$\exists FV(\Sigma_2). \{ \ell \mapsto x : T_2 * \Sigma_2 \}$
and, by definition, Snapshot$(\ell, y)$.

Case F-HEAP
By hypothesis,
$\llbracket \Gamma, \ell \mapsto x : T * \Sigma_1 \rrbracket \Rightarrow \exists FV(\Sigma_2). \{ \ell \mapsto x : T * \Sigma_2 \} \land \exists x_i$. Snapshot$(\ell, x_i)$

for each field$(x, f_1)$ in $T_1$. If we substitute field$(x, f_1)$ for $x$ in each conclusion, we conclude
$\{ \ell \mapsto x : T_2 * \Sigma_2 \}$
and we construct $y = \langle f_1 : x_i \rangle$.

Lemma 10. Given the definition
$C[a] = \exists! \Sigma. x : T$,
and the set of corresponding pure values, $C^*(A)$:
$\text{Snapshot}(\ell, y) \land \{ \ell \mapsto x : T * \Sigma \} \Rightarrow x_f \in C^*(A)$

Proof. By contrapositive. Assume $x_f \notin C(A)^*$. Then
$x_f \notin \text{Pure}_C(A, C^*, T)$.\Sigma$

The proof is by induction on the derivation of $x_f \notin \text{Pure}_C(A, C^*, T)$.\Sigma$
The base case starts with the root of Snapshot$(\ell, x_f)$

$\text{Snapshot}(\ell, x_f) = \ell \mapsto e \ast \cdots$

If $x_v \notin \text{Pure}_C(A, C^*, T)\Sigma$ then either $e$ is not a record value, trivially showing $\llbracket \ell \mapsto x : T * \Sigma \rrbracket$, or $x_v$ contains some field $f$ with value $e_f$ such that:

(i) $f : T_f \in T$;
(ii) and $e_f \notin \text{Pure}_C(A, C^*, T_f)$\Sigma$

Assuming $e_f \neq (\ell', e')$, if $e_f \notin \text{Pure}_C(A, C^*, T_f)$\Sigma$, then the assertion in Snapshot$(\ell, x_f)$, $\ell \mapsto \langle \ldots | e_f | \ldots \rangle$, must not have corresponding $\{ \ell \mapsto x : T \}$ by the definition of $\text{Pure}_C(A, C^*, T)\Sigma$. If $e_f = (\ell', e')$ for some $\ell', e'$, then $e' \notin \text{Pure}_C(A, C^*, T_f)$\Sigma$ However, by the same reasoning as above, Snapshot$(\ell', e')$ must not be equisatisfiable with $\llbracket \ell \mapsto x : T * \Sigma \rrbracket$

Theorem. [Statement Typing]
If $\Phi, \Gamma, \Sigma \vdash s : \Gamma' / \Sigma'$ then
$\{ \Phi \} \vdash \{ [ \Gamma, \Sigma ] \} \{ [ \Gamma', \Sigma' ] \}$

Proof. By induction on the typing derivation of $s$. 

2018/5/12
Case $x = e$:
   By assumption, $x$ does not appear in $\Gamma$. Since $e$ is well-typed, $x \notin \text{FV}(e)$. By the frame rule, since $x \notin \Gamma$ and $x \notin \Sigma$, we derive the following rule:

$$\Gamma, \Sigma \vdash (x = e)$$

We then perform the deduction by the definition of $\vdash$:

$$\Gamma, \Sigma \vdash [x : (\nu : s \mid \nu = e) ; \Gamma, \Sigma]$$

Case $s; s'$
   By hypothesis,
   $$\Phi \vdash \{ [\Gamma, \Sigma] \} s_1 \{ [\Gamma', \Sigma'] \}$$
   and
   $$\Phi \vdash \{ [\Gamma', \Sigma'] \} s_2 \{ [\Gamma'', \Sigma''] \}$$
   By straightforward application of the sequence rule, we immediately conclude
   $$\Phi \vdash \{ [\Gamma, \Sigma] \} s_1 ; s_2 \{ [\Gamma'', \Sigma''] \}$$

Case if $e$ then $s_1$ else $s_2$:
   By hypothesis:
   $$\Phi \vdash \{ e \land [\Gamma, \Sigma] \} s_1 \{ [\Gamma_1, \Gamma, \Sigma_1 \land \Sigma_1'] \}$$
   and
   $$\Phi \vdash \{ \neg e \land [\Gamma, \Sigma] \} s_2 \{ [\Gamma_2, \Gamma, \Sigma_2 \land \Sigma_2'] \}$$
   For each $x_i$, by $??$
   $$[\Gamma_1 ; \Gamma] \Rightarrow [x_i : T'] \quad \text{and} \quad [\Gamma_2 ; \Gamma] \Rightarrow [x_i : T']$$
   and by $??$
   $$[\Gamma_1 ; \Gamma] \Rightarrow [\Sigma_1 ] \Rightarrow [\Sigma' ] \quad \text{and} \quad [\Gamma_2 ; \Gamma] \Rightarrow [\Sigma_2 ] \Rightarrow [\Sigma' ]$$
   Combining these facts allows us to conclude,
   $$[\Gamma_1 ; \Gamma, \Sigma_1 ] \Rightarrow [x_i : T'_i ; \Gamma, \Sigma' ] \quad \text{and} \quad [\Gamma_2 ; \Gamma, \Sigma_2 ] \Rightarrow [x_i : T'_i ; \Gamma, \Sigma' ]$$
   and thus, by consequence,
   $$\Phi \vdash \{ e \land [\Gamma, \Sigma] \} s_1 \{ [x_i : T'_i ; \Gamma, \Sigma' ] \}$$
   and
   $$\Phi \vdash \{ \neg e \land [\Gamma, \Sigma] \} s_2 \{ [x_i : T'_i ; \Gamma, \Sigma' ] \}$$
   hence, by the conditional rule,
   $$\Phi \vdash \{ [\Gamma, \Sigma] \} \text{if } e \text{ then } s_1 \text{ else } s_2 \{ [x_i : T'_i ; \Gamma, \Sigma' ] \}$$

Case concr $(x,z)$:
   Letting $[\Gamma, \Sigma] = G * H$ by $??$, by assumption
   $$\{ G * H \}$$
   By hypothesis and $??$,
   $$\{ G * H * (x = \ell \land \ell \neq null) \land \ell \Rightarrow y : T_y \}$$
   By consequence, assuming without loss of generality that $T_y$ is a record type, substitution on the rule for concr, and applying the frame rule,
   $$\{ G * H * (x = \ell \land \ell \neq null) * x \Rightarrow y * [ y : T_y ] \}$$
   $$\text{conc}r(x,z)$$
   $$\{ G * H * (x = \ell \land \ell \neq null) * x \Rightarrow y * [ y : T_y ] \land z = y \}$$
   since $z$ is a fresh variable. By definition of $[-]$ and $T_y$, since $z$ does not appear free in $G, H$ or $T_y$.
   $$\{ [ y : T_y ; \Gamma, \ell \Rightarrow z : T_z \land \Sigma' ] \}$$

Case pad$(\ell, x)$
   Beginning with the proof rule for pad$(\ell, x)$,
   $$\vdash \{ \text{emp} \} \text{pad}(\ell, x) \{ \text{emp} \land \ell = \text{null} \land x = \text{null} \}$$
   By hypothesis, neither $\ell$ nor $x$ appear in $\Gamma$ or $\Sigma$, and thus do not appear in $[\Gamma, \Sigma]$, so we can apply the frame rule:
   $$\vdash \{ [\Gamma, \Sigma] \} \text{pad}(\ell, x) \{ [\Gamma, \Sigma] \land (\ell = \text{null} \land x = \text{null}) \}$$
   By consequence,
   $$\vdash \{ [\Gamma, \Sigma] \} \text{pad}(\ell, x) \{ [\Gamma, \Sigma] \land (\ell \Rightarrow x : T) \}$$
   And thus,
   $$\vdash \{ [\Gamma, \Sigma] \} \text{pad}(\ell, x) \{ [\Gamma, \Sigma] \land (\ell \Rightarrow x : T) \}$$

Case unfold$(\ell, \tau)$
   Letting $[\Gamma, \Sigma] = G * H * (\ell \Rightarrow x : C[\tau])$ by $??$ and by assumption,
   $$\{ G * H * ((\ell \neq \text{null}) \Rightarrow c([\tau], \ell, x)) \land ((\ell = \text{null}) \Rightarrow (x = \text{null})) \}$$
   First, assume $\ell = \text{null}$.
   Then,
   $$\{ G * H * (\ell = \text{null} \land x = \text{null}) \}$$
   And by $??$,
   $$\{ G * H * (x = \text{null}) \land \bigwedge_m [m(\text{null})]_{\exp} = e_m \}$$
   Quantifying over $x$ allows us to complete the proof via consequence and the frame rule:
   $$\{ [\Gamma, \Sigma] \}$$
   $$\exists \mbox{FV}(\ell, x). G * H * (\ell = \text{null} \land x = \text{null}) \land \bigwedge_m [m(\text{null})]_{\exp} = e_m \}$$
   $$\exists \mbox{FV}(\ell, x). G * H * (\ell = \text{null} \land x = \text{null}) \land \bigwedge_m [m(\text{null})]_{\exp} = e_m \}$$
   $$\{ \bigwedge_m [m(\text{null})]_{\exp} = e_m ; [\Gamma, \Sigma] \}$$
   Next, we assume $\ell \neq \text{null}$. By definition,
   $$\{ G * H \land \exists x_c. \exists \mbox{FV}(\Sigma_c). (\ell \Rightarrow x_c : T_c \land \Sigma) \land \text{Snapshot}(\ell, x) \}$$
   By hypothesis, $x_c$ and the free variables of $[\Sigma_c]$ have been $\alpha$-renamed and do not appear in $[\Gamma]$ or $[\Sigma]$. By the unfold axiom and the frame rule,
   $$\{ G * H * \exists x_c. \exists \mbox{FV}(\Sigma_c). (\ell \Rightarrow x_c : C[\tau][\tau'] T_c \land [T[\Sigma][\Sigma'] \land \text{Snapshot}(\ell, x) \land \text{unfold}(\ell, x_c : [\Sigma_c] \land \text{unfolds}(\Sigma_c)) \}$$
   $$\{ G * H * \exists x_c. \exists \mbox{FV}(\Sigma_c). (\ell \Rightarrow x_c : C[\tau][\tau'] T_c \land [T[\Sigma][\Sigma'] \land \text{Snapshot}(\ell, x) \land \text{unfold}(\ell, x_c : [\Sigma_c] \land \text{unfolds}(\Sigma_c)) \}$$
where $x \in C^\alpha$. Therefore, by ??, each well-formed measure on $C[\alpha]$, $m$, is defined on $x$ and we strengthen:

$$\{ G \ast H \ast \llbracket \ell \mapsto x \in \llbracket T[\alpha]T_\alpha \ast [T[\alpha]T_\alpha] \rrbracket \rrbracket \rrbracket \ast \text{Snapshot}(\ell, x) \ast \bigwedge_{m} \llbracket m(x) \rrbracket = \epsilon_m \Gamma \llbracket \llbracket \Sigma \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \rrbracket \reffertas
Case $x_r = f(\overline{x})$
Assuming $f : S$ and

$$S = \forall \overline{t}. \forall \overline{x}. (x : T) \Rightarrow \exists x_0. x_0 : T_0/\Sigma_0$$

let $P = \text{Pre}(S)$ and $Q = \text{Post}(S)$. We apply the substitution $\theta = [e_j/x_j]$ to obtain:

$$\theta P = [e_j: \theta T_j] \land [\theta \Sigma_i]$$
$$\theta Q = [x_0: \theta T_0] \land [\theta \Sigma_o]$$

By $????$,

$$[\Gamma] \Rightarrow [e_j: \theta T_j]$$
$$[\Gamma] \Rightarrow [\Sigma_m] \Rightarrow [\theta \Sigma_i]$$

and thus

$$[\Gamma] \Rightarrow [\Sigma_u \ast \Sigma_m] \Rightarrow [\Gamma] \land \theta P \ast \Sigma_u$$

which gives us, by consequence, and framing of $\Gamma$ and $\Sigma_u$, and unfolding the definition of $Q$,

$$\Phi \vdash \{[\Gamma, \Sigma_m \ast \Sigma_u] \} x = f(\overline{x}) \{[\Gamma, \Sigma_u \ast \theta \Sigma_o] \}$$

Theorem. [Procedure Typing]
If $\Phi \vdash f :: S$ then $\Phi \vdash \{\text{Pre}(S)\} \text{Body}(f) \{\text{Post}(S)\}$.

Proof. Let

$$P = \text{Pre}(S) = [x : T] \land [\Sigma_i]$$
$$Q = \text{Post}(S) = [x_0 : T_0] \land [\Sigma_o]$$

By inversion on the judgement, we must have typed a statement $s' \Rightarrow \text{return } e$ for some expression, or a sequence of statements ending in a return. By the statement typing theorem,

$$\Phi \vdash \{\text{Pre}(S)\} s \{[\Gamma_s, \Sigma_s]\}$$

Applying $???$ and $???$ to the hypotheses of the return typing rule, we determine

$$[\Gamma_s, \Sigma_s] \Rightarrow [e/x_0] [x_0 : T_0]$$
$$[\Gamma_s, \Sigma_s] \Rightarrow [e/x_0] [\Sigma_o]$$

and thus

$$\Phi \vdash \{[\Gamma_s, \Sigma_s]\} \text{return } e \{\text{Post}(S)\}.$$
Title Text
Subtitle Text, if any

Abstract
This is the text of the abstract.

Categories and Subject Descriptors  CR-number [subcategory]:
third-level

General Terms  term1, term2

Keywords  keyword1, keyword2

1. Introduction
The text of the paper begins here.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
More text.
Lots of text.
More text.
Lots of text.

A. Appendix Title
This is the text of the appendix, if you need one.

Acknowledgments
Acknowledgments, if needed.

References
[1] P. Q. Smith, and X. Y. Jones. "reference text..."