

Sound and Complete Type Inference in BitC

SRL Technical Report SRL2008-02

Swaroop Sridhar Jonathan S. Shapiro Scott F. Smith
swaroop@cs.jhu.edu shap@eros-os.org scott@cs.jhu.edu
Department of Computer Science
The Johns Hopkins University
3400 N.Charles Street. 224 NEB. Baltimore, MD 21218.

Abstract

This paper introduces a new type system designed for safe systems programming. The type system features a new mutability model that combines unboxed types with a consistent typing of mutability. The type system is provably sound, supports polymorphism, and eliminates the need for alias analysis to determine the immutability of a location. A sound and complete type inference algorithm for this system is presented.

1 Introduction

Recent advances in the theory and practice of programming languages have resulted in modern languages and tools that provide certain correctness guarantees regarding the execution of programs. However, these advances have not been effectively applied to the construction of *systems programs*, the core components of a computer system. One of the primary causes of this problem is the fact that existing languages do not simultaneously support modern language features — such as static type safety, type inference, higher order functions and polymorphism — as well as features that are critical to the correctness and performance of systems programs such as prescriptive data structure representation and mutability. In this paper, we endeavor to bridge this gap between modern language design and systems programming. We first discuss the support for these features in existing languages, identify the challenges in combining these feature sets and then describe our approach toward solving this problem.

Representation Control A systems programming language must be expressive enough to specify details of representation including boxed/unboxed data-structure layout and stack/heap allocation. For systems programs, this is both a correctness as well as a performance requirement. Systems programs interact with the hardware through data structures such as page tables whose representation is dictated by the hardware. Conformance to these representation specifications is necessary for correctness. Languages like ML [16] intentionally omit details of representation from the language definition, since this greatly simplifies the mathematical description of the language. Compilers like TIL [23] implement unboxed representation as a discretionary optimization. However, in systems programs, statements about representation are *prescriptive*, not *descriptive*. Formal treatment of representation is required in systems programming languages.

Systems programs also rely on representation control for performance since it affects cache locality and paging behavior. This expressiveness is also crucial for interfacing with external C [9] or assembly code and data. For example, a careful implementation of the TCP/IP protocol stack in Standard ML incurred a substantial overhead of up to 10x increase in system load and a 40x slowdown in accessing external memory relative to the equivalent C implementation [1, 3]. This shows that representation control is as important as, or even more important than, high level algorithms for the performance of systems tasks.

Complete Mutability One of the key features essential for systems programming is support for mutability. The support for mutability must be ‘complete’ in the sense that any location — whether on the stack, heap, or within other unboxed structures — can be mutated. Allocation of mutable cells on the stack boosts performance because (1) the top of the stack is typically accessible from the data cache (2) stack locations are directly addressable and

therefore do not require the extra dereferencing involved in the case of heap locations (3) stack allocation does not involve garbage collection overhead. This is particularly important for high confidence and/or embedded kernels as they cannot tolerate unpredictable variance in overhead caused by heap allocation and collection. ML-like languages require all mutable (`ref`) cells to reside on the heap. In pure languages like Haskell [14], the support for mutability is even more restrictive than ML. These restricted models of mutability are insufficiently expressive from a systems programming perspective.

Consistent Mutability The mutability support in a language is said to be ‘consistent’ if the (im)mutability of every location is invariant across all aliases over program execution. In this model, there is a sound notion of immutability of locations. This benefits tools that perform static analysis or model checking because conclusions drawn about the immutability of a location need never be conservative. It also increases the amount of optimization that a compiler can safely perform without complex alias analysis. Polymorphic type inference systems such as Hindley-Milner algorithm [15] also rely on a sound notion of immutability. ML supports consistent mutability since types are definitive about the (im)mutability of every location. In contrast, C does not support this feature. For example, in C it is legal to write:

```
const bool *cp = ...; bool *p = cp; *p = false; // OK!
```

The alleged “constness” of the location pointed to by `cp` is a local property (only) with respect to the alias `cp` and not a statement of true immutability of the target location. The analysis and optimization of critical systems programs can be improved by using a language with a consistent mutability model.

Type Inference and Polymorphism Type inference achieves the advantages of static typing with a lower burden on the programmer, facilitating rapid prototyping and development. Polymorphic type inference (c.f. ML or Haskell) combines the advantages of static type safety with much of the convenience provided by dynamically typed languages like Python [18]. Automatic inference of polymorphism simplifies generic programming, and therefore increases the reuse and reliability of code. Safe languages like Java [17], C# [4], or Vault [2] do not support type inference. Cyclone [10] features partial type inference and supports polymorphism only for functions with explicit type annotations.

The following table summarizes the support available in existing languages for the above features and static type safety:

	C/Asm	Safe-C	CCured	Cyclone	Vault	Java	ML	Haskell
Representation	✓			✓	✓			
Complete Mutability	✓	✓	✓	✓	✓	✓		
Consistent Mutability							✓	✓
Static Type Safety				✓	✓	✓	✓	✓
Poly. Type Inference							✓	✓

In this paper, we present a new type system and formal foundations for a safe systems programming language that supports all of the above features.

The combination of mutability and unboxed representation presents several challenges for type inference. Mutability is an attribute of the *location* storing a value and not the value itself. Therefore, two expressions across a copy boundary (ex: arguments copied at a function call) can differ in their mutability. We refer to this notion of mutability compatibility of types as *copy compatibility*. Copy compatibility creates ramifications for syntax-directed type and mutability inference. Type inference is further complicated due to well known problems with the interaction of mutability and polymorphism [25]. This has forced a second-class treatment of mutability in ML-like languages and a lack of inferred polymorphism in others.

We present a sound and complete polymorphic type inference algorithm for a language that supports consistent and complete mutability. In order to overcome the challenges posed by copy compatibility, the underlying type system uses a system of constrained types that range over mutability and polymorphism. Safety of the type system as well as the soundness and completeness of the type inference algorithm have been proved.

2 Informal Overview

In this section, we give an informal description of our type system and inference algorithm. For purposes of presentation in this paper, we define \mathbb{B} , a core systems programming language calculus. \mathbb{B} is a direct expression of lambda

calculus with side effects, extended to be able to reflect the semantics of explicit representation.

Identifiers	$x ::= y \mid z \mid \dots$	Vars	$\alpha ::= \alpha \mid \beta \mid \gamma \mid \delta \mid \varepsilon \mid \dots$
Booleans	$b ::= true \mid false$	$\varsigma ::= \alpha \mid \Psi\alpha$	
Indices	$i ::= 1 \mid 2 \mid !i$	Types	$\rho ::= \alpha \mid \text{unit} \mid \text{bool} \mid \tau \rightarrow \tau$ $\mid \tau \times \tau \mid \uparrow\tau \mid \Psi\rho$
Values	$v ::= () \mid b \mid \lambda x.e \mid (v, v)$	$\varrho ::= \rho \mid \alpha \mid \rho$	
Left Expr	$l ::= x \mid e^\wedge \mid l.i \mid l:\tau$	$\tau ::= \varrho \mid \varsigma \mid \rho$	
Expressions	$e ::= v \mid x \mid e e \mid l := e$ $\mid \text{if } e \text{ then } e \text{ else } e \mid e:\tau$ $\mid \text{dup}(e) \mid e^\wedge \mid (e, e) \mid e.i$ $\mid \text{let } x[:\tau] = e \text{ in } e$	Scheme	$\sigma ::= \tau \mid \forall \bar{\alpha}.\tau \setminus \mathcal{D}$
		Ct. Set	$\mathcal{D} ::= \emptyset \mid \{\star_x^\varkappa(\tau)\} \mid \mathcal{D} \cup \mathcal{D}$
		Kinds	$\varkappa ::= \kappa \mid \psi \mid \forall$

The type $\uparrow\tau$ represents a reference (pointer) type and $\Psi\rho$ represents a mutable type. The expression $\text{dup}(e)$, where e has type τ , returns a reference of type $\uparrow\tau$ to a heap-allocated *copy* of the value of e . The \wedge operator is used to dereference heap cells. Pairs $(,)$ are *unboxed* structures whose constituent elements are contiguously allocated on the stack, or in their containing data-structure. $e.1$ and $e.2$ perform selection from pairs. We define $!2 = 2$ and $!1 = 1$. The `let` construct can be used for allocating (possibly mutable) stack variables and to create let-polymorphic bindings. `let $x[:\tau] = e$ in e` represents optional type qualification of let-bound variables.

The Mutability Model \mathbb{B} supports consistent, complete mutability. The mutability support is complete since the `:=` operator mutates both stack locations (let-bound locals, function parameters) and heap locations (`dup`-ed values). It can also perform in-place updates to individual fields of unboxed pairs. The mutability support is consistent since we impose the “one location, one type” rule. For example, in the following expression,

`let $cp : \uparrow\text{bool} = \text{dup}(true)$ in let $p : \uparrow\Psi\text{bool} = cp$ (* Error *)`
 cp has the type reference to `bool` ($\uparrow\text{bool}$), which is incompatible with that of p , reference to mutable-`bool` ($\uparrow\Psi\text{bool}$). Unlike ML, `:=` does not dereference its target. The expressions that can appear on the left of an assignment `:=` are restricted to left expressions (defined by the above grammar). This not only preserves the programmer’s mental model of the relationship between locations storage, but also ensures that compiler transformations are semantics preserving.

Copy Compatibility \mathbb{B} is a call-by-value language, and supports copy compatibility, which permits locations across a copy boundary to differ in their mutability. For example, in the following expression:

`let $fn\ xn = \lambda x.(x := false)$ in let $y : \text{bool} = true$ in $fn\ xn\ y$`
the type of $fn\ xn$ is $(\Psi\text{bool}) \rightarrow \text{unit}$, whereas that of the actual argument y is `bool`. Since x is a *copy* of y and occupies a different location, this expression is type safe. Thus, we write $\text{bool} \cong \Psi\text{bool}$, where \cong indicates copy compatibility.

Copy compatibility must not extend past a reference boundary in order to ensure that every location has a unique type. We define copy compatibility for \mathbb{B} as:

$$\frac{}{\tau \cong \tau} \quad \frac{\tau_1 \cong \tau_2}{\tau_2 \cong \tau_1} \quad \frac{\tau_1 \cong \tau_2 \quad \tau_2 \cong \tau_3}{\tau_1 \cong \tau_3} \quad \frac{\tau \cong \rho}{\tau \cong \Psi\rho} \quad \frac{\tau_1 \cong \tau'_1 \quad \tau_2 \cong \tau'_2}{\tau_1 \times \tau_2 \cong \tau'_1 \times \tau'_2}$$

Copy compatibility is allowed at all positions where a copy is performed: at argument passing, new variable binding, assignment, and basically in all expressions where a left-expression is not expected or returned. For example, the expression $(x : \tau) : \Psi\tau$ is ill typed, but the branches of a conditional can have different but copy compatible types as in `if $true$ then $a : \tau$ else $b : \Psi\tau$` .

2.1 Type Inference

We now consider the problem of designing a type inference algorithm for \mathbb{B} . Due to copy compatibility, it is no longer possible to infer a unique (simple) type for all expressions. For example, in the expression `let $p = true$` , we know that the type of the literal `true` is `bool`, but the type of p could either be `bool` or Ψbool . Therefore, unlike ML, we cannot use a straightforward syntax-directed type inference algorithm in \mathbb{B} .

It is natural to ask why mutability should be inferred at all. That is: why not require explicit annotation for all mutable values, and infer immutable types by default? Unfortunately, in a language with copy compatibility, this will result in a proliferation of type annotations. Constructor applications, polymorphic type instantiations, accessor functions, *etc.* will have to be explicitly annotated with their types. For example, if fst is an accessor function that returns the first element of a pair, and m is a variable of type Ψbool , we will have to write:

`let $xyz = \text{dup}(fst\ (m, false) : \Psi\text{bool} \times \text{bool}) : \uparrow\Psi\text{bool}$ in ...`

Therefore, if mutability is not inferred, it results in a substantial increase in the number of programmer annotations, and type inference becomes ineffective. It is desirable that the inference algorithm must automatically infer polymorphism (without any programmer annotations) as well, since this leads to better software engineering by maximizing code reuse.

Therefore, the desirable characteristics of a type inference algorithm for \mathbb{B} are:

- (1) It must be sound, complete, and decidable without programmer annotations.
- (2) It must automatically infer both polymorphism and mutability.
- (3) It must infer types that are intelligible to the programmer. That is, it must avoid the main drawback of many inference systems with subtyping, where the inferred principal type is presented as a set of equations and inequations.

In order to address the above requirements, we propose a variant of the Hindley-Milner algorithm [15]. This algorithm uses constrained types that range over mutability and polymorphism in order to infer principal types for \mathbb{B} programs.

Polymorphism Over Mutability In order to infer principal types in a language with copy compatibility, we define the following constrained types that allow us to infer types with variable mutability. Let \simeq be a equivalence relation on types such that $\rho \simeq \Psi\rho$. Let $\tau \setminus \eta$ denote a constrained type where τ is constrained by the set of (in)equations η . We write :

$\alpha\downarrow\rho \equiv \alpha \setminus \{\alpha \simeq \rho\}$: any type equal to base type ρ except for top level mutability.

$\varsigma\downarrow\rho \equiv \varsigma \setminus \{\varsigma \cong \rho\}$: any type copy compatible with ρ , where $\varsigma = \alpha$ or $\Psi\alpha$.

Now, in the expression `let p = true`, we can give p the type $\alpha\downarrow\text{bool}$. During inference, the type can later get resolved to either `bool` or Ψbool . The forms $\alpha\downarrow\rho$ and $\varsigma\downarrow\rho$ respectively provide fine grained and coarse grained control over expressing types with variable mutability. For example:

Type	Instances	Non-Instances
$\alpha\downarrow(\text{bool} \times \text{unit})$	<code>bool</code> \times <code>unit</code> , $\Psi(\text{bool} \times \text{unit})$	$\Psi\text{bool} \times \text{unit}$
$\alpha\downarrow(\text{bool} \times \text{unit})$	<code>bool</code> \times <code>unit</code> , $\Psi(\Psi\text{bool} \times \Psi\text{unit})$ $\Psi\text{bool} \times \text{unit}$, $\beta\downarrow(\text{bool} \times \text{unit})$	<code>unit</code> \times <code>bool</code>
$\Psi\alpha\downarrow(\text{bool} \times \text{unit})$	$\Psi(\text{bool} \times \text{unit})$, $\Psi(\text{bool} \times \Psi\text{unit})$	<code>bool</code> \times <code>unit</code> , $\beta\downarrow(\text{bool} \times \text{unit})$
$\alpha\downarrow\uparrow\text{bool}$	$\uparrow\text{bool}$, $\Psi\uparrow\text{bool}$	$\uparrow\Psi\text{bool}$

By embedding constraints within types, we obtain an elegant representation of constrained types that are self contained. The programmer is just presented a type, rather than a type associated with a set of unsolved inequations. Every type of the form $\varsigma\downarrow\rho$ can be realized through a canonical representation using $\alpha\downarrow\rho$ types. However, types of the form $\varsigma\downarrow\rho$ are critical for type inference. For example, the type $\alpha\downarrow\beta$ represents a type that is compatible with β , even if β later resolves to a more concrete (ex: pair) type.

Since we allow copy compatibility at function argument and return positions, two function types are equal regardless of the shallow mutability of the argument and return types. Therefore, we follow a convention of writing all function types with immutable types at copy compatible positions. The intuition here is that the type of a function must be described in the interface form, and must hide the “internal” mutability information. For example, the function $\lambda x.(x := \text{true})$, has external type `bool` \rightarrow `unit` even though the internal type is $\Psi\text{bool} \rightarrow \text{unit}$.

\mathbb{B} is a let-polymorphic language. At a let boundary, we would like to quantify over variables that range over mutability, in order to achieve mutability polymorphism. The next sections discuss certain complications that arise during the inference of such types, present our solution to the problem.

Soundness implications Like ML, \mathbb{B} enforces the value restriction [25] to preserve soundness of polymorphic typing. This means that the type of x in `let x = e1 in e2` can only be generalized if e_1 is an *immutable* syntactic value. For example, in the expression `let id = $\lambda x.x$` , the type of id before generalization is $\beta\downarrow(\alpha \rightarrow \alpha)$. However, giving id the generalized type $\forall\alpha.\beta.\beta\downarrow(\alpha \rightarrow \alpha)$ is unsound, since it permits expressions such as `let id = $\lambda x.x$ in (id := $\lambda x.\text{true}$, id ())` to type check. We can give id either the polymorphic type $\forall\alpha.\alpha \rightarrow \alpha$, or the monomorphic type $\beta\downarrow(\alpha \rightarrow \alpha)$. However, neither is a principal type for id .

Overloading Polymorphism Due to the above interaction of polymorphism and unboxed mutability, a traditional HM-style inference algorithm cannot defer decisions about the mutability of types past their generalization. Therefore, current algorithms fix the mutability of types before generalization based on certain heuristics — thus sacrificing completeness [22]. In order to alleviate this problem, we use a new form of constrained types that range over both mutability and polymorphism.

We introduce constraints $\star_x^\varkappa(\tau)$ to enforce consistency restrictions on instantiations of generalized types. The constraint $\star_x^\varkappa(\tau)$ requires that the identifier x only be instantiated according to the kind \varkappa , where $\varkappa = \psi$ or \forall . If $\varkappa = \psi$, the instantiation of x must be monomorphic. That is, all uses of x must instantiate τ to the same type τ' . Here, τ' is permitted to be a mutable type. If $\varkappa = \forall$, different uses of x can instantiate τ differently, but all such instantiations must be immutable. At the point of definition (`let`), if the exact instantiation kind of a variable is unknown, we add the constraint $\star_x^\kappa(\tau)$, where κ ranges over ψ and \forall . The correct instantiation kind is determined later based on the uses of x , and consistency semantics are enforced accordingly. The variable x in $\star_x^\kappa(\tau)$ represents the program point (`let`) at which this constraint is generated. We assume that there are no name collisions so that every such x names a unique program point.

In this approach, the definition of id will be given the principal constrained type:

$$\text{let } id = \lambda x.x \text{ in } e \quad id : \forall \alpha \beta. \beta \downarrow (\alpha \rightarrow \alpha) \setminus \{ \star_{id}^\kappa(\beta \downarrow (\alpha \rightarrow \alpha)) \}$$

Every time id is instantiated to type τ' in e , the constraints $\star_{id}^\kappa(\tau')$ are collected. e is declared type correct only if the set of all instantiated constraints are consistent for some κ . Note that we do not quantify over κ .

Example of e	Constraint set	Kind assignment
$(id \text{ true}, id \text{ ()})$	$\{ \star_{id}^\kappa(\text{bool} \rightarrow \text{bool}), \star_{id}^\kappa(\text{unit} \rightarrow \text{unit}) \}$	$\kappa \mapsto \forall$
$id := \lambda x.x$	$\{ \star_{id}^\kappa(\Psi(\gamma \rightarrow \gamma)) \}$	$\kappa \mapsto \psi$
$(id \text{ true}, id := \lambda x.())$	$\{ \star_{id}^\kappa(\text{bool} \rightarrow \text{bool}), \star_{id}^\kappa(\Psi(\text{unit} \rightarrow \text{unit})) \}$	Type Error
(id, id)	$\{ \star_{id}^\kappa(\beta_1 \downarrow (\alpha_1 \rightarrow \alpha_1)), \star_{id}^\kappa(\beta_2 \downarrow (\alpha_2 \rightarrow \alpha_2)) \}$	$\kappa \mapsto \psi$ or \forall

The final case type checks with either kind, under the type assignments ($\alpha_1 = \alpha_2, \beta_1 = \beta_2$) if $\kappa \mapsto \psi$ and ($\beta_1 = \alpha_1 \rightarrow \alpha_1, \beta_2 = \alpha_2 \rightarrow \alpha_2$) if $\kappa \mapsto \forall$. The intuition behind $\star_x^\kappa(\tau)$ constraints is to achieve a form of *overloading* over polymorphism and mutability. We can think of $\star_x^\kappa(\tau)$ as a type class [11] constraint that has exactly one possibly mutable instance $\star_x^\psi(\tau_m)$, and an infinite number of $\star_x^\forall(\tau_p)$ instances where all types $\overline{\tau_p}$ are immutable.

In practice, once the correct kind of instantiation is inferred, the type scheme can be presented in a simplified form to the programmer. For example, consider the expression `let f = λx.if x then () else () in (f m, f n)`, where $m : \uparrow \Psi \text{bool}$ and $n : \uparrow \text{bool}$. Here, $f : \forall \alpha \beta. \beta \downarrow (\uparrow \alpha \downarrow \text{bool} \rightarrow \text{unit}) \setminus \{ \star_f^\kappa(\beta \downarrow (\uparrow \alpha \downarrow \text{bool} \rightarrow \text{unit})) \}$. However, based on the polymorphic usage, we conclude that $\kappa \mapsto \forall$. We can now simplify the type scheme of f to obtain $f : \forall \alpha. \uparrow \alpha \downarrow \text{bool} \rightarrow \text{unit}$. Since all function types are immutable, the mutability of the argument type need not be fixed, thus preserving mutability polymorphism. In order to ensure that type inference is modular, the $\star_x^\kappa(\tau)$ constraints must not be exposed across a module boundary. For every top-level definition in a module, an arbitrary choice of $\kappa = \psi$ or $\kappa = \forall$ must be made for every surviving $\star_x^\kappa(\tau)$ constraint.

In summary, we have used a system of constrained types to design a polymorphic type inference system that meets all of the design goals set at the beginning of this section. In the next section, we present a formal description of our type system and inference algorithm.

3 Formal Description

Locations	$L ::= 1 \mid \ell$	Stack	$S ::= \emptyset \mid S, 1 \mapsto v$
Stack Loc	$l ::= l_1 \mid l_2 \mid \dots$	Heap	$H ::= \emptyset \mid H, \ell \mapsto v$
Heap Loc	$\ell ::= \ell_1 \mid \ell_2 \mid \dots$	Env.	$\Gamma ::= \emptyset \mid \Gamma, x \mapsto \sigma$
Sel Path	$p ::= i \mid p.p$	Store Typ	$\Sigma ::= \emptyset \mid \Sigma, L \mapsto \tau$
Values	$v ::= \dots \mid \ell$	Subst	$\theta ::= \langle \rangle \mid [\alpha \mapsto \tau] \mid [\kappa \mapsto \varkappa] \mid \theta \circ \theta$
Expr	$e ::= \dots \mid 1 \mid \ell$	Unf. Ctset	$\mathcal{C} ::= \mathcal{D} \mid \{ \tau = \tau \} \mid \{ \kappa = \varkappa \} \mid \mathcal{C} \cup \mathcal{C}$
Left Expr	$l ::= \dots \mid 1$	Redex	$\mathcal{R} ::= - \mid \mathcal{R} e \mid v \mathcal{R} \mid \mathcal{L} := \mathcal{R} \mid \text{dup}(\mathcal{R})$
Syn. Val	$v ::= v \mid x \mid 1 \mid (v, v)$		$\mid \mathcal{R}^\wedge \mid \text{if } \mathcal{R} \text{ then } e \text{ else } e \mid (\mathcal{R}, e)$
lvalues	$\mathcal{L} ::= 1 \mid \ell^\wedge \mid 1.p \mid \ell^\wedge.p$		$\mid (v, \mathcal{R}) \mid \mathcal{R}.i \mid \text{let}^\varkappa x = \mathcal{R} \text{ in } e$

Figure 1: Extended \mathbb{B} grammar

In order to formalize the semantics of \mathbb{B} , we extend the calculus with stack and heap locations (Fig. 1). Heap locations are first class values, but stack locations are not. Further, we annotate all `let` expressions with a kind — `letψ`: monomorphic, possibly mutable definition, and `let∀`: polymorphic definitions. The two kinds of `let` expressions

Rule	Pre-conditions	Evaluation Step
E-Rval	$S(l) = v$	$S; H; l \Rightarrow S; H; v$
E-#	$S; H; e \Rightarrow S'; H'; e'$	$S; H; \mathcal{R}[e] \Rightarrow S'; H'; \mathcal{R}[e']$
E-App	$l \notin \text{dom}(S)$	$S; H; \lambda x. e \Rightarrow S; l \mapsto v; H; e[l/x]$
E-If	$b_1 = \text{true} \quad b_2 = \text{false}$	$S; H; \text{if } b_1 \text{ then } e_1 \text{ else } e_2 \Rightarrow S; H; e_i$
E-i		$S; H; (v_1, v_2) \cdot i \Rightarrow S; H; v_i$
E-Dup	$\ell \notin \text{dom}(H)$	$S; H; \text{dup}(v) \Rightarrow S; H; \ell \mapsto v; \ell$
EL-^#	$S; H; e \Rightarrow S'; H'; e'$	$S; H; e^\wedge \Rightarrow S'; H'; e'^\wedge$
E-^	$H(\ell) = v$	$S; H; \ell^\wedge \Rightarrow S; H; v$
E-:=#	$S; H; l \Rightarrow S'; H'; l'$	$S; H; l := e \Rightarrow S'; H'; l' := e$
E-:=Stack		$S, l \mapsto v; H; l := v' \Rightarrow S, l \mapsto v'; H; ()$
E-:=Heap		$S; H, \ell \mapsto v; \ell^\wedge := v' \Rightarrow S; H, \ell \mapsto v'; ()$
E-:=S.p	$v'_i = v_i \quad S, l \mapsto v_i; H; l.p := v'_i$	$S, l \mapsto (v_1, v_2); H; l.i.p := v'_i$
E-:=H.p	$v'_i = v_i \quad S; H, \ell \mapsto v_i; \ell^\wedge.p := v'_i$	$S; H, \ell \mapsto (v_1, v_2); \ell^\wedge.i.p := v'_i$
E-Let-M	$l \notin \text{dom}(S)$	$S; H; \text{let}^\psi x = v_1 \text{ in } e_2 \Rightarrow S, l \mapsto v_1; H; e_2[l/x]$
E-Let-P		$S; H; \text{let}^\forall x = v_1 \text{ in } e_2 \Rightarrow S; H; e_2[v_1/x]$

Figure 2: Small Step Operational Semantics

τ	$\Delta(\tau)$	$\nabla(\tau)$	$\blacktriangle(\tau)$	$\blacktriangledown(\tau)$	$\mathcal{J}(\tau)$	$\{\{\tau\}\}$
α	$\Psi\alpha$	α	$\Psi\alpha$	α	α	$\{\alpha\}$
unit	Ψunit	unit	Ψunit	unit	unit	\emptyset
bool	Ψbool	bool	Ψbool	bool	bool	\emptyset
$\tau_1 \rightarrow \tau_2$	$\Psi(\tau_1 \rightarrow \tau_2)$	$\tau_1 \rightarrow \tau_2$	$\Psi(\tau_1 \rightarrow \tau_2)$	$\tau_1 \rightarrow \tau_2$	$\tau_1 \rightarrow \tau_2$	$\{\{\tau_1\}\} \cup \{\{\tau_2\}\}$
$\uparrow\tau$	$\Delta\uparrow\tau$	$\uparrow\tau$	$\Psi\uparrow\tau$	$\uparrow\tau$	$\uparrow\mathcal{J}(\tau)$	$\{\{\tau\}\}$
$\Psi\rho$	$\Delta(\rho)$	$\nabla(\rho)$	$\blacktriangle(\rho)$	$\blacktriangledown(\rho)$	$\mathcal{J}(\rho)$	$\{\{\rho\}\}$
$\tau_1 \times \tau_2$	$\Psi(\Delta(\tau_1) \times \Delta(\tau_2))$	$\nabla(\tau_1) \times \nabla(\tau_2)$	$\Psi(\tau_1 \times \tau_2)$	$\tau_1 \times \tau_2$	$\mathcal{J}(\tau_1) \times \mathcal{J}(\tau_2)$	$\{\{\tau_1\}\} \cup \{\{\tau_2\}\}$
$\alpha \downarrow \rho$	$\Delta(\rho)$	$\nabla(\rho)$	$\blacktriangle(\rho)$	$\blacktriangledown(\rho)$	$\mathcal{J}(\rho)$	$\{\alpha\} \cup \{\{\rho\}\}$
$\varsigma \downarrow \rho$	$\Delta(\rho)$	$\nabla(\rho)$	$\blacktriangle(\varsigma) \downarrow \rho$	$\blacktriangledown(\varsigma) \downarrow \rho$	$\mathcal{J}(\rho)$	$\{\varsigma\} \cup \{\{\rho\}\}$

τ	Immut(τ)	Mut(τ)	$\square(\tau)$	$\theta(\tau)$
α	false	false	false	τ if $[\alpha \mapsto \tau] \in \theta$, else α .
unit	true	false	true	unit
bool	true	false	true	bool
$\tau_1 \rightarrow \tau_2$	true	false	true	$\theta(\tau_1) \rightarrow \theta(\tau_2)$
$\uparrow\tau$	Immut(τ)	Mut(τ)	$\square(\tau)$	$\uparrow\theta(\tau)$
$\Psi\rho$	false	true	$\square(\rho)$	$\Psi\theta(\rho)$
$\tau_1 \times \tau_2$	$\text{Immut}(\tau_1) \wedge \text{Immut}(\tau_2)$	$\text{Mut}(\tau_1) \vee \text{Mut}(\tau_2)$	$\square(\tau_1) \wedge \square(\tau_2)$	$\theta(\tau_1) \times \theta(\tau_2)$
$\alpha \downarrow \rho$	false	$\text{Mut}(\blacktriangledown(\rho))$	$\square(\rho)$	$\alpha' \downarrow \theta(\rho)$ if $\theta(\alpha) = \alpha'$ ρ' if $\theta(\alpha) = \rho' \neq \alpha'$
$\varsigma \downarrow \rho$	false	$\text{Mut}(\varsigma) \vee \text{Mut}(\nabla(\rho))$	$\square(\rho)$	$\varsigma' \downarrow \theta(\rho)$ if $\theta(\varsigma) = \varsigma'$ ϱ if $\theta(\varsigma) = \varrho \neq \varsigma'$

Figure 3: Operations and Predicates on Types

have different execution semantics. We write let^κ to range over the two kinds of let expressions. This distinction is similar to Smith and Volpano's Polymorphic-C [21]. However, unlike Polymorphic-C, let -kind is *meta syntax*, and is not a part of the input program. The correct kind of let is inferred from the static type information. We do not show the semantics for type-qualified expressions as they are trivial.

Dynamic Semantics The system state is represented by the triple $S; H; e$ consisting of the stack S , the heap H , and the expression e to be evaluated. Evaluation itself is a two place relation $S; H; e \Rightarrow S'; H'; e'$ that denotes a single step of execution. Fig. 2 shows the evaluation rules for our core language. We assume that the program is alpha-converted so that there are no name collisions due to inner bindings. Following the theoretical development of [6], we give separate execution semantics for left evaluation (execution of left expressions l on the LHS of an assignment, denoted by \Rightarrow) and right evaluation (\Rightarrow) respectively.

Since the E-Dup and E-^ rules work only on the heap, we can only capture references to heap cells. Stack locations cannot escape beyond their scope since E-Rval rule performs implicit value extraction from stack locations in rvalue contexts. State updates can be performed either on the stack or on the heap (E-:=* rules). The stack is modeled as a pseudo-heap. This enables us to abstract away details such as closure-construction and garbage collection while illustrating the core semantics, as they can later be reified independently.

The execution semantics do not perform a copy operation in all cases where copy compatibility is permitted. For example, the E-If rule does not introduce a copy step in the branching expression. Since if -expressions are not lvalues, they cannot be the target of an assignment. Therefore, the value that either branch evaluates to, can itself be used in all cases where a copy of that value can be.

Static Semantics Fig. 3 defines several operators and predicates on types that we use in this section. The operators \blacktriangle and \blacktriangledown respectively increase and decrease the shallow top-level mutability of a type. \triangle and ∇ maximize / minimize the mutability of a type up to a reference or function boundary. $\bar{\triangleright}$ removes all mutability in a type up to a function boundary. We write $\tau_1 \stackrel{\blacktriangledown}{=} \tau_2$ as shorthand for $\blacktriangledown(\tau_1) = \blacktriangledown(\tau_2)$ and $\tau_1 \stackrel{\nabla}{=} \tau_2$ for $\nabla(\tau_1) = \nabla(\tau_2)$. In our algebra of types, the mutable type constructor is idempotent ($\Psi\Psi\tau \equiv \Psi\tau$). We also define the equivalences: $\alpha|\rho \equiv \alpha|\rho'$, where $\rho \stackrel{\blacktriangledown}{=} \rho'$ and $\varsigma|\rho \equiv \varsigma|\rho'$, where $\rho \stackrel{\nabla}{=} \rho'$. The predicates *Immut* and *Mut* identify types that are observably immutable and mutable respectively. The $\square(\tau)$ predicate tests if the type τ is concretizable by fixing variables that range over mutability.

$\theta\langle\tau\rangle$ denotes the application of a substitution θ on τ as defined in Fig. 3. $\theta\langle e\rangle$ performs substitutions for κ annotations in e . $\{\tau\}$ denotes the set of all constrained types and unconstrained type variables structurally present in τ . $\theta\langle\ \rangle$ and $\{\ \}$ are extended to σ, Γ, Σ , and $\{\bar{\tau}\}$ in the natural, capture-avoiding manner.

Definition I (Canonical Expressions). *An expression e is said to be canonical if all let expressions in e are annotated with one of the kinds ψ or \forall .*

Definition II (Consistency of Constrained types). *Let $\text{mtv}(\bar{\tau})$, $\text{Mtv}(\bar{\tau})$, and $\text{ntv}(\bar{\tau})$ be the set of all type variables appearing in $\{\bar{\tau}\}$ constrained by $\alpha|\rho$, by $\varsigma|\rho$ and unconstrained respectively. We say that the set of types $\{\bar{\tau}\}$ is consistent, written $\Vdash\{\bar{\tau}\}$, if: (1) For all $\{\alpha|\rho, \alpha|\rho'\} \subseteq \{\bar{\tau}\}$, we have $\rho \stackrel{\blacktriangledown}{=} \rho'$.*

(2) For all $\{\varsigma|\rho, \varsigma'|\rho'\} \subseteq \{\bar{\tau}\}$ such that $\varsigma \stackrel{\blacktriangledown}{=} \varsigma'$, we have $\rho \stackrel{\nabla}{=} \rho'$.

(3) $\text{mtv}(\bar{\tau})$, $\text{Mtv}(\bar{\tau})$, and $\text{ntv}(\bar{\tau})$ are mutually exclusive.

Definition III (Consistency of substitutions). *A substitution θ is said to be consistent over a set of types $\{\bar{\tau}\}$, written $\theta \Vdash\{\bar{\tau}\}$ if: (1) $\Vdash\theta\{\bar{\tau}\}$.*

(2) For all $\alpha|\rho \in \{\bar{\tau}\}$, we have $\theta\langle\alpha\rangle = \beta$, or $\theta\langle\alpha\rangle = \rho'$ such that $\rho' \stackrel{\blacktriangledown}{=} \theta\langle\rho\rangle$.

(3) For all $\varsigma|\rho \in \{\bar{\tau}\}$, we have $\theta\langle\varsigma\rangle = \varsigma'$, or $\theta\langle\varsigma\rangle = \rho$ such that $\rho \stackrel{\nabla}{=} \theta\langle\rho\rangle$.

Definition IV (Consistency of \star constraints). *A set of \star constraints \mathcal{D} is said to be consistent, written $\models\mathcal{D}$ if:*

(1) For all $\star_x^\forall(\tau) \in \mathcal{D}$, we have $\text{Immut}(\tau)$.

(2) For all $\star_x^\psi(\tau_1) \dots \star_x^\psi(\tau_n) \in \mathcal{D}$, we have $\tau_1 = \dots = \tau_n$.

(3) For all $\star_x^{\varkappa_1}(\tau_1) \in \mathcal{D}$ and $\star_y^{\varkappa_2}(\tau_2) \in \mathcal{D}$, $\varkappa_1 \neq \varkappa_2$ implies $x \neq y$.

Declarative Type Rules Fig. 4 presents a declarative definition of the type system of \mathbb{B} . In this type system, copy compatibility is realized through *copy coercion* (\leq) rules that are similar to subtyping rules (S-* rules in Fig. 4). Since reference types $\uparrow\tau$ are handled only by S-Refl, types cannot be coerced beyond a reference boundary. Also, two function types are coercible only if they are structurally identical. Here, the contravariance/covariance of argument/return types is unnecessary as we can follow a standard convention with respect to the mutability of argument/return types at copy positions. The rules for typing expressions (T-* rules) introduce these coercions at all copy-compatible positions.

The type judgment $\mathcal{D}; \Gamma; \Sigma \vdash e : \tau$ is understood as: given a binding environment Γ and store typing Σ , the expression e has type τ subject to the set of \star constraints \mathcal{D} . We write $e \leq : \tau$ as a shorthand for $e : \tau'$ and $\tau' \leq : \tau$, for some type τ' . The rule T-Lambda permits the interface type of a function to be different from its internal type, as explained in Sec. 2.1. The rule T-App introduces copy-coercions at argument and return positions of an application. T-Let-M rule types let expressions monomorphically, and thus requires a let^ψ annotation. In this case, the expression e_1 is permitted to be expansive (i.e. need not be a syntactic value v). The T-Let-MP rule types let expressions where the expression being bound is a syntactic value. It assigns x a constrained type scheme along with the constraint $\star_x^{\varkappa}(\tau)$. The T-Id rule instantiates types and constraints. The instantiated constraints are collected over the entire derivation, so that we can enforce instantiation consistency. $\stackrel{\neq}{=} \bar{\alpha}$ identifies fresh type variables.

We prove the soundness of our type system by demonstrating subject reduction. Here, we prove that the type of an expression is preserved exactly by left-execution, which ensures that the type of a location does not change during the execution of a program. We also show that right execution preserves types except for shallow mutability. The result of a right execution can only be used in copy compatible positions, or as the target of a dereference. In the former case, preservation of shallow mutability is unnecessary, and in the later, the type within the reference is preserved exactly.

The interesting case is the safety of polymorphic let expressions. The T-Let-MP rule does not require that the type τ being quantified over be immutable, but adds the $\star_x^{\varkappa}(\tau)$ constraint. Now, if we have a derivation $\mathcal{D}; \Gamma; \Sigma \vdash e : \tau$ such that $\models\mathcal{D}$, then one of the two cases must follow. (1) If any instantiation of τ is mutable, then $\varkappa = \psi$. In this case, execution proceeds through the E-Let-M rule, which create a stack location for x . Therefore, x is permitted to

$$\begin{array}{c}
\begin{array}{c} \text{S-Refl} \\ \hline \tau \triangleq: \tau \end{array} \quad \begin{array}{c} \text{S-Trans} \\ \hline \frac{\tau_0 \triangleq: \tau_1 \quad \tau_1 \triangleq: \tau_2}{\tau_0 \triangleq: \tau_2} \end{array} \quad \begin{array}{c} \text{S-Mut} \\ \hline \frac{\rho \triangleq: \rho'}{\Psi \rho \triangleq: \Psi \rho'} \end{array} \quad \begin{array}{c} \text{S-Pair} \\ \hline \frac{\tau_1 \triangleq: \tau'_1 \quad \tau_2 \triangleq: \tau'_2}{\tau_1 \times \tau_2 \triangleq: \tau'_1 \times \tau'_2} \end{array} \\
\begin{array}{c} \text{S-Mt1} \\ \hline \frac{\nabla(\rho) = \tau}{\alpha \downarrow \rho \triangleq: \tau} \end{array} \quad \begin{array}{c} \text{S-Mt2} \\ \hline \frac{\blacktriangle(\rho) = \tau}{\tau \triangleq: \alpha \downarrow \rho} \end{array} \quad \begin{array}{c} \text{S-Mf1} \\ \hline \frac{\nabla(\rho) = \tau}{\alpha \downarrow \rho \triangleq: \tau} \end{array} \quad \begin{array}{c} \text{S-Mf2} \\ \hline \frac{\alpha \downarrow \rho \triangleq: \rho'}{\Psi \alpha \downarrow \rho \triangleq: \Psi \rho'} \end{array} \quad \begin{array}{c} \text{S-Mf3} \\ \hline \frac{\triangle(\rho) = \tau}{\tau \triangleq: \varsigma \downarrow \rho} \end{array} \\
\begin{array}{c} \text{T-Unit} \\ \hline \emptyset; \Gamma; \Sigma \vdash () : \text{unit} \end{array} \quad \begin{array}{c} \text{T-Bool} \\ \hline \emptyset; \Gamma; \Sigma \vdash \text{b} : \text{bool} \end{array} \quad \begin{array}{c} \text{T-Id} \\ \hline \frac{\Gamma(x) = \forall \bar{\alpha}. \tau \setminus \mathcal{D} \quad \theta \Vdash \{ \tau, \mathcal{D} \} \quad \text{dom}(\theta) = \{ \bar{\alpha} \}}{\theta \langle \mathcal{D} \rangle; \Gamma; \Sigma \vdash x : \theta \langle \tau \rangle} \end{array} \\
\begin{array}{c} \text{T-Hloc} \\ \hline \frac{\Sigma(\ell) = \tau}{\emptyset; \Gamma; \Sigma \vdash \ell : \uparrow \tau} \end{array} \quad \begin{array}{c} \text{T-Sloc} \\ \hline \frac{\Sigma(l) = \tau}{\emptyset; \Gamma; \Sigma \vdash l : \tau} \end{array} \quad \begin{array}{c} \text{T-Lambda} \\ \hline \frac{\mathcal{D}; \Gamma, x \mapsto \tau_1; \Sigma \vdash e : \tau_2 \quad \tau_1 \stackrel{\nabla}{=} \tau'_1 \quad \tau_2 \stackrel{\nabla}{=} \tau'_2}{\mathcal{D}; \Gamma; \Sigma \vdash \lambda x. e : \tau'_1 \rightarrow \tau'_2} \end{array} \\
\begin{array}{c} \text{T-App} \\ \hline \frac{\mathcal{D}_1; \Gamma; \Sigma \vdash e_1 \triangleq: \tau_a \rightarrow \tau_r \quad \mathcal{D}_2; \Gamma; \Sigma \vdash e_2 \triangleq: \nabla(\tau_a) \quad \triangle(\tau_r) \triangleq: \tau}{\mathcal{D}_1 \cup \mathcal{D}_2; \Gamma; \Sigma \vdash e_1 e_2 : \tau} \end{array} \\
\begin{array}{c} \text{T-If} \\ \hline \frac{\mathcal{D}_1; \Gamma; \Sigma \vdash e_1 \triangleq: \text{bool} \quad \mathcal{D}_2; \Gamma; \Sigma \vdash e_2 \triangleq: \tau \quad \mathcal{D}_3; \Gamma; \Sigma \vdash e_3 \triangleq: \tau \quad \tau' \triangleq: \tau}{\mathcal{D}_1 \cup \mathcal{D}_2 \cup \mathcal{D}_3; \Gamma; \Sigma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau'} \end{array} \\
\begin{array}{c} \text{T-Pair} \\ \hline \frac{\mathcal{D}_1; \Gamma; \Sigma \vdash e_1 \triangleq: \tau_1 \quad \mathcal{D}_2; \Gamma; \Sigma \vdash e_2 \triangleq: \tau_2 \quad \tau'_1 \triangleq: \tau_1 \quad \tau'_2 \triangleq: \tau_2}{\mathcal{D}_1 \cup \mathcal{D}_2; \Gamma; \Sigma \vdash (e_1, e_2) : \tau'_1 \times \tau'_2} \end{array} \quad \begin{array}{c} \text{T-Sel} \\ \hline \frac{\mathcal{D}; \Gamma; \Sigma \vdash e : \tau \quad \tau \stackrel{\nabla}{=} \tau_1 \times \tau_2}{\mathcal{D}; \Gamma; \Sigma \vdash e.i : \tau_i} \end{array} \\
\begin{array}{c} \text{T-Set} \\ \hline \frac{\mathcal{D}_1; \Gamma; \Sigma \vdash l \triangleq: \Psi \rho \quad \mathcal{D}_2; \Gamma; \Sigma \vdash e \triangleq: \rho}{\mathcal{D}_1 \cup \mathcal{D}_2; \Gamma; \Sigma \vdash l := e : \text{unit}} \end{array} \quad \begin{array}{c} \text{T-Dup} \\ \hline \frac{\mathcal{D}; \Gamma; \Sigma \vdash e \triangleq: \tau \quad \tau' \triangleq: \tau}{\mathcal{D}; \Gamma; \Sigma \vdash \text{dup}(e) : \uparrow \tau'} \end{array} \quad \begin{array}{c} \text{T-Deref} \\ \hline \frac{\mathcal{D}; \Gamma; \Sigma \vdash e \triangleq: \uparrow \tau}{\mathcal{D}; \Gamma; \Sigma \vdash e^\wedge : \tau} \end{array} \\
\begin{array}{c} \text{T-Let-M} \\ \hline \frac{\mathcal{D}_1; \Gamma; \Sigma \vdash e_1 \triangleq: \tau_1 \quad \tau \triangleq: \tau_1 \quad \mathcal{D}_2; \Gamma, x \mapsto \tau; \Sigma \vdash e_2 : \tau_2}{\mathcal{D}_1 \cup \mathcal{D}_2; \Gamma; \Sigma \vdash (\text{let}^{\psi} x = e_1 \text{ in } e_2) : \tau_2} \end{array} \\
\begin{array}{c} \text{T-Let-MP} \\ \hline \frac{\mathcal{D}_1; \Gamma; \Sigma \vdash v \triangleq: \tau_1 \quad \tau \triangleq: \tau_1 \quad \mathcal{D} = \mathcal{D}_1 \cup \{ \star_x^{\varkappa}(\tau) \} \quad \{ \bar{\alpha} \} = \text{ftv}(\tau, \mathcal{D}) \setminus \text{ftv}(\Gamma, \Sigma)}{\mathcal{D}_2; \Gamma, x \mapsto \forall \bar{\alpha}. \tau \setminus \mathcal{D}; \Sigma \vdash e : \tau_2 \quad \Vdash_{\bar{\alpha}ew} \bar{\beta}}}{\mathcal{D}[\bar{\beta}/\bar{\alpha}] \cup \mathcal{D}_2; \Gamma; \Sigma \vdash (\text{let}^{\varkappa} x = v \text{ in } e) : \tau_2} \end{array}
\end{array}$$

Figure 4: Declarative Type Rules

be the target of an assignment. $\models \mathcal{D}$ guarantees that all instantiations of τ are identical, which ensures that the type of a location cannot change. (2) If τ is instantiated polymorphically, then $\varkappa = \forall$. Execution proceeds through the E-Let-P rule, which performs a value substitution. Here, $\models \mathcal{D}$ guarantees that all instantiations are deeply immutable. Therefore, x cannot be directly used (in the forms x or $x.p$) as the target of an assignment, which ensures that the value substitution cannot lead to a stuck state.

Definition V (Consistent Type Derivation). Let $\{\{\mathcal{D}; \Gamma; \Sigma \vdash e : \tau\}\}$ denote the extension $\{\}\{\}\{\}$ function to the set of all types used in the derivation of $\mathcal{D}; \Gamma; \Sigma \vdash e : \tau$. We say that $\mathcal{D}; \Gamma; \Sigma \vdash e : \tau$ is a consistent derivation if $\mathcal{D}'; \Gamma; \Sigma \vdash e : \tau$ for some $\mathcal{D}' \subseteq \mathcal{D}$, and $\Vdash \{\{\mathcal{D}\}\} \cup \{\{\mathcal{D}'; \Gamma; \Sigma \vdash e : \tau\}\}$.

Definition VI (Stack and Heap Typing) A heap H and a stack S are said to be well typed with respect to Γ, Σ and \mathcal{D} , written $\mathcal{D}; \Gamma; \Sigma \vdash_* H + S$, if:

- (1) $\text{dom}(\Sigma) = \text{dom}(H) \cup \text{dom}(S)$
- (2) $\forall \ell \in \text{dom}(H), \mathcal{D}; \Gamma; \Sigma \vdash_* H(\ell) : \tau$ such that $\Sigma(\ell) \stackrel{\nabla}{=} \tau$
- (3) $\forall l \in \text{dom}(S), \mathcal{D}; \Gamma; \Sigma \vdash_* S(l) : \tau$ such that $\Sigma(l) \stackrel{\nabla}{=} \tau$

Definition VII (Valid Lvalues). We say that an lvalue \mathcal{L} is valid with respect to a stack S and heap H , written $H + S \vdash_v \mathcal{L}$ if for some p , either (1) $\mathcal{L} = l$ or $\mathcal{L} = l.p$ where $l \in \text{dom}(S)$; or (2) $\mathcal{L} = l^\wedge$ or $\mathcal{L} = l^\wedge.p$ where $l \in \text{dom}(H)$.

Lemma I (Progress). If e is a closed canonical well typed expression, that is, $\mathcal{D}; \emptyset; \Sigma \vdash_* e : \tau$ for some τ and Σ , given any heap and stack such that $\mathcal{D}; \emptyset; \Sigma \vdash_* H + S$,

- (1) If e is a left expression ($e = l$), then e is either a valid lvalue (that is, $e = \mathcal{L}$ and $H + S \vdash_v \mathcal{L}$) or else $\exists e', S', H'$ such that $S; H; e \Rightarrow S'; H'; e'$.
- (2) e is a value v or else $\exists e', S', H'$ such that $S; H; e \Rightarrow S'; H'; e'$.

Lemma II (Preservation). For any canonical expression e , if $\mathcal{D}; \Gamma; \Sigma \vdash_* e : \tau$, $\mathcal{D}; \Gamma; \Sigma \vdash_* H + S$ and $\models \mathcal{D}$ then,

- (1) If $S; H; e \Rightarrow S'; H'; e'$, then, $\exists \Sigma' \supseteq \Sigma$ such that $\mathcal{D}; \Gamma; \Sigma' \vdash_* e' : \tau$

I-Unit $\frac{}{\Gamma; \Sigma \vdash_7 0 : \text{unit} \mid \emptyset}$	I-Bool $\frac{}{\Gamma; \Sigma \vdash_7 b : \text{bool} \mid \emptyset}$	I-Id $\frac{\Gamma(x) = \forall \bar{\alpha}. \tau \setminus \mathcal{D} \quad \theta = [\alpha \mapsto \beta] \quad \frac{}{\Gamma; \Sigma \vdash_7 x : \theta(\tau) \mid \theta(\mathcal{D})}}{\Gamma; \Sigma \vdash_7 x : \theta(\tau) \mid \theta(\mathcal{D})}$
I-Hloc $\frac{\Sigma(\ell) = \tau}{\Gamma; \Sigma \vdash_7 \ell : \uparrow \tau \mid \emptyset}$	I-Sloc $\frac{\Sigma(l) = \tau}{\Gamma; \Sigma \vdash_7 l : \tau \mid \emptyset}$	I-Lambda $\frac{\Gamma, x \mapsto \beta \downarrow \alpha; \Sigma \vdash_7 e : \tau \mid \mathcal{C} \quad \frac{}{\Gamma; \Sigma \vdash_7 \lambda x. e : \beta' \downarrow \alpha \rightarrow \gamma' \downarrow \delta \mid \mathcal{C} \cup \{\tau = \gamma \downarrow \delta\}}}{\Gamma; \Sigma \vdash_7 \lambda x. e : \beta' \downarrow \alpha \rightarrow \gamma' \downarrow \delta \mid \mathcal{C} \cup \{\tau = \gamma \downarrow \delta\}}$
I-App $\frac{\Gamma; \Sigma \vdash_7 e_1 : \tau_1 \mid \mathcal{C}_1 \quad \Gamma; \Sigma \vdash_7 e_2 : \tau_2 \mid \mathcal{C}_2 \quad \frac{}{\Gamma; \Sigma \vdash_7 e_1 e_2 : \varepsilon \downarrow \gamma \mid \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 = \alpha \downarrow (\beta' \downarrow \beta \rightarrow \gamma' \downarrow \gamma), \tau_2 = \delta \downarrow \beta\}}}{\Gamma; \Sigma \vdash_7 e_1 e_2 : \varepsilon \downarrow \gamma \mid \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 = \alpha \downarrow (\beta' \downarrow \beta \rightarrow \gamma' \downarrow \gamma), \tau_2 = \delta \downarrow \beta\}}$		
I-If $\frac{\Gamma; \Sigma \vdash_7 e_1 : \tau_1 \mid \mathcal{C}_1 \quad \Gamma; \Sigma \vdash_7 e_2 : \tau_2 \mid \mathcal{C}_2 \quad \Gamma; \Sigma \vdash_7 e_3 : \tau_3 \mid \mathcal{C}_3 \quad \frac{}{\Gamma; \Sigma \vdash_7 \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \varepsilon \downarrow \gamma \mid \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\tau_1 = \alpha \downarrow \text{bool}, \tau_2 = \beta \downarrow \gamma, \tau_3 = \delta \downarrow \gamma\}}}{\Gamma; \Sigma \vdash_7 \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \varepsilon \downarrow \gamma \mid \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\tau_1 = \alpha \downarrow \text{bool}, \tau_2 = \beta \downarrow \gamma, \tau_3 = \delta \downarrow \gamma\}}$		
I-Set $\frac{\Gamma; \Sigma \vdash_7 l : \tau_1 \mid \mathcal{C}_1 \quad \Gamma; \Sigma \vdash_7 e : \tau_2 \mid \mathcal{C}_2 \quad \frac{}{\Gamma; \Sigma \vdash_7 l := e : \text{unit} \mid \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 = (\Psi \alpha) \downarrow \beta, \tau_2 = \gamma \downarrow \beta\}}}{\Gamma; \Sigma \vdash_7 l := e : \text{unit} \mid \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 = (\Psi \alpha) \downarrow \beta, \tau_2 = \gamma \downarrow \beta\}}$	I-Deref $\frac{\Gamma; \Sigma \vdash_7 e : \tau \mid \mathcal{C} \quad \frac{}{\Gamma; \Sigma \vdash_7 e^\wedge : \alpha \mid \mathcal{C} \cup \{\tau = \beta \downarrow \uparrow \alpha\}}}{\Gamma; \Sigma \vdash_7 e^\wedge : \alpha \mid \mathcal{C} \cup \{\tau = \beta \downarrow \uparrow \alpha\}}$	
I-Dup $\frac{\Gamma; \Sigma \vdash_7 e : \tau \mid \mathcal{C} \quad \frac{}{\Gamma; \Sigma \vdash_7 \text{dup}(e) : \uparrow(\alpha \downarrow \beta) \mid \mathcal{C} \cup \{\tau = \gamma \downarrow \beta\}}}{\Gamma; \Sigma \vdash_7 \text{dup}(e) : \uparrow(\alpha \downarrow \beta) \mid \mathcal{C} \cup \{\tau = \gamma \downarrow \beta\}}$	I-Sel $\frac{\Gamma; \Sigma \vdash_7 e : \tau \mid \mathcal{C} \quad \tau_1 = \alpha \downarrow \beta \quad \tau_2 = \gamma \downarrow \delta \quad \frac{}{\Gamma; \Sigma \vdash_7 e.i : \tau_i \mid \mathcal{C} \cup \{\tau = \varepsilon \downarrow (\tau_1 \times \tau_2)\}}}{\Gamma; \Sigma \vdash_7 e.i : \tau_i \mid \mathcal{C} \cup \{\tau = \varepsilon \downarrow (\tau_1 \times \tau_2)\}}$	
I-Pair $\frac{\Gamma; \Sigma \vdash_7 e_1 : \tau_1 \mid \mathcal{C}_1 \quad \Gamma; \Sigma \vdash_7 e_2 : \tau_2 \mid \mathcal{C}_2 \quad \frac{}{\Gamma; \Sigma \vdash_7 (e_1, e_2) : \alpha \downarrow \gamma \times \beta \downarrow \delta \mid \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 = \alpha' \downarrow \gamma, \tau_2 = \beta' \downarrow \delta\}}}{\Gamma; \Sigma \vdash_7 (e_1, e_2) : \alpha \downarrow \gamma \times \beta \downarrow \delta \mid \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 = \alpha' \downarrow \gamma, \tau_2 = \beta' \downarrow \delta\}}$		
I-Let-Exp $\frac{\Gamma; \Sigma \vdash_7 e_1 : \tau_1 \mid \mathcal{C}_1 \quad e_1 \neq v \quad \Gamma, x \mapsto \alpha \downarrow \beta; \Sigma \vdash_7 e_2 : \tau_2 \mid \mathcal{C}_2 \quad \frac{}{\Gamma; \Sigma \vdash_7 \text{let}^x x = e_1 \text{ in } e_2 : \tau_2 \mid \mathcal{C}_1 \cup \{\tau_1 = \gamma \downarrow \beta, \kappa = \psi\} \cup \mathcal{C}_2}}{\Gamma; \Sigma \vdash_7 \text{let}^x x = e_1 \text{ in } e_2 : \tau_2 \mid \mathcal{C}_1 \cup \{\tau_1 = \gamma \downarrow \beta, \kappa = \psi\} \cup \mathcal{C}_2}$		
I-Let-Val $\frac{\Gamma; \Sigma \vdash_7 v : \tau_1 \mid \mathcal{C}_1 \quad \mathcal{C}'_1 = \mathcal{C}_1 \cup \{\tau_1 = \gamma \downarrow \beta\} \quad \mathcal{U}(\mathcal{C}'_1) = (\mathcal{D}', \theta) \quad \mathcal{D} = \mathcal{D}' \cup \{\star_x^e(\tau)\} \quad \tau = \theta(\delta \downarrow \beta) \quad \{\bar{\alpha}\} = \text{ftv}(\tau, \mathcal{D}) \setminus \text{ftv}(\theta(\Gamma), \theta(\Sigma)) \quad \Gamma, x \mapsto \forall \bar{\alpha}. \tau \setminus \mathcal{D}; \Sigma \vdash_7 e : \tau_2 \mid \mathcal{C}_2 \quad \frac{}{\Gamma; \Sigma \vdash_7 \text{let}^x x = v \text{ in } e : \tau_2 \mid \mathcal{C}'_1[\bar{\alpha}]/\bar{\alpha} \cup \mathcal{C}_2}}{\Gamma; \Sigma \vdash_7 \text{let}^x x = v \text{ in } e : \tau_2 \mid \mathcal{C}'_1[\bar{\alpha}]/\bar{\alpha} \cup \mathcal{C}_2}$		

Figure 5: Type Inference Algorithm

- and $\mathcal{D}; \Gamma; \Sigma' \vdash_* H' + S'$.
- (2) If $\mathcal{S}; H; e \Rightarrow S'; H'; e'$, then, $\exists \Sigma' \supseteq \Sigma$ such that $\mathcal{D}; \Gamma; \Sigma' \vdash_* e' : \tau'$,
 $\mathcal{D}; \Gamma; \Sigma' \vdash_* H' + S'$ and $\tau \stackrel{\nabla}{=} \tau'$.

Definition VIII (Stuck State). A system state $\mathcal{S}; H; e$ is said to be stuck if $e \neq v$ and there are no S', H' , and e' such that $\mathcal{S}; H; e \Rightarrow S'; H'; e'$.

Theorem I (Type Soundness). Let $\stackrel{*}{\Rightarrow}$ denote the reflexive-transitive-closure of \Rightarrow . For any canonical expression e , if $\mathcal{D}; \emptyset; \Sigma \vdash_* e : \tau$, $\mathcal{D}; \emptyset; \Sigma \vdash_* H + S$, $\models \mathcal{D}$, and $\mathcal{S}; H; e \stackrel{*}{\Rightarrow} S'; H'; e'$, then $S'; H'; e'$ is not stuck. That is, execution of a closed, canonical, well typed expression cannot lead to a stuck state.

Type Inference Algorithm Type inference is a program transformation that accepts a program in which `let` expressions are not annotated with their kinds, and returns the same program with `let` expressions annotated with their kinds and all expressions annotated with their types. The type inference algorithm is shown in Fig. 5. The inference judgment $\Gamma; \Sigma \vdash_7 e : \tau \mid \mathcal{C}$ is understood as: given a binding environment Γ and store typing Σ , the expression e has type τ subject to the constraints \mathcal{C} .

The inference algorithm introduces constrained types of the form $\varsigma \downarrow \rho$ at all copy compatible positions. For example, the I-App rule introduces copy compatibility for the function type itself, the argument and the return types. The I-Sel rule represents the pair type as $\varepsilon \downarrow (\alpha \downarrow \beta \times \gamma \downarrow \delta)$, which (1) permits top-level mutability of the pair type to be either mutable or immutable (2) ensures that the type of the selection is exactly same as the type of the field being selected (3) propagates full copy compatibility “one level down.”

The unification algorithm is shown in Fig. 6. The unification of a constraint set \mathcal{C} either fails with an error \perp , or produces the pair (\mathcal{D}, θ) . θ is a solution for all equality constraints and some of the \star constraints in \mathcal{C} . \mathcal{D} is the set of \star constraints in \mathcal{C} on which θ has been applied. \cup represents disjoint union of sets.

The U-Ct* rules perform unification of constrained types with other constrained or unconstrained types. First, immutable versions of the two types are unified to establish compatibility (through constraints involving ∇ and \equiv). Then, the constrained type is made to exactly equal the other type by unifying its variable part with the other type. The key observation here is that the copy compatibility is a special restricted form of subtyping. Since the type of the

U-Empty	$\mathcal{U}(\emptyset)$	$=$	$(\emptyset, \langle \rangle)$
U-Refl	$\mathcal{U}(\{\tau = \tau\} \cup \mathcal{C})$	$=$	$\mathcal{U}(\mathcal{C})$
U-Sym	$\mathcal{U}(\{\tau_1 = \tau_2\} \cup \mathcal{C})$	$=$	$\mathcal{U}(\{\tau_2 = \tau_1\} \cup \mathcal{C})$
U-Var	$\mathcal{U}(\{\alpha = \tau\} \cup \mathcal{C}) \mid \alpha \notin \tau$	$=$	$(\mathcal{D}, \theta_\alpha \circ \theta_u)$ where $\theta_\alpha = [\alpha \mapsto \tau]$ and $\mathcal{U}(\theta_\alpha(\mathcal{C})) = (\mathcal{D}, \theta_u)$
U-Fn	$\mathcal{U}(\{\tau_\alpha \rightarrow \tau_r = \tau'_\alpha \rightarrow \tau'_r\} \cup \mathcal{C})$	$=$	$\mathcal{U}(\mathcal{C} \cup \{\tau_\alpha = \tau'_\alpha, \tau_r = \tau'_r\})$
U-Ref	$\mathcal{U}(\{\uparrow\tau_1 = \uparrow\tau_2\} \cup \mathcal{C})$	$=$	$\mathcal{U}(\mathcal{C} \cup \{\tau_1 = \tau_2\})$
U-Mut	$\mathcal{U}(\{\Psi\rho_1 = \Psi\rho_2\} \cup \mathcal{C})$	$=$	$\mathcal{U}(\mathcal{C} \cup \{\rho_1 = \rho_2\})$
U-Pair	$\mathcal{U}(\{\tau_1 \times \tau_2 = \tau'_1 \times \tau'_2\} \cup \mathcal{C})$	$=$	$\mathcal{U}(\mathcal{C} \cup \{\tau_1 = \tau'_1, \tau_2 = \tau'_2\})$
U-Ct1	$\mathcal{U}(\{\alpha \downarrow \rho_1 = \beta \downarrow \rho_2\} \cup \mathcal{C})$	$=$	$\mathcal{U}(\mathcal{C} \cup \{\rho_1 \stackrel{\nabla}{=} \rho_2, \alpha = \beta\})$
U-Ct2	$\mathcal{U}(\{\alpha \downarrow \rho = \rho'\} \cup \mathcal{C})$	$=$	$\mathcal{U}(\mathcal{C} \cup \{\rho \stackrel{\nabla}{=} \rho', \alpha = \rho'\})$
U-Ct3	$\mathcal{U}(\{\varsigma_1 \downarrow \rho_1 = \varsigma_2 \downarrow \rho_2\} \cup \mathcal{C})$	$=$	$\mathcal{U}(\mathcal{C} \cup \{\rho_1 \stackrel{\nabla}{=} \rho_2, \varsigma_1 = \varsigma_2\})$
U-Ct4	$\mathcal{U}(\{\varsigma \downarrow \rho = \varrho\} \cup \mathcal{C})$	$=$	$\mathcal{U}(\mathcal{C} \cup \{\rho \stackrel{\nabla}{=} \varrho, \varsigma = \varrho\})$
U-K	$\mathcal{U}(\{\kappa = \varkappa\} \cup \mathcal{C})$	$=$	$(\mathcal{D}, \theta_k \circ \theta_u)$ where $\theta_k = [\kappa \mapsto \varkappa]$ and $\mathcal{U}(\theta_k(\mathcal{C})) = (\mathcal{D}, \theta_u)$
U-Om1	$\mathcal{U}(\{\star_x^\psi(\tau_1), \star_x^\psi(\tau_2)\} \cup \mathcal{C})$	$=$	$(\mathcal{D} \cup \theta\{\star_x^\psi(\tau_1), \star_x^\psi(\tau_2)\}, \theta)$ where $\mathcal{U}(\mathcal{C} \cup \{\tau_1 = \tau_2\}) = (\mathcal{D}, \theta)$
U-Op1	$\mathcal{U}(\{\star_x^\forall(\tau)\} \cup \mathcal{C}) \mid \Box(\tau)$	$=$	$(\mathcal{D} \cup \theta\{\star_x^\forall(\tau)\}, \theta)$ where $\mathcal{U}(\mathcal{C} \cup \{\tau = \exists(\tau)\}) = (\mathcal{D}, \theta)$
U-Om2	$\mathcal{U}(\{\star_x^\kappa(\tau)\} \cup \mathcal{C}) \mid \text{Mut}(\tau)$	$=$	$(\mathcal{D}, \theta_k \circ \theta_u)$ where $\theta_k = [\kappa \mapsto \psi]$ and $\mathcal{U}(\theta_k(\{\star_x^\kappa(\tau)\} \cup \mathcal{C})) = (\mathcal{D}, \theta_u)$
U-Op2	$\mathcal{U}(\{\star_x^\kappa(\tau_1), \star_x^\kappa(\tau_2)\} \cup \mathcal{C})$ where $\mathcal{U}(\{\tau_1 = \tau_2\} \cup \mathcal{C}) = \perp$	$=$	$(\mathcal{D}, \theta_k \circ \theta_u)$ where $\theta_k = [\kappa \mapsto \forall]$ and $\mathcal{U}(\theta_k(\{\star_x^\kappa(\tau_1), \star_x^\kappa(\tau_2)\} \cup \mathcal{C})) = (\mathcal{D}, \theta_u)$
U-Error	$\mathcal{U}(c \cup \mathcal{C}) \mid c \notin \mathcal{C}_v \cup \mathcal{C}_s \cup \mathcal{C}_p$	$=$	\perp

$\mathcal{C}_v = \forall \alpha, \varsigma, \rho, \tau, \tau' \mid \alpha \notin \tau'. \{\tau = \tau, \alpha = \tau', \tau' = \alpha, \alpha \downarrow \rho = \tau, \tau = \alpha \downarrow \rho, \varsigma \downarrow \rho = \tau, \tau = \varsigma \downarrow \rho\}$
 $\mathcal{C}_s = \forall \rho, \rho', \tau, \tau', \tau_1, \tau'_1. \{\tau \rightarrow \tau_1 = \tau' \rightarrow \tau'_1, \uparrow\tau = \uparrow\tau', \Psi\rho = \Psi\rho'\}$
 $\mathcal{C}_p = \forall x, \kappa, \varkappa, \tau, \tau' \mid \neg \text{Mut}(\tau'). \{\kappa = \varkappa, \star_x^\psi(\tau), \star_x^\forall(\tau'), \star_x^\kappa(\tau)\}$

Figure 6: Unification Algorithm

copy can be anywhere in the lattice of copy compatible types, subtyping requirements are always with respect a local maxima (the most immutable compatible type). We exploit this behavior to design a simple unification algorithm that only uses equality constraints over constrained types.

The U-Om1 ensures that all instantiations of monomorphic kind are the same. U-Op1 rule forces any concretizable instantiation of polymorphic kind to be immutable. The U-Om2 rule infers monomorphic kind based on the mutability of the instantiated type, and U-Op2 infers polymorphic kind if a variable x is instantiated polymorphically to two types that do not inter-unify.

Definition IX (Constraint Satisfaction). *The satisfaction of a constraint set \mathcal{C} by a substitution θ is defined as follows.*

$$\frac{\forall (\tau_1 = \tau_2) \in \mathcal{C}, \theta\langle\tau_1\rangle = \theta\langle\tau_2\rangle \quad \forall (\kappa = \varkappa) \in \mathcal{C}, \theta\langle\kappa\rangle = \theta\langle\varkappa\rangle}{\theta \vdash_{\text{sol}} \mathcal{C} \rightsquigarrow \mathcal{D}} \quad \frac{\theta \vdash_{\text{sol}} \mathcal{C} \rightsquigarrow \mathcal{D}}{\models \mathcal{D}}}{\theta \vdash_{\text{sat}} \mathcal{C} \rightsquigarrow \mathcal{D}}$$

Definition X (Notational Derivations). *We write:*

- (1) $\theta; \Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{D}$ if $\Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{C}, \theta \vdash_{\text{sol}} \mathcal{C} \rightsquigarrow \mathcal{D}$, and $\theta \Vdash \{\Gamma, \Sigma, \tau, \mathcal{C}\}$
- (2) $\theta; \mathcal{D}; \Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau$ if $\theta\langle\mathcal{D}\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash_{\bar{\tau}} \theta\langle e \rangle : \theta\langle\tau\rangle$

Lemma III (Correctness of Unification). *If $\mathcal{U}(\mathcal{C}) = (\mathcal{D}, \theta)$, then $\theta \vdash_{\text{sol}} \mathcal{C} \rightsquigarrow \mathcal{D}$*

Lemma IV (Satisfiability of Unified Constraints). *If $\mathcal{U}(\mathcal{C}) = (\mathcal{D}, \theta_u)$, then there exists a substitution θ_s such that $\theta_u \circ \theta_s \vdash_{\text{sat}} \mathcal{C} \rightsquigarrow \mathcal{D}$.*

Lemma V (Principality of Unification). *If $\mathcal{U}(\mathcal{C}) = (\mathcal{D}, \theta_u)$, where \mathcal{C} is a set of constraints obtained from the type inference algorithm, then, for all θ_s such that $\theta_s \vdash_{\text{sol}} \mathcal{C} \rightsquigarrow \mathcal{D}'$, we have $\theta_s \supseteq \theta_u$.*

Lemma VI (Decidability of Unification). *The problem of computing a canonical derivation of $\mathcal{U}(\mathcal{C})$ for an arbitrary \mathcal{C} , where no two applications of U-Sym rule happen consecutively is decidable.*

Theorem II (Soundness of Type Inference). *If $\theta; \Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{D}$, then $\theta; \mathcal{D}; \Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau$.*

Lemma VII (Type Checkability). *If $\Gamma; \Sigma \vdash e : \tau \mid \mathcal{C}$ and $\mathcal{U}(\mathcal{C}) = (\mathcal{D}, \theta)$, then $\exists \theta'$ such that $\models \theta' \langle \mathcal{D} \rangle$ and $\theta \circ \theta' \langle e \rangle$ is canonical, and $\theta \circ \theta'; \mathcal{D}; \Gamma; \Sigma \vdash e : \tau$.*

Theorem III (Completeness of Type Inference). *If $\theta; \mathcal{D}; \Gamma; \Sigma \vdash_* e : \tau$, then there exists a $\theta' \supseteq \theta$ such that $\theta'; \Gamma; \Sigma \vdash e : \tau \mid \mathcal{D}$.*

4 Related Work

Grossman [6] provides a theory of using quantified types with imperative C style mutation for Cyclone. However, his formalization requires explicit annotation for all polymorphic definitions and instantiations. In contrast, we believe that the best way to integrate polymorphism into the systems programming paradigm is by automatic inference. A further contribution of our work (in comparison to [6]) is that we give a formal specification and proof of correctness of the inference algorithm, not just the type system. Cyclone [10] uses region analysis to provide safe support for the address & operator. This technique is complementary to our work, and can be used to incorporate & operator in \mathbb{B} .

C’s `const` notion of immutability-by-alias offers localized checking of immutability properties, and encourages good programming practice by serving as documentation of programmers’ intentions. Other systems have proposed immutability-by-name [2], referential immutability [19, 24] (transitive immutability-by-reference), *etc.* These techniques are orthogonal and complementary to the immutability-by-location property in \mathbb{B} . For example, we could have types like `(const $\Psi\tau$)` that can express both global and local usage properties of a location.

A monadic model [13] of mutability is used in pure functional languages like Haskell [14]. In this model, the type system distinguishes side-effecting computations from pure ones (and not just mutable locations from immutable ones). Even though this model is beneficial for integration with verification systems, it is considerably removed from the idioms needed by systems programmers. For example, Hughes argues that there is no satisfactory way of creating and using global mutable variables using monads [7]. There have been proposals for adding unboxed representation control to Haskell [12, 8]. However, these systems are pure and therefore do not consider the effects of mutability.

Cqual [5] provides a framework of type qualifiers, which can be used to infer maximal `const` qualifications for C programs. However, CQual does not deal with polymorphism of types. In a monomorphic language, we can infer types and qualifiers independently. Adding polymorphism to CQual would introduce substantial challenges, particularly if polymorphism should be automatically inferred. The inference of types and qualifiers (mutability) becomes co-dependent: we need base types to infer qualifiers; but, we also need the qualifiers to infer base types due to the value restriction. \mathbb{B} supports a polymorphic language and performs simultaneous inference of base types and mutability.

5 Conclusions

In this paper, we have defined a language and type system for systems programming which integrates all of unboxed representation, consistent complete mutability support, and polymorphism. The mutability model is expressive enough to permit mutation of unboxed/stack locations, and at the same time guarantees that types are definitive about the mutability of every location across all aliases.

Complete support for mutability introduces challenges for type inference at copy boundaries. We have developed a novel algorithm that infers principal types using a system of constrained types. To our knowledge, this is the first sound and complete algorithm that infers both mutability and polymorphism in a systems programming language with copy compatibility.

The type inference algorithm is implemented as part of the BitC [20] language compiler. The core of the compiler involves 22,433 lines of C++ code, of which implementation of the type system accounts for about 7,816 lines. The source code can be obtained from <http://bitc-lang.org>.

References

- [1] E. Biagioni, R. Harper, and P. Lee “A network protocol stack in Standard ML” *Higher Order and Symbolic Computation, Vol.14, No.4*, 2001.
- [2] R. Deline and M. Fähndrich, “VAULT: a programming language for reliable systems” <http://research.microsoft.com/vault>, 2001
- [3] H. Derby, “The performance of FoxNet 2.0” *Technical Report CMU-CS-99-137* School of Computer Science, Carnegie Mellon University, June 1999.
- [4] ECMA International “Standard ECMA-334 C# Language Specification” <http://www.ecma-international.org/publications/standards/Ecma-334.htm>
- [5] J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken “Flow-Insensitive Type Qualifiers” *Trans. on Programming Languages and Systems*. 28(6):1035-1087, Nov. 2006.
- [6] D. Grossman, “Quantified Types in an Imperative Language” *ACM Transactions on Programming Languages and Systems*, 2006.
- [7] J. Hughes “Global variables in Haskell” *Journal of Functional Programming archive* Volume 14, Issue 5, Sept. 2004.
- [8] I. S. Diatchki, M. P. Jones, and R. Leslie. “High- level Views on Low-level Representations.” *Proc. ACM Int. Conference on Functional Programming* pp. 168–179, 2005.
- [9] International Std. Organization *ISO/IEC 9899:1999 (Prog. Languages - C)*, 1999.
- [10] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang “Cyclone: A safe dialect of C.” *Proc. of USENIX Annual Technical Conference* pp 275288, 2002.
- [11] M. P. Jones “Qualified types: theory and practice.” *Cambridge Distinguished Dissertations In Computer Science* ISBN:0-521-47253-9, 1995
- [12] S. L. Peyton Jones and J. Launchbury “Unboxed values as first class citizens in a non-strict functional language.” *Functional Programming Languages and Computer Architecture*, 1991
- [13] S. L. Peyton Jones and P. Wadler “Imperative functional programming.” *Proc. ACM SIGPLAN Principles of Programming Languages.*, 1993
- [14] S. L. Peyton Jones (ed.). *Haskell 98 Language and Libraries: The Revised report*. Cambridge University Press, 2003.
- [15] R. Milner “A theory of type polymorphism in programming.” *Journal of Computer and System Sciences* pp 348-375, 1978.
- [16] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML - Revised* The MIT Press, May 1997.
- [17] J. Gosling, B. Joy, G. Steele, and G. Bracha “The Java Language Specification,” Third Edition <http://java.sun.com/docs/books/jls>
- [18] G. van Rossum, “Python Reference Manual” F. L. Drake, Jr. (ed.) <http://docs.python.org/ref/ref.html>, 2006.
- [19] J. S. Shapiro, J. M. Smith, and D. J. Farber. “EROS: a fast capability system” *ACM Symposium on Operating Systems Principles*, Dec. 1999.
- [20] J. S. Shapiro, S. Sridhar, M. S. Doerrie, “BitC Language Specification” <http://www.bitc-lang.org/docs/bitc/spec.html>

$$\begin{array}{c}
\frac{[\kappa \mapsto \varkappa] \notin \theta}{\theta(\kappa) = \varkappa} \quad \frac{[\kappa \mapsto \varkappa] \in \theta}{\theta(\kappa) = \varkappa} \quad \frac{\theta(\tau_1) = \tau'_1 \quad \theta(\tau_2) = \tau'_2}{\theta(\tau_1 = \tau_2) = (\tau'_1 = \tau'_2)} \quad \frac{\theta(\varkappa) = \varkappa' \quad \theta(\tau) = \tau'}{\theta(\star_x^\varkappa(\tau)) = \star_x^{\varkappa'}(\tau')} \quad \frac{\forall u \in \mathcal{C}, \theta(u) \in \mathcal{C}'}{\theta(\mathcal{C}) = \mathcal{C}'} \\
\frac{}{\theta(\alpha) = \alpha} \quad \frac{\theta(\rho) = \rho'}{\theta(\alpha \simeq \rho) = \alpha \simeq \nabla(\rho')} \quad \frac{\theta(\rho) = \rho'}{\theta(\alpha \cong \rho) = \alpha \cong \nabla(\rho')} \quad \frac{\forall c \in \mathcal{C}, \theta(c) \in \mathcal{C}'}{\theta(\mathcal{C}) = \mathcal{C}'} \\
\frac{\text{ftv}(\theta) \cap \{\bar{\alpha}\}}{\theta(\tau) = \tau' \quad \theta(\mathcal{D}) = \mathcal{D}'} \quad \frac{\forall x \mapsto \sigma \in \Gamma, \quad x \mapsto \theta(\sigma) \in \Gamma'}{\theta(\Gamma) = \Gamma'} \quad \frac{\forall L \mapsto \tau \in \Sigma, \quad L \mapsto \theta(\tau) \in \Sigma'}{\theta(\Sigma) = \Sigma'} \quad \frac{\forall \kappa_j \text{ in } e, \theta(\kappa_j) = \varkappa_j \quad e' = e[\varkappa_j/\bar{\kappa}_j]}{\theta(e) = e'}
\end{array}$$

Figure 7: Substitution Rules

- [21] G. Smith and D. Volpano. “A sound polymorphic type system for a dialect of C.” *Science of Computer Programming* **32**(2–3):49–72, 1998.
- [22] S. Sridhar and J. S. Shapiro. “Type Inference for Unboxed Types and First Class Mutability” *Proc. 3rd Workshop on Prog. Languages and Operating Systems*, 2006.
- [23] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. “TIL: A type-directed optimizing compiler for ML” *Proc. ACM SIGPLAN PLDI*, 1996.
- [24] M. S. Tschantz and M. D. Ernst, “Javari: Adding reference immutability to Java” *Object-Oriented Programming Systems, Languages, and Applications*, Oct 2005.
- [25] A. Wright, “Simple Imperative Polymorphism” *Lisp and Symbolic Comp.* 8(4):343-355, 1995.

A Properties of \mathbb{B} Type Algebra

Syntax

Cmp. Constraints	$c ::= \alpha \mid \alpha \simeq \rho \mid \alpha \cong \rho$
Cmp. Constraint Sets	$\mathcal{C} ::= \emptyset \mid \{\bar{c}\} \mid \mathcal{C} \cup \mathcal{C}$
Poly. Constraints	$d ::= \star_x^\varkappa(\tau)$
Poly. Constraint Sets	$\mathcal{D} ::= \emptyset \mid \{\bar{d}\} \mid \mathcal{D} \cup \mathcal{D}$
Unf. Constraints	$u ::= \tau = \tau \mid \kappa = \varkappa \mid d$
Unf. Constraint Sets	$\mathcal{C} ::= \emptyset \mid \{\bar{u}\} \mid \mathcal{C} \cup \mathcal{C} \mid \mathcal{C} \cup \mathcal{C}$
Solvable Entities	$\omega ::= \tau \mid \mathcal{C} \mid \sigma \mid \Gamma \mid \Sigma$

We represent mathematical properties as: assumption $\stackrel{\text{property}}{=}$ subject. As a matter of notational convenience in the case of substitutions, we write: $\theta_{a,b}$ to mean $\theta_a \circ \theta_b$. We also abbreviate substitution over sets $\theta\{\dots\}$ as $\theta\{\dots\}$. The operator \cup represents the disjoint union of sets. The \otimes operator denotes mutual exclusion of sets. That is, $\chi_1 \otimes \chi_2$ iff $\chi_1 \cap \chi_2 = \emptyset$. As is customary, we write $\chi_1 \otimes \chi_2 \otimes \chi_3$ iff $\chi_1 \otimes \chi_2$ and $\chi_2 \otimes \chi_3$.

Definition 1 (Algebraic equivalences). *In our algebra of types, we define the following equivalence: $\Psi\Psi\rho = \Psi\rho$. That is, the mutable type constructor is idempotent.*

Definition 2 (Structural Containment). *We define a structural containment relation $\tau \in \omega$ as follows:*

1. $\tau \in \tau'$ if τ is structurally present as a part of τ' .
2. $\tau \in \forall \bar{\alpha}. \tau'$ if $\tau \in \{\bar{\alpha}\}$ or $\tau \in \tau'$.
3. $\tau \in \Gamma$ if $\exists x \mapsto \sigma \in \Gamma$ such that $\tau \in \sigma$.
4. $\tau \in \Sigma$ if $\exists L \mapsto \tau' \in \Sigma$ such that $\tau \in \tau'$.
5. $\tau \in \bar{\omega}$ if $\tau \in \omega$, for any $\omega \in \{\bar{\omega}\}$.

Definition 3 (Well-formed Substitutions). *A substitution θ is said to be well-formed with respect to a sentence X in the above grammar if:*

1. θ is idempotent.
2. $\theta\langle X \rangle$ is still a valid sentence in the same language.

The first condition requires that for any X , $\theta\langle X \rangle = \theta\langle\theta\langle X \rangle\rangle$. The actual condition we require here is that the substitution θ satisfies the occurs check. Since any substitution that satisfies the occurs check can be written equivalently as an idempotent substitution, we require this stronger property without loss of generality. This means that substitutions such as $[\alpha \mapsto \beta] \circ [\beta \mapsto \text{unit}]$ are not well formed, and must instead be written (equivalently) as: $[\alpha \mapsto \text{unit}] \circ [\beta \mapsto \text{unit}]$. An implication of idempotence is that $\text{dom}(\theta) \cap \text{range}(\theta) = \emptyset$.

The second condition requires that the substitution θ does not violate the syntax of the language. For example, the substitution $[\alpha \mapsto \beta] \tau$ is not a well formed substitution on $\Psi\alpha$. In the rest of the document, we say substitutions to mean well-formed substitutions unless otherwise specified.

Definition 4 (Specialization). We write $\tau_1 \sqsubseteq \tau_2$, that is, τ_1 is a specialization of (or less general than) τ_2 iff $\exists \theta$ such that $\tau_1 = \theta\langle \tau_2 \rangle$. We write $\tau_1 \supseteq \tau_2$, that is, τ_1 is more general than τ_2 iff $\tau_2 \sqsubseteq \tau_1$.

We write $\theta_1 \sqsubseteq \theta_2$ iff $\exists \theta$ such that $\theta_1 = \theta \circ \theta_2$. We write $\theta_1 \supseteq \theta_2$ iff $\theta_2 \sqsubseteq \theta_1$.

Definition 5 (Constraint Satisfaction). We write $\theta \models_{\text{sol}} \mathcal{C}$ to denote the fact that the substitution θ satisfies the set of constraints \mathcal{C} as defined below:

$$\frac{\begin{array}{l} \forall \star_x^\psi(\tau_1) \dots \star_x^\psi(\tau_n) \in \mathcal{D}, \tau_1 = \dots = \tau_n \\ \forall \star_y^\forall(\tau) \in \mathcal{D}, \text{Immut}(\tau) \\ \forall \star_x^{\varkappa_1}(\tau_1) \in \mathcal{D} \text{ and } \star_y^{\varkappa_2}(\tau_2) \in \mathcal{D}, \varkappa_1 \neq \varkappa_2 \text{ implies } x \neq y \end{array}}{\models \mathcal{D}}$$

$$\frac{\begin{array}{l} \forall (\tau_1 = \tau_2) \in \mathcal{C}, \theta\langle \tau_1 \rangle = \theta\langle \tau_2 \rangle \\ \forall (\kappa = \varkappa) \in \mathcal{C}, \theta\langle \kappa \rangle = \theta\langle \varkappa \rangle \\ \mathcal{D} = \{ \forall \theta\langle \star_x^\varkappa(\tau) \rangle \mid \star_x^\varkappa(\tau) \in \mathcal{C} \} \end{array}}{\theta \models_{\text{sol}} \mathcal{C} \rightsquigarrow \mathcal{D}}$$

$$\frac{\theta \models_{\text{sol}} \mathcal{C} \rightsquigarrow \mathcal{D} \quad \models \mathcal{D}}{\theta \models_{\text{sat}} \mathcal{C} \rightsquigarrow \mathcal{D}}$$

Definition 6 (Application to Solvable Entities). If F is a function from τ to $\{\tau\}$, we extend the definition of the function to all solvable entities (that is, ω to $\{\tau\}$) as follows:

$$\begin{aligned} F(\forall \bar{\alpha}. \tau \setminus \mathcal{D}) &= F(\tau) \cup F(\mathcal{D}) \setminus \{\bar{\alpha}\} \\ F(\Gamma) &= \bigcup F(\sigma_j), \forall x_j \mapsto \sigma_j \in \Gamma \\ F(\Sigma) &= \bigcup F(\tau_j), \forall L_j \mapsto \tau_j \in \Sigma \end{aligned}$$

We also write

$$\begin{aligned} F(\bar{\omega}) &= \bigcup F(\omega) \\ F(\{\bar{\omega}\}) &= \bigcup F(\omega) \\ F(\theta) &= F(\text{dom}(\theta)) \cup F(\text{range}(\theta)) \\ F(\alpha) &= F(\alpha) \\ F(\alpha \simeq \tau) &= F(\alpha) \cup F(\tau) \\ F(\alpha \cong \tau) &= F(\alpha) \cup F(\tau) \\ F(\tau_1 = \tau_2) &= F(\tau_1) \cup F(\tau_2) \\ F(\star_x^\varkappa(\tau)) &= F(\tau) \\ F(\mathcal{C}) &= \bigcup F(u_j), \forall u_j \in \mathcal{C} \\ F(\mathcal{C}) &= \bigcup F(c_j), \forall c_j \in \mathcal{C} \end{aligned}$$

Definition 7 (Constraint Collection). $\{\tau\}$ denotes the set of all constrained types and unconstrained type variables structurally present in τ , as defined in Fig. 3.

The $\{\tau\}$ relation, is extended to $\{\omega\}$ through the F operator (Definition 6), except in the case of type schemes (σ). Here, we take a monomorphic view of constraints embedded in type schemes and define $\{\sigma\}$ as:

$$\{\forall \bar{\alpha}. \tau \setminus \mathcal{D}\} = \{\tau\} \cup \{\mathcal{D}\}$$

The collected constraints can be understood to be working in a flat overlay type system, which operates above the tree-structured underlying type system that respects variable quantification.

Definition 8 (Free Type Variables). We denote the set of free type variables in a type τ as $\text{ftv}(\tau)$.

$$\begin{aligned}
\text{ftv}(\alpha) &= \{\alpha\} \\
\text{ftv}(\text{unit}) &= \{\} \\
\text{ftv}(\text{bool}) &= \{\} \\
\text{ftv}(\uparrow\tau) &= \text{ftv}(\tau) \\
\text{ftv}(\Psi\rho) &= \text{ftv}(\tau) \\
\text{ftv}(\tau_1 \rightarrow \tau_2) &= \text{ftv}(\tau_1) \cup \text{ftv}(\tau_2) \\
\text{ftv}(\tau_1 \times \tau_2) &= \text{ftv}(\tau_1) \cup \text{ftv}(\tau_2) \\
\text{ftv}(\alpha\downarrow\rho) &= \text{ftv}(\alpha) \cup \text{ftv}(\rho) \\
\text{ftv}(\varsigma\downarrow\rho) &= \text{ftv}(\varsigma) \cup \text{ftv}(\rho)
\end{aligned}$$

Definition 9 (mtvs , Mtv s, Ntv s). $\text{mtv}(\omega)$, $\text{Mtv}(\omega)$, and $\text{ntv}(\omega)$ denote the set of all type variables appearing in ω constrained by $\alpha\downarrow\rho$, by $\varsigma\downarrow\rho$ and unconstrained respectively.

1. $\text{mtv}(\bar{\tau}) = \{\alpha \mid \alpha\downarrow\rho \in \{\bar{\tau}\}\}$
2. $\text{Mtv}(\bar{\tau}) = \{\alpha \mid \alpha\downarrow\rho \in \{\bar{\tau}\} \text{ or } \Psi\alpha\downarrow\rho \in \{\bar{\tau}\}\}$
3. $\text{ntv}(\bar{\tau}) = \{\alpha \mid \alpha \in \{\bar{\tau}\}\}$

Definition 10 (Compatibility Constraint Set). We write $\mathfrak{R}(C)$ if the constraint set C

1. A type variable α appears at most once in C in only one of the positions: α , $\alpha \simeq \rho$, or $\alpha \cong \rho$.
2. $\forall \alpha \simeq \rho \in C, \rho = \blacktriangledown(\rho)$.
3. $\forall \alpha \cong \rho \in C, \rho = \triangledown(\rho)$.

We write $\text{dom}(C) = \{\bar{\alpha} \mid \alpha \in C, \alpha \simeq \rho \in C, \text{ or } \alpha \cong \rho \in C\}$. We write $\text{range}(C) = \{\bar{\rho} \mid \alpha \simeq \rho \in C, \text{ or } \alpha \cong \rho \in C\}$.

Definition 11 (Consistency of Maybe types). For any constraint set C such that $\mathfrak{R}(C)$, we write $C \Vdash \tau$ to denote that fact that the constraints embedded in τ are consistent with C . Similarly for $C \Vdash \{\bar{\omega}\}$.

$$\begin{array}{c}
\frac{\alpha \in C}{C \Vdash \alpha} \quad \frac{\alpha \simeq \rho \in C \quad \rho \stackrel{\blacktriangledown}{=} \rho'}{C \Vdash \alpha\downarrow\rho'} \\
\frac{\alpha \cong \rho \in C \quad \varsigma \stackrel{\blacktriangledown}{=} \alpha \quad \rho \stackrel{\triangledown}{=} \rho'}{C \Vdash \varsigma\downarrow\rho'} \quad \frac{C \Vdash \{\bar{\omega}\}}{C \Vdash \{\bar{\omega}\}} \\
\frac{\{\bar{\tau}\} = \{\bar{\tau}\} \quad \forall \tau \in \{\bar{\tau}\}, C \Vdash \tau}{C \Vdash \{\bar{\tau}\}} \quad \frac{\exists C \mid C \Vdash \{\bar{\omega}\}}{\Vdash \{\bar{\omega}\}}
\end{array}$$

Definition 12 (Consistency of substitutions). A consistent substitution is a well formed substitution that does not violate the constraints embedded as maybe types.

$$\frac{\forall \alpha\downarrow\rho \in \{\bar{\omega}\}, \theta\langle\alpha\rangle = \beta \text{ or } \theta\langle\alpha\rangle = \rho' \mid \rho' \stackrel{\blacktriangledown}{=} \theta\langle\rho\rangle \quad \forall \varsigma\downarrow\rho \in \{\bar{\omega}\}, \theta\langle\varsigma\rangle = \varsigma' \text{ or } \theta\langle\varsigma\rangle = \varrho \mid \varrho \stackrel{\triangledown}{=} \theta\langle\rho\rangle}{\theta \models_{\text{cst}} \{\bar{\omega}\}}$$

$$\frac{\theta \models_{\text{cst}} \{\bar{\omega}\} \quad \Vdash \theta\{\bar{\omega}\}}{\theta \Vdash \{\bar{\omega}\}}$$

Lemma 1 (Properties of consistent types). 1. $\Vdash \{\bar{\omega}\}$ iff

- (a) $\forall \{\alpha\downarrow\rho, \alpha\downarrow\rho'\} \subseteq \{\bar{\omega}\}$, we have $\rho \stackrel{\blacktriangledown}{=} \rho'$.

(b) $\forall \{\varsigma \downarrow \rho, \varsigma' \downarrow \rho'\} \subseteq \{\overline{\omega}\}$ such that $\varsigma \stackrel{\nabla}{=} \varsigma'$, we have $\rho \stackrel{\nabla}{=} \rho'$.

(c) $\text{mtv}(\overline{\omega}) \circ \text{Mtv}(\overline{\omega}) \circ \text{ntv}(\overline{\omega})$

2. If $C \Vdash \{\overline{\omega}\}$, then $\Vdash \{\overline{\omega}\}$.
3. If $\Vdash \{\overline{\omega}\}$, then $\exists C$ such that $C \Vdash \{\overline{\omega}\}$.
4. If $\Vdash \{\overline{\omega_b}\}$, $C \Vdash \{\overline{\omega_s}\}$ and $\{\overline{\omega_s}\} \subseteq \{\overline{\omega_b}\}$, then $\exists C' \supseteq C$ such that $C \Vdash \{\overline{\omega_b}\}$.
5. $\Vdash \{\overline{\omega}\}$ iff $\Vdash \{\overline{\omega}\}$
6. $C \Vdash \{\overline{\omega}\}$ iff $C \Vdash \{\overline{\omega}\}$
7. If $C \Vdash \{\overline{\omega}\}$, then $\text{ftv}(\{\overline{\omega}\}) \subseteq \text{dom}(C)$.
8. If $C \Vdash \{\overline{\omega}\}$ and $\text{dom}(C) = \text{ftv}(\{\overline{\omega}\})$, then $\text{ftv}(\text{range}(C)) \subseteq \text{ftv}(\{\overline{\omega}\})$.

Proof. Evident from Definition 11. □

Lemma 2 (Weakening of Consistency). *If $\{\overline{\omega_2}\} \subseteq \{\overline{\omega_1}\}$, then,*

1. $C \Vdash \{\overline{\omega_1}\}$ implies $C \Vdash \{\overline{\omega_2}\}$
2. $\Vdash \{\overline{\omega_1}\}$ implies $\Vdash \{\overline{\omega_2}\}$
3. $\theta \Vdash_{\text{cst}} \{\overline{\omega_1}\}$ implies $\theta \Vdash_{\text{cst}} \{\overline{\omega_2}\}$
4. $\theta \Vdash \{\overline{\omega_1}\}$ implies $\theta \Vdash \{\overline{\omega_2}\}$
5. If $C \Vdash \{\overline{\omega}\}$, then $\forall C' \supseteq C$ such that $\mathfrak{R}(C')$, $C' \Vdash \{\overline{\omega}\}$.
6. If $\theta \Vdash \{\overline{\omega}\}$, then $\theta \Vdash_{\text{cst}} \{\overline{\omega}\}$
7. If $\theta \Vdash \{\overline{\omega}\}$, then $\Vdash \theta\{\overline{\omega}\}$.

Proof. Evident from Definition 11 and Definition 12. □

Lemma 3 (Substitution Canonicalization). *For any type τ and substitution θ , such that $\theta \Vdash_{\text{cst}} \{\tau\}$, we have:*

1. $\nabla(\theta\langle\nabla(\tau)\rangle) = \nabla(\theta\langle\blacktriangle(\tau)\rangle) = \nabla(\theta\langle\tau\rangle)$
2. $\blacktriangle(\theta\langle\blacktriangle(\tau)\rangle) = \blacktriangle(\theta\langle\nabla(\tau)\rangle) = \blacktriangle(\theta\langle\tau\rangle)$
3. $\nabla(\theta\langle\nabla(\tau)\rangle) = \nabla(\theta\langle\triangle(\tau)\rangle) = \nabla(\theta\langle\tau\rangle)$
4. $\triangle(\theta\langle\triangle(\tau)\rangle) = \triangle(\theta\langle\nabla(\tau)\rangle) = \triangle(\theta\langle\tau\rangle)$

Proof. By straightforward induction on the structure of τ . □

Lemma 4 (Substitution over mutability Minimization). *If $\theta \Vdash_{\text{cst}} \{\tau_1, \tau_2\}$, then*

1. $\tau_1 \stackrel{\nabla}{=} \tau_2$ implies $\theta\langle\tau_1\rangle \stackrel{\nabla}{=} \theta\langle\tau_2\rangle$
2. $\tau_1 \stackrel{\nabla}{=} \tau_2$ implies $\theta\langle\tau_1\rangle \stackrel{\nabla}{=} \theta\langle\tau_2\rangle$

Proof. 1. Property 1: We have $\tau_1 \stackrel{\nabla}{=} \tau_2$. That is, $\nabla(\tau_1) = \nabla(\tau_2)$. Therefore, we must have $\theta\langle\nabla(\tau_1)\rangle = \theta\langle\nabla(\tau_2)\rangle$, and further, $\nabla(\theta\langle\nabla(\tau_1)\rangle) = \nabla(\theta\langle\nabla(\tau_2)\rangle)$.

2. From $\theta \models_{\text{cst}} \{\tau_1, \tau_2\}$ and Lemma 2 (weakening), we have $\theta \models_{\text{cst}} \tau_1$.
3. Now using case (2) and Lemma 3, we obtain $\nabla(\theta\langle\nabla(\tau_1)\rangle) = \nabla(\theta\langle\tau_1\rangle)$. Similarly, we obtain $\nabla(\theta\langle\nabla(\tau_2)\rangle) = \nabla(\theta\langle\tau_2\rangle)$.
4. Substituting the results of case (3) in case (1), we obtain $\nabla(\theta\langle\tau_1\rangle) = \nabla(\theta\langle\tau_2\rangle)$. That is, $\theta\langle\tau_1\rangle \stackrel{\nabla}{=} \theta\langle\tau_2\rangle$.
5. Property 2: Similar to Property 1. □

Lemma 5 (Aggregation of Consistency). 1. If $\theta \models_{\text{cst}} \{\overline{\omega}_1\}$ and $\theta \models_{\text{cst}} \{\overline{\omega}_2\}$, then $\theta \models_{\text{cst}} \{\overline{\omega}_1, \overline{\omega}_2\}$.

2. If $C \Vdash \{\overline{\omega}_1\}$ and $C \Vdash \{\overline{\omega}_2\}$, then $C \Vdash \{\overline{\omega}_1, \overline{\omega}_2\}$.

Proof. Evident from Definition 12 (consistent substitution) and Definition 11 (consistent types). □

Lemma 6 (Strengthening of Consistency). *If*

1. $C \Vdash \{\tau_1, \tau_2\}$
2. $\theta \models_{\text{cst}} \{\tau_1, \tau_2\}$
3. $\theta\langle C \rangle \Vdash \theta\{\tau_1\}$
4. $\text{dom}(\theta) \cap \text{ftv}(\tau_2) \subseteq \text{ftv}(\tau_1)$

Then, $\theta\langle C \rangle \Vdash \theta\{\tau_2\}$

Proof. Let $C_s = \theta\langle C \rangle$. The proof is by induction on the structure of τ_2 . We proceed by case analysis on the final step.

1. From premise (1) and Lemma 1 (property-3), we obtain $C \Vdash \{\tau_1, \tau_2\}$. Using Lemma 2 (weakening), we obtain $C \Vdash \{\tau_1\}$, $C \Vdash \{\tau_1\}$, $C \Vdash \{\tau_2\}$, and $C \Vdash \{\tau_2\}$.
2. Case: $\tau_2 = \alpha$. Due to $C \Vdash \{\tau_2\}$, we have $\alpha \in C$. We proceed by further case analysis.
 - (a) Case $[\alpha \mapsto \tau] \notin \theta$: Here, $\theta\langle\tau_2\rangle = \tau_2 = \alpha$. Since $\theta\langle\alpha\rangle = \alpha$, $\alpha \in C_s$. Further, $\theta\{\tau_2\} = \theta\{\alpha\} = \theta\{\alpha\} = \{\alpha\}$. Now, it is evident that $C_s \Vdash \theta\{\tau_2\}$.
 - (b) Case $[\alpha \mapsto \tau] \notin \theta$: Here, $\theta\langle\tau_2\rangle = \tau$.
 - i. Since $\alpha \in \text{dom}(\theta)$, due to premise (3), we have $\alpha \in \text{ftv}(\tau_1)$, and due to premise (1) $\{\alpha\} \subseteq \{\tau_1\}$.
 - ii. Therefore, $\theta\{\alpha\} \subseteq \theta\{\tau_1\}$. That is, $\{\tau\} \subseteq \theta\{\tau_1\}$.
 - iii. From case (3.b.ii), premise (3), and Lemma 2 (weakening), we obtain $C_s \Vdash \{\tau\}$.
 - iv. We know that $\{\tau_2\} = \{\alpha\} = \{\alpha\}$. Therefore, $\theta\{\tau_2\} = \theta\{\alpha\} = \{\tau\}$.
 - v. From cases (3.c and 3.d), we conclude that $C_s \Vdash \theta\{\tau_2\}$.
3. Case $\tau_2 = \alpha \downarrow \rho$:
 - (a) Due to premise (1), we have $\alpha \simeq \rho_c \in C$, for some $\rho_c \stackrel{\nabla}{=} \rho$.
 - (b) By induction hypothesis, we have $C_s \Vdash \theta\{\rho\}$. Note that we do not apply induction hypothesis on α since it is not an independent first-class component of the type $\alpha \downarrow \rho$.
 - (c) Since $\{\tau_2\} = \{\rho\} \cup \{\alpha \downarrow \rho\}$, $\theta\{\tau_2\} = \theta\{\rho\} \cup \theta\{\alpha \downarrow \rho\}$. Given the hypothesis in case (3.b), if we have $C_s \Vdash \theta\{\alpha \downarrow \rho\}$, we can prove $C_s \Vdash \theta\{\alpha \downarrow \rho\}$ using Lemma 5. Now, we proceed by further case analysis to prove $C_s \Vdash \theta\{\alpha \downarrow \rho\}$.
 - (d) Case $\theta\langle\alpha\rangle = \alpha$: That is, $\alpha \notin \text{dom}(\theta)$. Here, $\theta\langle\alpha \downarrow \rho\rangle = \alpha \downarrow \theta\langle\rho\rangle$.
 - i. Since $\alpha \simeq \rho_c \in C$, we have $\alpha \simeq \theta\langle\rho_c\rangle \in C_s$.

- ii. From $\rho_c \stackrel{\nabla}{=} \rho$, premise (2), and Lemma 4, we obtain $\theta\langle\rho_c\rangle \stackrel{\nabla}{=} \theta\langle\rho\rangle$.
- iii. From cases (3.d.i and 3.d.ii) and Definition 11, we obtain $C_s \Vdash \alpha\downarrow\theta\langle\rho\rangle$. That is, $C_s \Vdash \theta\langle\alpha\downarrow\rho\rangle$.
- (e) Case $\theta\langle\alpha\rangle = \beta$: Here, $\theta\langle\alpha\downarrow\rho\rangle = \beta\downarrow\theta\langle\rho\rangle$.
 - i. Since $\alpha \in \text{dom}(\theta)$, due to premise (3), we have $\alpha \in \text{ftv}(\tau_1)$. Due to premise (1), Definition 11, and Lemma 1 (property-1), it must be true that $\{\alpha\downarrow\rho_1\} \subseteq \{\tau_1\}$, for some $\rho_1 \stackrel{\nabla}{=} \rho$.
 - ii. Therefore, $\{\beta\downarrow\theta\langle\rho_1\rangle\} \subseteq \theta\{\tau_1\}$
 - iii. From premise (3), case (3.e.ii) and Lemma 2 (weakening), we obtain $C_s \Vdash \beta\downarrow\theta\langle\rho_1\rangle$.
 - iv. From case (3.e.iii) and Definition 11, we conclude that $\beta \simeq \rho_c \in C_s$ such that $\rho_c \stackrel{\nabla}{=} \theta\langle\rho_1\rangle$.
 - v. From case (3.e.i), premise (2), and Lemma 4, we obtain $\theta\langle\rho_1\rangle \stackrel{\nabla}{=} \theta\langle\rho\rangle$. Using this with case (3.e.iv), we conclude that $\rho_c \stackrel{\nabla}{=} \theta\langle\rho\rangle$.
 - vi. From cases (3.e.iv and 3.e.v), and Definition 11, we obtain $C_s \Vdash \beta\downarrow\theta\langle\rho\rangle$. That is, $C_s \Vdash \theta\langle\alpha\downarrow\rho\rangle$.
- (f) Case $\theta\langle\alpha\rangle = \rho'$: Here, $\theta\langle\alpha\downarrow\rho\rangle = \rho'$.
 - i. Similar to case (3.d.i), we have $\{\alpha\downarrow\rho_1\} \subseteq \{\tau_1\}$, for some $\rho_1 \stackrel{\nabla}{=} \rho$. Therefore, $\{\rho'\} \subseteq \theta\{\tau_1\}$.
 - ii. From premise (3), case (3.f.i) and Lemma 2 (weakening), we obtain $C_s \Vdash \{\rho'\}$. That is, $C_s \Vdash \theta\langle\alpha\downarrow\rho\rangle$.
- 4. Case $\tau_2 = \alpha\downarrow\rho$: Similar to case (2).
- 5. $\tau_2 = \alpha\downarrow\rho$ and $\tau_2 = \alpha\downarrow\rho$ are trivial.
- 6. $\tau_2 = \Psi\rho$ and $\tau_2 = \uparrow\tau'_2$ follow from induction hypothesis.
- 7. $\tau_2 = \tau'_2 \rightarrow \tau''_2$ and $\tau_2 = \tau'_2 \times \tau''_2$ follow from induction hypothesis and Lemma 5.

□

Lemma 7 (Corollary to Strengthening of Consistency). *If $C \Vdash \{\overline{\omega}_b\}$, $\theta \Vdash_{\text{cst}} \{\overline{\omega}_b\}$, $\{\overline{\omega}_s\} \subseteq \{\overline{\omega}_b\}$, $\theta\langle C \rangle \Vdash \theta\{\overline{\omega}_s\}$, and $\text{dom}(\theta) \cap \text{ftv}(\overline{\omega}_b) \subseteq \text{ftv}(\overline{\omega}_s)$ then, $\theta\langle C \rangle \Vdash \theta\{\overline{\omega}_b\}$.*

Proof. By repeated application of Lemma 6, and using Lemma 5. □

B Declarative Type System

Definition 13 (Canonical Expressions). *We say that an expression e is canonical if all let expressions in e are annotated with one of the kinds ψ or \forall .*

Definition 14 (Weakened Type Derivation). *We write $\mathcal{D}; \Gamma; \Sigma \vdash_{\text{w}} e : \tau$ if $\mathcal{D}'; \Gamma; \Sigma \vdash e : \tau$ for some $\mathcal{D}' \subseteq \mathcal{D}$.*

Definition 15 (Constraint Collection over Type Derivation). *We write $\{\mathcal{D}; \Gamma; \Sigma \vdash e : \tau\}$ to denote the set of all constrained types and unconstrained type variables used in the derivation of $\mathcal{D}; \Gamma; \Sigma \vdash e : \tau$.*

$$\begin{aligned}
\{\mathcal{D}; \Gamma; \Sigma \vdash x : \tau\} &= \{\Gamma, \Sigma, \tau, \mathcal{D}\} \\
\{\mathcal{D}; \Gamma; \Sigma \vdash () : \text{unit}\} &= \{\Gamma, \Sigma, \mathcal{D}\} \\
\{\mathcal{D}; \Gamma; \Sigma \vdash \text{true} : \text{bool}\} &= \{\Gamma, \Sigma, \mathcal{D}\} \\
\{\mathcal{D}; \Gamma; \Sigma \vdash \text{false} : \text{bool}\} &= \{\Gamma, \Sigma, \mathcal{D}\} \\
\{\mathcal{D}; \Gamma; \Sigma \vdash \ell : \uparrow\tau\} &= \{\Gamma, \Sigma, \tau, \mathcal{D}\} \\
\{\mathcal{D}; \Gamma; \Sigma \vdash l : \tau\} &= \{\Gamma, \Sigma, \tau, \mathcal{D}\} \\
\{\mathcal{D}; \Gamma; \Sigma \vdash \lambda x. e : \tau'_1 \rightarrow \tau'_2\} &= \{\mathcal{D}; \Gamma, x \mapsto \tau_1; \Sigma \vdash e : \tau_2\} \\
&\quad \cup \{\tau'_1, \tau'_2\} \\
\{\mathcal{D}; \Gamma; \Sigma \vdash e_1 e_2 : \tau\} &= \{\mathcal{D}; \Gamma; \Sigma \vdash e_1 : \tau_1\} \\
&\quad \cup \{\mathcal{D}; \Gamma; \Sigma \vdash e_2 : \tau_2\} \\
&\quad \cup \{\tau_a, \tau_r, \tau\}
\end{aligned}$$

Other cases are similar. We write:

$$\{\mathcal{D}; \Gamma; \Sigma \vdash_s e : \tau\} = \{\mathcal{D}'; \Gamma; \Sigma \vdash_s e : \tau, \mathcal{D}\}.$$

Definition 16 (Consistent Type Derivation). *We say that $C; \mathcal{D}; \Gamma; \Sigma \vdash e : \tau$, is a consistent type derivation under the constraint set C if $\mathcal{D}; \Gamma; \Sigma \vdash e : \tau$ and $C \Vdash \{\mathcal{D}; \Gamma; \Sigma \vdash e : \tau\}$.*

Similarly, we write $C; \mathcal{D}; \Gamma; \Sigma \vdash_s e : \tau$ if $\mathcal{D}; \Gamma; \Sigma \vdash_s e : \tau$ and $C \Vdash \{\mathcal{D}; \Gamma; \Sigma \vdash_s e : \tau\}$.

We say that $\mathcal{D}; \Gamma; \Sigma \vdash_* e : \tau$, is a consistent type derivation if $\mathcal{D}; \Gamma; \Sigma \vdash_s e : \tau$ and $\Vdash \{\mathcal{D}; \Gamma; \Sigma \vdash_s e : \tau\}$.

Definition 17 (Stack and Heap Typing). *A heap H and a stack S are said to be well typed with respect to a binding context Γ and store typing Σ , and written $\mathcal{D}; \Gamma; \Sigma \vdash_s H + S$ if*

1. $\text{dom}(\Sigma) = \text{dom}(H) \cup \text{dom}(S)$
2. $\forall \ell \in \text{dom}(H), \mathcal{D}; \Gamma; \Sigma \vdash_s H(\ell) : \tau$ such that $\Sigma(\ell) \stackrel{\nabla}{=} \tau$
3. $\forall l \in \text{dom}(S), \mathcal{D}; \Gamma; \Sigma \vdash_s S(l) : \tau$ such that $\Sigma(l) \stackrel{\nabla}{=} \tau$

Similarly, we define $C; \mathcal{D}; \Gamma; \Sigma \vdash_s H + S$ and $\mathcal{D}; \Gamma; \Sigma \vdash_* H + S$

Definition 18 (Valid Lvalues). *We say that an lvalue \mathcal{L} is valid with respect to a stack S and heap H , written $H + S \vdash_v \mathcal{L}$ if one of the following conditions hold for some p :*

1. $\mathcal{L} = l$ or $\mathcal{L} = l.p$, where $l \in \text{dom}(S)$.
2. $\mathcal{L} = \ell^\wedge$ or $\mathcal{L} = \ell^\wedge.p$, where $\ell \in \text{dom}(H)$.

Lemma 8 (Inversion of Typing Relation). *1. If $\mathcal{D}; \Gamma; \Sigma \vdash_s () : \tau$ then $\tau = \text{unit}$.*

2. *If $\mathcal{D}; \Gamma; \Sigma \vdash_s \text{true} : \tau$ then $\tau = \text{bool}$.*
3. *If $\mathcal{D}; \Gamma; \Sigma \vdash_s \text{false} : \tau$ then $\tau = \text{bool}$.*
4. *If $\mathcal{D}; \Gamma; \Sigma \vdash_s \ell : \tau$ then $\tau = \uparrow\tau'$.*
5. *If $\mathcal{D}; \Gamma; \Sigma \vdash_s \lambda x. e : \tau$ then $\tau = \tau'_1 \rightarrow \tau'_2$, such that $\mathcal{D}; \Gamma, x \mapsto \tau_1; \Sigma \vdash e : \tau_2$ and $\tau_1 \stackrel{\nabla}{=} \tau'_1$ and $\tau_2 \stackrel{\nabla}{=} \tau'_2$.*
6. *If $\mathcal{D}; \Gamma; \Sigma \vdash_s e^\wedge : \tau$ then $\mathcal{D}; \Gamma; \Sigma \vdash_s e \trianglelefteq \uparrow\tau$.*
7. *Other cases are similar.*

Proof. Immediate from the definition of typing relation. □

Lemma 9 (Inversion of Copy Coercion). *For any type ρ and α , let $Q(\rho) = \{\rho, \Psi\rho, \alpha\downarrow\rho, \alpha\downarrow\Psi\rho, \alpha\downarrow\rho, \alpha\downarrow\Psi\rho, \Psi\alpha\downarrow\rho, \Psi\alpha\downarrow\Psi\rho\}$. Then,*

1. *If $\tau \trianglelefteq \text{bool}$ then $\tau \in Q(\text{bool})$*
2. *If $\tau \trianglelefteq \text{unit}$ then $\tau \in Q(\text{unit})$.*
3. *If $\tau \trianglelefteq \uparrow\tau'$ then $\tau \in Q(\uparrow\tau')$.*
4. *If $\tau \trianglelefteq \tau_1 \rightarrow \tau_2$ then $\tau \in Q(\tau_1 \rightarrow \tau_2)$.*
5. *If $\tau \trianglelefteq \tau_1 \times \tau_2$ then $\tau \in Q(\tau_1 \times \tau_2)$, such that $\tau_1 \trianglelefteq \tau'_1$ and $\tau_2 \trianglelefteq \tau'_2$.*
6. *If $\tau \trianglelefteq \Psi\rho$ then $\tau = \Psi\rho'$, such that $\rho' \trianglelefteq \rho$.*

Proof. By induction on the copy coercion derivation. □

Lemma 10 (Canonical Forms). *1. If $\mathcal{D}; \Gamma; \Sigma \vdash_s v \trianglelefteq \text{unit}$, then, v is $()$.*

2. If $\mathcal{D}; \Gamma; \Sigma \vdash_s v \trianglelefteq: \text{bool}$, then, v is either true or false.
3. If $\mathcal{D}; \Gamma; \Sigma \vdash_s v \trianglelefteq: \uparrow\tau$, then, v is ℓ , $\ell \in \text{dom}(\Sigma)$.
4. If $\mathcal{D}; \Gamma; \Sigma \vdash_s v \trianglelefteq: \tau_1 \rightarrow \tau_2$, then, v is $\lambda x.e$.
5. If $\mathcal{D}; \Gamma; \Sigma \vdash_s v \trianglelefteq: \tau_1 \times \tau_2$, then, v is (v_1, v_2) .

Proof. By inspecting the possibilities for the derivation of $\mathcal{D}; \Gamma; \Sigma \vdash_s v \trianglelefteq: \tau$.

According to the grammar of the language \mathbb{B} , values can be of one of the following forms: $()$, *true*, *false*, ℓ , $\lambda x.e$, (v, v)

Consider the case (2), where $\mathcal{D}; \Gamma; \Sigma \vdash_s v \trianglelefteq: \text{bool}$. That is, $\mathcal{D}; \Gamma; \Sigma \vdash_s v : \tau$ and $\tau \trianglelefteq: \text{bool}$. From Lemma 2 (inversion of copy-coercion), we know that $\tau \in \mathbb{Q}(\text{bool})$. That is, for some α , the type τ equals one of: bool , Ψbool , $\alpha\downarrow\text{bool}$, $\alpha\downarrow\Psi\text{bool}$, $\alpha\downarrow\text{bool}$, $\alpha\downarrow\Psi\text{bool}$, $\Psi\alpha\downarrow\text{bool}$, or $\Psi\alpha\downarrow\Psi\text{bool}$. If $\tau = \text{bool}$, it is clear that the final rule in the derivation must be T-Bool. A derivation with these rules is only possible if $v = \text{b}$. That is, $v = \text{true}$ or $v = \text{false}$. Further, the cases like $\tau = \Psi\text{bool}$ cannot happen because there is no rule that derives a mutable/maybe type for a value.

Other cases of the lemma are similar. □

Theorem 1 (Progress). *If e is a closed well typed canonical expression, that is, $\mathcal{D}; \emptyset; \Sigma \vdash_s e : \tau$ for some τ and Σ , given any heap H and stack S such that $\mathcal{D}; \emptyset; \Sigma \vdash_s H + S$,*

1. *If e is a left expression ($e = l$), then e is either a valid lvalue (that is, $e = \mathcal{L}$ and $H + S \vdash_v \mathcal{L}$) or else $\exists e', S', H'$ such that $S; H; e \Rightarrow S'; H'; e'$.*
2. *e is a value v or else $\exists e', S', H'$ such that $S; H; e \Rightarrow S'; H'; e'$.*

Proof. By induction on the type derivation. $\mathcal{D}; \emptyset; \Sigma \vdash_s e : \tau$

1. Case T-Unit, T-Bool, T-Hloc, T-Lambda: (Values) Result is immediate for right execution, and cannot happen for left execution.
2. Case T-Id: cannot happen. $e = x$ is not a closed term.
3. Case T-Sloc: Immediate for left execution. Right execution and can always continue with E-Rval rule as the stack is well typed ($\mathcal{D}; \emptyset; \Sigma \vdash_s H + S$).
4. Case T-App: Only right execution is possible. We have: $e = e_1 e_2$, $\mathcal{D}; \emptyset; \Sigma \vdash_s e_1 \trianglelefteq: \tau_1 \rightarrow \tau_2$, and $\mathcal{D}; \emptyset; \Sigma \vdash_s e_2 \trianglelefteq: \tau_1$. If e_1 is not a value, execution can continue via the E-# (Right context redex) due to induction hypothesis. Similarly, if e_2 is not a value, we can again take E-#. If both e_1 and e_2 are values, since we know that $\mathcal{D}_1; \emptyset; \Sigma \vdash_s e_1 \trianglelefteq: \tau_1 \rightarrow \tau_2$, from Lemma 3 (canonical forms) we conclude that e_1 is of the form $\lambda x.e'$. Now, we can take the step E-App.
5. Case T-If: Similar to T-App, only right execution is permitted.
6. Case T-Set: Only right execution is applicable. We have: $e = l := e$, $\mathcal{D}; \emptyset; \Sigma \vdash_s l \trianglelefteq: \Psi\rho$, and $\mathcal{D}; \emptyset; \Sigma \vdash_s e \trianglelefteq: \rho$. If l not an lvalue (that is, $l \neq \mathcal{L}$), we can take E-L# for any l by induction hypothesis. If e is not a value, we can take E-#.

Finally, we consider the case where $l = \mathcal{L}$ and $e = v$. \mathcal{L} should be one of $l, l.p, \ell^\wedge$, or $\ell^\wedge.p$. Now, we proceed by induction on the length of \mathcal{L} .

- (a) Suppose $l = \mathcal{L} = l$. We know from induction hypothesis that that $H; S \vdash_v l$ and from Definition 6, we conclude that $l \in \text{dom}(S)$. Now, execution can continue with E-:=Stack.
- (b) Similarly, if $l = \mathcal{L} = \ell^\wedge$, the execution can continue with step E-:=Heap.
- (c) Assume (by hypothesis) that the execution can continue for some $e'_1 = l.p$.

- (d) Now, let $e_1 = 1.1.p$.
- (e) We know that $\mathcal{D}; \emptyset; \Sigma \vdash_s e_1 \leq: \Psi\rho$, which is equivalent to $\mathcal{D}; \emptyset; \Sigma \vdash_s e_1 : \tau_1$ and $\tau_1 \leq: \Psi\rho$. That is, $\mathcal{D}; \emptyset; \Sigma \vdash_s 1.1.p : \tau_1$. The first two steps of this derivation must be T-Sloc and T-Sel. From the assumption of T-Sel rule, we must have $\mathcal{D}; \emptyset; \Sigma \vdash_s l \leq: \tau_1 \times \tau_2$ for some type τ_2 . Now, from the assumption of T-Sloc rule, we must have $\Sigma(l) \leq: \tau_1 \times \tau_2$.
- (f) Now, if $S(l) = v$, since we have $\mathcal{D}; \emptyset; \Sigma \vdash_s S : H$, we conclude that $\mathcal{D}; \emptyset; \Sigma \vdash_s v \leq: \tau_1 \times \tau_2$.
- (g) From case (f) and Lemma 2 (inversion of copy-coercion), we conclude that $v = (v_1, v_2)$.
- (h) From cases (c and g), we conclude that the execution can continue using the E-:=S.p rule.
- (i) Similarly, if $e_1 = 1.2.p$, the execution can again continue using E-:=S.p rule.
- (j) Similar to the above induction, we can show that if $e_1 = \ell^{\wedge}.i.p$, execution can continue using the E-:=H.p rule.
7. Case T-Dup: Only right execution is permitted, and can take E-# or E-Dup as applicable.
8. Case T-Deref: We have: $e = e_1^{\wedge}$, and $\mathcal{D}; \emptyset; \Sigma \vdash_s e_1 \leq: \uparrow\tau$. Execution can take EL-^# or E-# as applicable if e_1 is not a value. If e_1 is a value, then, from Lemma 3 (canonical forms), we conclude that $e_1 = \ell$, $\ell \in \text{dom}(\Sigma)$. Now, since this is an lvalue, we are done in the case of left execution. In the case of right execution, we can take step E-^.
9. Case T-Pair: Similar to case T-Dup
10. Case T-Sel: Similar to case T-Deref.
11. Case T-Let-M: Only right execution is applicable. We have: $e = \text{let}^{\psi} x = e_1 \text{ in } e_2$, $\mathcal{D}; \emptyset; \Sigma \vdash_s e \leq: \tau$. If e_1 is not a value, we can take E-#. Otherwise, we can take E-Let-M.
12. Case T-Let-MP: Only right execution is applicable. We have: $e = (\text{let}^{\varkappa} x = e_1 \text{ in } e_2)$, $\emptyset; \Sigma \vdash_s e \leq: \tau$. If e_1 is not a value, we can take E-#. Otherwise, the execution can take E-Let-P if $\varkappa = \forall$ or E-Let-M if $\varkappa = \psi$.

□

Lemma 11 (Weakening). *If $\mathcal{D}; \Gamma; \Sigma \vdash e : \tau$, then $\forall \Gamma' \supseteq \Gamma$ and $\Sigma' \supseteq \Sigma$, $\mathcal{D}; \Gamma'; \Sigma' \vdash e : \tau$.*

If $\mathcal{D}; \Gamma; \Sigma \vdash_s e : \tau$, then $\forall \Gamma' \supseteq \Gamma$, $\Sigma' \supseteq \Sigma$ and $\mathcal{D}' \supseteq \mathcal{D}$, $\mathcal{D}'; \Gamma'; \Sigma' \vdash_s e : \tau$.

Proof. By straightforward induction on the derivation of $\mathcal{D}; \Gamma; \Sigma \vdash e : \tau$. □

Lemma 12 (Properties of Copy Coercion). *1. If $\tau \leq: \tau'$, then,*

- (a) $\tau \leq: \nabla(\tau')$ and $\Delta(\tau) \leq: \tau'$
- (b) $\Delta(\tau) = \Delta(\tau')$ and $\nabla(\tau) = \nabla(\tau')$.

2. If $\Delta(\tau) = \Delta(\tau')$ or equivalently $\nabla(\tau) = \nabla(\tau')$, then $\tau \leq: \nabla(\tau')$ and $\Delta(\tau) \leq: \tau'$.

3. If $\tau \leq: \tau_1$, and $\tau \leq: \tau_2$, then, $\exists \tau_3$ such that $\tau_1 \leq: \tau_3$, and $\tau_2 \leq: \tau_3$.

4. If $\tau_1 \leq: \tau$, and $\tau_2 \leq: \tau$, then, $\exists \tau_3$ such that $\tau_3 \leq: \tau_1$, and $\tau_3 \leq: \tau_2$.

Proof. Property 1 and 2: By straightforward induction on the derivation of $\tau_1 \leq: \tau_2$.

Property 3: By construction of τ_3 .

- 1. From property 1.b above, we have $\tau \stackrel{\nabla}{=} \tau_1$. Similarly, we have $\tau \stackrel{\nabla}{=} \tau_2$. Therefore, $\tau \stackrel{\nabla}{=} \tau_1 \stackrel{\nabla}{=} \tau_2$.
- 2. Let us pick τ_3 such that $\tau_3 = \tau \stackrel{\nabla}{=} \tau_1 \stackrel{\nabla}{=} \tau_2$.
- 3. From S-Refl rule in figure 4, we know that $\tau_1 \leq: \tau_1$. From this using property 1.a, we obtain $\tau_1 \leq: \nabla(\tau_1)$. From case (2), we can write this as $\tau_1 \leq: \tau_3$. Similarly, we obtain $\tau_2 \leq: \tau_3$.

Property 4: Similar to Property 3. □

Lemma 13 (Subtype Substitution). *1. If $\tau_1 \leq \tau_2$ and $\theta \Vdash_{\text{cst}} \{\tau_1, \tau_2\}$, then, $\theta\langle\tau_1\rangle \leq \nabla(\theta\langle\tau_2\rangle)$ and $\Delta(\theta\langle\tau_1\rangle) \leq \theta\langle\tau_2\rangle$*

2. If $\theta \Vdash_{\text{cst}} \{\alpha\downarrow\rho_1, \beta\downarrow\rho_2\}$, then $\theta\langle\alpha\downarrow\rho_1\rangle \Vdash \theta\langle\beta\downarrow\rho_2\rangle$

3. If $\theta \Vdash_{\text{cst}} \{\varsigma_1\downarrow\rho_1, \varsigma_2\downarrow\rho_2\}$, then $\theta\langle\varsigma_1\downarrow\rho_1\rangle \Vdash \theta\langle\varsigma_2\downarrow\rho_2\rangle$

4. If $\theta \Vdash_{\text{cst}} \{\varsigma_1\downarrow\rho_1, \varsigma_2\downarrow\rho_2\}$, then $\Delta(\theta\langle\varsigma_1\downarrow\rho_1\rangle) \leq \theta\langle\varsigma_2\downarrow\rho_2\rangle$ and $\theta\langle\varsigma_1\downarrow\rho_1\rangle \leq \nabla(\theta\langle\varsigma_2\downarrow\rho_2\rangle)$.

Proof. By using Lemma 5 (properties 1 and 2) and Lemma 4. □

Lemma 14 (Type Renaming). *For any substitution $\theta = \overline{[\alpha \mapsto \beta]}$ where $\{\overline{\beta}\} \cap \text{ftv}(\Gamma, \Sigma, \tau)$,*

1. If $\mathcal{D}; \Gamma; \Sigma \vdash e : \tau$ then $\theta\langle\mathcal{D}\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e : \theta\langle\tau\rangle$.

2. If $C; \mathcal{D}; \Gamma; \Sigma \vdash e : \tau$ then $\theta\langle C\rangle; \theta\langle\mathcal{D}\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e : \theta\langle\tau\rangle$.

Proof. By straightforward induction on the derivation of $\mathcal{D}; \Gamma; \Sigma \vdash e : \tau$ and $C; \mathcal{D}; \Gamma; \Sigma \vdash e : \tau$. □

Lemma 15 (Consistency of Substitution over Type Derivation). *If $\Vdash \{\mathcal{D}; \Gamma; \Sigma \vdash e : \tau\}$, and $\theta \Vdash_{\text{cst}} \{\Gamma, \Sigma, \tau, \mathcal{D}\}$, then, $\theta \Vdash_{\text{cst}} \{\mathcal{D}; \Gamma; \Sigma \vdash e : \tau\}$.*

Proof. By induction on the derivation of $\mathcal{D}; \Gamma; \Sigma \vdash e : \tau$.

1. At any step of the derivation, let τ_s be a type used in the derivation such that $\tau_s \in \Gamma$, $\tau_s \in \Sigma$, and $\tau_s \in \tau$ do not (necessarily) hold. For example, τ_1 and τ_2 in T-Lambda, $\tau_a \rightarrow \tau_r$ in T-App, or the type obtained by any sub-derivation (for some sub-expression).

2. Through a suitable renaming of variables, and using Lemma 7, we can ensure $\text{dom}(\theta) \cap \text{ftv}(\tau_s) \subseteq \text{ftv}(\Gamma, \Sigma, \tau)$.

3. We want to show $\theta \Vdash_{\text{cst}} \tau_s$. Therefore, we proceed by induction on the structure of τ_s . The only interesting cases are $\tau_s = \alpha\downarrow\rho$ and $\tau_s = \varsigma\downarrow\rho$.

4. Case $\tau_s = \alpha\downarrow\rho$:

(a) By induction hypothesis, we have $\theta \Vdash_{\text{cst}} \rho$.

(b) If $\alpha \in \text{ftv}(\Gamma, \Sigma, \tau)$:

i. Due to the premise $\Vdash \{\mathcal{D}; \Gamma; \Sigma \vdash e : \tau\}$, and Definition 11, $\exists \alpha\downarrow\rho' \in \Gamma, \Sigma$, or τ such that $\rho \Vdash \rho'$.

ii. From the premise $\theta \Vdash_{\text{cst}} \{\Gamma, \Sigma, \tau, \mathcal{D}\}$, and Lemma 2 (weakening), we obtain $\theta \Vdash_{\text{cst}} \rho'$. From this, case (4.a) and Lemma 5 (aggregation), we obtain $\theta \Vdash_{\text{cst}} \{\rho, \rho'\}$.

iii. From cases (4.b.i and 4.b.ii), and Lemma 4, we obtain $\theta\langle\rho\rangle \Vdash \theta\langle\rho'\rangle$.

iv. From the premise $\theta \Vdash_{\text{cst}} \{\Gamma, \Sigma, \tau\}$, case (4.b.i), and Definition 12, we know that $\theta\langle\alpha\rangle = \beta$, for some β , or $\theta\langle\alpha\rangle = \rho''$, such that $\rho'' \Vdash \theta\langle\rho'\rangle$. Using case (4.b.iii), we can write this as $\theta\langle\rho'\rangle \Vdash \theta\langle\rho\rangle \Vdash \rho''$.

(c) If $\alpha \notin \text{ftv}(\Gamma, \Sigma, \tau)$, then, we know that $\alpha \notin \text{dom}(\theta)$, and therefore, $\theta\langle\alpha\rangle = \alpha$.

Therefore, in all cases, for some β and ρ'' , we have $\theta\langle\alpha\rangle = \beta$, or $\theta\langle\alpha\rangle = \rho''$ such that $\rho'' \Vdash \theta\langle\rho\rangle$.

5. Case $\tau_s = \varsigma\downarrow\rho$: Similar to the previous case, we conclude that for some ς' and ρ'' , we have $\theta\langle\varsigma\rangle = \varsigma'$, or $\theta\langle\varsigma\rangle = \rho''$ such that $\rho'' \Vdash \theta\langle\rho\rangle$.

6. From cases (4 and 5), and Definition 12, we conclude that $\theta \Vdash_{\text{cst}} \tau_s$.

7. By repeating the same argument over all types τ_s used in the derivation of $\mathcal{D}; \Gamma; \Sigma \vdash e : \tau$, we finally obtain $\theta \Vdash_{\text{cst}} \{\mathcal{D}; \Gamma; \Sigma \vdash e : \tau\}$.

□

Lemma 16 (Composition Consistency). *If:*

$\theta_1 \Vdash \{\bar{\omega}\}$, $\theta_2 \Vdash \{\bar{\omega}\}$, $\theta_1 \Vdash \{\theta_2\langle\bar{\omega}\rangle\}$, and $\text{dom}(\theta_2) \cap \text{ftv}(\theta_1)$,

Then, $\exists \theta'_2$ such that:

$\text{dom}(\theta_2) = \text{dom}(\theta'_2)$, $\theta'_2\langle\theta_1\langle\bar{\omega}\rangle\rangle = \theta_1\langle\theta_2\langle\bar{\omega}\rangle\rangle$, and $\theta'_2 \Vdash \{\theta_1\langle\bar{\omega}\rangle\}$.

Proof. By construction of θ'_2 . Let θ be an idempotent substitution equivalent to $\theta_1 \circ \theta_2$. θ'_2 can be chosen as a part of θ that contains substitutions only for $\text{dom}(\theta_2)$. □

Lemma 17 (Type Substitution). *If*

1. $C; \mathcal{D}; \Gamma; \Sigma \vdash e : \tau$
2. $\theta \Vdash_{\text{cst}} \{\mathcal{D}; \Gamma; \Sigma \vdash e : \tau\}$
3. $\theta\langle C \rangle \Vdash \theta\{\mathcal{D}; \Gamma; \Sigma \vdash e : \tau\}$

Then, $\theta\langle C \rangle; \theta\langle \mathcal{D} \rangle; \theta\langle \Gamma \rangle; \theta\langle \Sigma \rangle \vdash e : \theta\langle \tau \rangle$.

Proof. By induction on the derivation of $\mathcal{D}; \Gamma; \Sigma \vdash e : \tau$. We proceed by case analysis on the last step of the derivation. Let $C_s = \theta\langle C \rangle$.

1. Cases T-Unit, T-Bool, T-Hloc and T-Sloc are trivial.
2. Case T-Id:

(a) We have

- i. $\frac{\Gamma(x) = \forall \bar{\alpha}. \tau_x \setminus \mathcal{D}_x \quad \theta_x \Vdash \{\tau_x, \mathcal{D}_x\} \quad \text{dom}(\theta_x) = \{\bar{\alpha}\}}{\theta_x\langle \mathcal{D}_x \rangle; \Gamma; \Sigma \vdash x : \theta\langle \tau \rangle}$
- ii. $e = x$, $\tau = \theta_x\langle \tau_x \rangle$ and $\mathcal{D} = \theta_x\langle \mathcal{D}_x \rangle$
- iii. $\theta \Vdash_{\text{cst}} \{\mathcal{D}; \Gamma; \Sigma \vdash x : \theta_x\langle \tau_x \rangle\}$.
- iv. $C_s \Vdash \theta\{\mathcal{D}; \Gamma; \Sigma \vdash x : \theta_x\langle \tau_x \rangle\}$

(b) From premise (4), we have $\text{ftv}(\theta) \cap \{\bar{\alpha}\} = \emptyset$. Since $\text{dom}(\theta_x) = \{\bar{\alpha}\}$, we have $\text{dom}(\theta_x) \cap \text{ftv}(\theta) = \emptyset$.

(c) From cases (2.a.i and 2.b), we have $\theta\langle \Gamma \rangle(x) = \forall \bar{\alpha}. \theta\langle \tau_x \rangle$.

(d) i. From Definition 3, we have $\{\mathcal{D}; \Gamma; \Sigma \vdash x : \tau\} = \{\Gamma, \Sigma, \tau, \mathcal{D}\}$

ii. From case (2.a.iii and 2.d.i) and Lemma 2 (weakening), we have $\theta \Vdash_{\text{cst}} \{\tau_x, \mathcal{D}_x\}$. (Note: $\{\tau_x\} \subseteq \{\Gamma\}$).

Again using case (2.a.iv), we obtain $C_s \Vdash \theta\{\tau_x, \mathcal{D}_x\}$, which implies $\Vdash \theta\{\tau_x, \mathcal{D}_x\}$.

Now, using Definition 12, we obtain $\theta \Vdash \{\tau_x, \mathcal{D}_x\}$.

iii. From case (2.a.i), we have $\theta_x \Vdash \{\tau_x, \mathcal{D}_x\}$.

iv. Similar to case (2.d.ii), we obtain $\theta \Vdash \{\theta_x\langle \tau_x \rangle, \theta_x\langle \mathcal{D}_x \rangle\}$.

(e) From cases (2.d.ii, 2.d.iii, 2.d.iv, and 2.b), and Lemma 9, we conclude that $\exists \theta_m$ such that:

- i. $\text{dom}(\theta_m) = \text{dom}(\theta_x) = \{\bar{\alpha}\}$.
- ii. $\theta_m\langle \theta\langle \tau_x \rangle \rangle = \theta\langle \theta_x\langle \tau_x \rangle \rangle$. That is, $\theta_m\langle \theta\langle \tau_x \rangle \rangle = \theta\langle \tau \rangle$.
- iii. $\theta_m\langle \theta\langle \mathcal{D}_x \rangle \rangle = \theta\langle \theta_x\langle \mathcal{D}_x \rangle \rangle$. That is, $\theta_m\langle \theta\langle \mathcal{D}_x \rangle \rangle = \theta\langle \mathcal{D} \rangle$.
- iv. $\theta_m \Vdash \{\theta\langle \tau_x \rangle, \theta\langle \mathcal{D}_x \rangle\}$.

(f) From cases (2.c, 2.e.i and 2.e.iv), and the T-Id rule we obtain $\theta_m\langle \theta\langle \mathcal{D}_x \rangle \rangle; \theta\langle \Gamma \rangle; \theta\langle \Sigma \rangle \vdash x : \theta_m\langle \theta\langle \tau_x \rangle \rangle$.

- (g) From cases (2.f, 2.e.ii and 2.e.iii), we obtain $\theta\langle\mathcal{D}\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash x : \theta\langle\tau\rangle$.
- (h) From Definition 3, we have: $\{\theta\langle\mathcal{D}\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash x : \theta\langle\tau\rangle\} = \{\theta\langle\Gamma\rangle, \theta\langle\Sigma\rangle, \theta\langle\tau\rangle, \theta\langle\mathcal{D}\rangle\}$. Clearly, $\{\theta\langle\Gamma\rangle, \theta\langle\Sigma\rangle, \theta\langle\tau\rangle, \theta\langle\mathcal{D}\rangle\} \subseteq \{\theta\{\Gamma, \Sigma, \tau, \mathcal{D}\}\}$.
- (i) From cases (2.a.iv and 2.d.i), we have $C_s \Vdash \theta\{\Gamma, \Sigma, \tau, \mathcal{D}\}$. Now, from Lemma 1 (property-3), we obtain $C_s \Vdash \{\theta\{\Gamma, \Sigma, \tau, \mathcal{D}\}\}$.
- (j) Now, from cases (3.i and 3.h) and Lemma 2 (weakening), we obtain $C_s \Vdash \{\theta\langle\Gamma\rangle, \theta\langle\Sigma\rangle, \theta\langle\tau\rangle, \theta\langle\mathcal{D}\rangle\}$ and therefore $C_s \Vdash \{\theta\langle\mathcal{D}\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash x : \theta\langle\tau\rangle\}$
- (k) Finally, from cases (2.g and 2.j) and Definition 4, we conclude that $\theta\langle C\rangle; \theta\langle\mathcal{D}\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash x : \theta\langle\tau\rangle$.

3. Case T-Lambda: Follows from induction hypothesis, using Lemma 4 and the T-Lambda rule.

4. Case T-App:

(a) In this case, we have:

$$i. \frac{\mathcal{D}_1; \Gamma; \Sigma \vdash e_1 : \tau_1 \quad \tau_1 \leq: \tau_a \rightarrow \tau_r \quad \mathcal{D}_2; \Gamma; \Sigma \vdash e_2 \leq: \tau_2 \quad \tau_2 \leq: \nabla(\tau_a) \quad \Delta(\tau_r) \leq: \tau}{\Gamma; \Sigma \vdash e_1 e_2 : \tau}$$

- ii. $\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2$
- iii. $\theta \stackrel{\text{cst}}{\Vdash} \{\mathcal{D}; \Gamma; \Sigma \vdash e_1 e_2 : \tau\}$
- iv. $\theta\langle C\rangle \Vdash \theta\{\mathcal{D}; \Gamma; \Sigma \vdash e_1 e_2 : \tau\}$

(b) By induction hypothesis (by using case (4.a.i), weakening on cases (4.a.iii and 4.a.iv), and premise (4)), we have: $C_s; \theta\langle\mathcal{D}_1\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e_1 : \theta\langle\tau_1\rangle$ and $C_s; \theta\langle\mathcal{D}_2\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e_2 : \theta\langle\tau_2\rangle$. That is,

- i. $\theta\langle\mathcal{D}_1\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e_1 : \theta\langle\tau_1\rangle$
- ii. $C_s \Vdash \{\theta\langle\mathcal{D}_1\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e_1 : \theta\langle\tau_1\rangle\}$
- iii. $\theta\langle\mathcal{D}_2\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e_2 : \theta\langle\tau_2\rangle$
- iv. $C_s \Vdash \{\theta\langle\mathcal{D}_2\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e_2 : \theta\langle\tau_2\rangle\}$

(c) From case (4.a.i), we have $\tau_1 \leq: \tau_a \rightarrow \tau_r$. Using weakening on case (4.a.iii) and Lemma 6, we obtain $\theta\langle\tau_1\rangle \leq: \nabla(\theta\langle\tau_a \rightarrow \tau_r\rangle)$, which is equivalent to $\theta\langle\tau_1\rangle \leq: \theta\langle\tau_a\rangle \rightarrow \theta\langle\tau_r\rangle$.

(d) Similarly, we obtain $\theta\langle\tau_2\rangle \leq: \nabla(\theta\langle\nabla(\tau_a)\rangle)$. Using weakening on case (4.a.iii) and Lemma 3, we have $\nabla(\theta\langle\nabla(\tau_a)\rangle) = \nabla(\theta\langle\tau_a\rangle)$. Therefore, $\theta\langle\tau_2\rangle \leq: \nabla(\theta\langle\tau_a\rangle)$.

(e) Similarly, we obtain $\Delta(\theta\langle\tau_r\rangle) \leq: \theta\langle\tau\rangle$.

(f) From cases (4.b.i, 4.b.iii, 4.c, 4.d and 4.e) and using T-App rule, we conclude that $\theta\langle\mathcal{D}_1\rangle \cup \theta\langle\mathcal{D}_2\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e_1 e_2 : \theta\langle\tau\rangle$ That is, $\theta\langle\mathcal{D}\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e_1 e_2 : \theta\langle\tau\rangle$.

(g) Using weakening on case (4.a.iv), we obtain $C_s \Vdash \{\tau_a \rightarrow \tau_r, \tau\}$. Using this and cases (4.b.ii and 4.b.iv), and Lemma 5 (aggregation), along with Definition 3 (constraint collection), we conclude that $C_s \Vdash \{\theta\langle\mathcal{D}\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e_1 e_2 : \theta\langle\tau\rangle\}$

(h) From cases (4.f and 4.g), we obtain $\theta\langle C\rangle; \theta\langle\mathcal{D}\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e_1 e_2 : \theta\langle\tau\rangle$

5. Case T-Let-MP: In this case, we have

$$\frac{\mathcal{D}_1; \Gamma; \Sigma \vdash v \leq: \tau_1 \quad \tau_x \leq: \tau_1 \quad \mathcal{D}_x = \mathcal{D}_1 \cup \{\star_x^x(\tau_x)\} \quad \{\bar{\alpha}\} = \text{ftv}(\tau_x, \mathcal{D}_x) \setminus \text{ftv}(\Gamma, \Sigma) \quad \stackrel{\text{ncw}}{\Vdash} \bar{\beta}}{\mathcal{D}_2; \Gamma, x \mapsto \forall \bar{\alpha}. \tau_x \setminus \mathcal{D}_x; \Sigma \vdash e : \tau_2} \quad \frac{}{\mathcal{D}[\bar{\beta}/\bar{\alpha}] \cup \mathcal{D}_2; \Gamma; \Sigma \vdash (\text{let}^x x = v \text{ in } e) : \tau_2}$$

Proof is similar to T-App case. Without loss of generality, we can assume that $\text{ftv}(\theta) \cap \{\bar{\alpha}\}$ since type schemes are equal under α -renaming of generalized variables.

6. Other cases are similar.

□

Lemma 18 (Value Substitution). *If*

1. $C; \mathcal{D}; \Gamma, x : \forall \bar{\alpha}. \tau_v \setminus \mathcal{D}_v; \Sigma \vdash e : \tau$
2. $C; \mathcal{D}_v; \Gamma; \Sigma \vdash v : \tau_v$
3. $\text{Immut}(\tau_v)$

Then, $\exists C' \supseteq C$ such that $C'; \mathcal{D}; \Gamma; \Sigma \vdash e[v/x] : \tau$

Proof. By induction on the type derivation of $C; \mathcal{D}; \Gamma, x : \forall \bar{\alpha}. \tau_v; \Sigma \vdash e : \tau$ We proceed by case analysis on the final step of the derivation.

1. Case T-Id: We have $e = y$, where $y \in \Gamma, x, \sigma$.

(a) If $e = y \neq x$, then, $y[v/x] = y$. and the desired result $C; \mathcal{D}; \Gamma; \Sigma \vdash y : \tau$ is immediate from the assumption $C; \mathcal{D}; \Gamma, x : \forall \bar{\alpha}. \tau_v; \Sigma \vdash e : \tau$.

(b) If $e = x = y$, we have $y[v/x] = v$, $\mathcal{D} = \theta\langle \mathcal{D}_v \rangle$ and $\tau = \theta\langle \tau_v \rangle$. Let $\Gamma' = \Gamma, y : \forall \bar{\alpha}. \tau_v$. In this case, we have

- i.
$$\frac{\Gamma'(y) = \forall \bar{\alpha}. \tau_v \setminus \mathcal{D}_v \quad \theta \Vdash \{\tau_v, \mathcal{D}_v\} \quad \text{dom}(\theta) = \{\bar{\alpha}\}}{\theta\langle \mathcal{D}_v \rangle; \Gamma'; \Sigma \vdash y : \theta\langle \tau_v \rangle}$$
- ii. $C \Vdash \{\theta\langle \mathcal{D}_v \rangle; \Gamma'; \Sigma \vdash y : \theta\langle \tau_v \rangle\}$
- iii. $C; \mathcal{D}_v; \Gamma; \Sigma \vdash v : \tau_v$. That is,
 - A. $\mathcal{D}_v; \Gamma; \Sigma \vdash v : \tau_v$
 - B. $C \Vdash \{\mathcal{D}_v; \Gamma; \Sigma \vdash v : \tau_v\}$

We need to show that $\exists C' \supseteq C$ such that $C'; \theta\langle \mathcal{D}_v \rangle; \Gamma; \Sigma \vdash v : \theta\langle \tau_v \rangle$.

(c) We know that

- i. $\{\bar{\alpha}\} \circ \text{ftv}(\Gamma)$. That is, $\theta\langle \Gamma \rangle = \Gamma$.
- ii. $\{\bar{\alpha}\} \circ \text{ftv}(\Sigma)$. That is, $\theta\langle \Sigma \rangle = \Sigma$.
- iii. Without loss of generality, we can assume that $\text{ftv}(\text{range}(\theta))$ consists of variables in $\text{ftv}(\Gamma, \Sigma, \tau_v, \mathcal{D}_v)$, or fresh type variables. This property can always be made to hold by suitable renaming of variables and using Lemma 7.
- iv. Without loss of generality, we can assume that $\{\bar{\alpha}\} \subseteq \text{ftv}(\tau_v, \mathcal{D}_v)$, since any $\{\bar{\beta}\} \subseteq \{\bar{\alpha}\}$ such that $\bar{\beta} \notin \text{ftv}(\tau_v, \mathcal{D}_v)$ can be ignored.

(d) From case (1.b.i and 1.b.ii) we can say that $\exists C_{big} \supseteq C$ such that $C_{big} \Vdash \theta\{\tau_v, \mathcal{D}_v\}$. This is because:

- i. From case (1.b.i) we have $\theta \Vdash \{\tau_v, \mathcal{D}_v\}$. From this, using Lemma 2 (weakening), we obtain $\Vdash \theta\{\tau_v, \mathcal{D}_v\}$.
- ii. From case (1.b.ii) and Lemma 2 (weakening), we obtain $C \Vdash \{\theta\langle \tau_v \rangle, \theta\langle \mathcal{D}_v \rangle\}$.
- iii. Since $\{\theta\langle \tau_v \rangle, \theta\langle \mathcal{D}_v \rangle\} \subseteq \theta\{\tau_v, \mathcal{D}_v\}$, using case (1.d.i) and Lemma 1 (property-4), we conclude that $\exists C_{big} \supseteq C$ such that $C_{big} \Vdash \theta\{\tau_v, \mathcal{D}_v\}$.

(e) For all $\beta \simeq \rho$ or $\beta \cong \rho \in C$, where $\beta \notin \{\bar{\alpha}\}$, we must have $\text{ftv}(\rho) \circ \{\bar{\alpha}\}$. Otherwise, due to case (1.d.iii), there exists $\beta \simeq \theta\langle \rho \rangle$ or $\beta \cong \theta\langle \rho \rangle \in C_{big}$ such that $\theta\langle \rho \rangle \neq \rho$, which violates the restriction $\mathfrak{R}(C_{big})$.

(f) We can write $C = C_e \cup C_t \cup C_a \cup C_r$ where:

- i. $\text{dom}(C_e) = \text{ftv}(\Gamma, \Sigma)$
- ii. $\text{dom}(C_t) = \text{ftv}(\tau_v, \mathcal{D}_v) \setminus \text{ftv}(\Gamma, \Sigma) \setminus \{\bar{\alpha}\}$
- iii. $\text{dom}(C_a) = \{\bar{\alpha}\}$
- iv. $\text{dom}(C_e) \circ \text{dom}(C_t) \circ \text{dom}(C_a) \circ \text{dom}(C_r)$

(g) i. Let $C_{et} = C_e \cup C_t$ and $C_{ar} = C_a \cup C_r$

ii. From case(1.f.i), weakening on (1.b.ii), and Lemma 1 (property-8), we have $\text{ftv}(\text{range}(C_e)) \subseteq \text{ftv}(\Gamma, \Sigma)$.

- iii. Similarly, using case (1.e), we conclude that $\text{ftv}(\text{range}(C_t)) \subseteq \text{ftv}(\Gamma, \Sigma, \tau_v, \mathcal{D}_v) \setminus \{\bar{\alpha}\}$.
- iv. Therefore, we have $\text{ftv}(C_{et}) \cap \text{dom}(C_{at})$.

(h) Let

- i. $\{\bar{\beta}\} = \text{dom}(C_r)$.
- ii. $\theta_a = [\bar{\alpha} \mapsto \bar{\gamma}]$, where $\bar{\gamma}$ are new type variables.
- iii. $\theta_b = [\bar{\beta} \mapsto \bar{\delta}]$, where $\bar{\delta}$ are new type variables.
- iv. $\theta_{ab} = \theta_a \circ \theta_b$
- v. $C_{ab} = C[\bar{\gamma}/\bar{\alpha}][\bar{\delta}/\bar{\beta}]$.

Due to case (1.g.iv), $C_{ab} = C_{et} \cup C'_{ar}$, where $C'_{ar} = C_{ar}[\bar{\gamma}/\bar{\alpha}][\bar{\delta}/\bar{\beta}]$.

- vi. $\tau_{va} = \theta_a \langle \tau_v \rangle$, and $\mathcal{D}_{va} = \theta_a \langle \mathcal{D}_v \rangle$
- vii. $\theta' = [\bar{\gamma} \mapsto \tau_s]$, where $\theta = [\bar{\alpha} \mapsto \tau_s]$

Note that for all τ_s above, $\theta_{ab} \langle \tau_s \rangle = \tau_s$ due to case (1.c.iii). Similarly, due to cases (3.c.i, 3.c.ii and 3.h.vii) we have $\theta' \langle \Gamma \rangle = \Gamma$, $\theta' \langle \Sigma \rangle = \Sigma$, $\theta' \langle \tau_{va} \rangle = \theta \langle \tau_v \rangle = \tau$, and $\theta' \langle \mathcal{D}_{va} \rangle = \theta \langle \mathcal{D}_v \rangle = \mathcal{D}$.

- (i) i. Let $C' = C \cup C_{ab}$
- ii. This can be written as:
 $C' = C_{et} \cup C_{ar} \cup C_{et} \cup C'_{ar} = C \cup C'_{ar}$.
- iii. Since $\text{dom}(C'_{ar})$ consists of new type variables, we have $\mathfrak{R}(C')$.
- iv. Further, since $\theta' \langle C \rangle = C$, we have $\theta' \langle C' \rangle = C \cup \theta' \langle C'_{ar} \rangle$.
- (j) From cases (1.b.iii and 1.h) using Lemma 7 (renaming), we obtain $C_{ab}; \theta_{ab} \langle \mathcal{D}_v \rangle; \theta_{ab} \langle \Gamma \rangle; \theta_{ab} \langle \Sigma \rangle \vdash v : \theta_{ab} \langle \tau_v \rangle$.
From cases (1.f and 1.h), we can write this as $C_{ab}; \mathcal{D}_{va}; \Gamma; \Sigma \vdash v : \tau_{va}$.
- (k) From cases (1.j, 1.i.i and 1.i.iii) and Lemma 2 (weakening) (property-5), we obtain $C'; \mathcal{D}_{va}; \Gamma; \Sigma \vdash v : \tau_{va}$.
- (l) Similarly, by applying variable renaming on case (1.b.i and 1.b.ii), we obtain

$$\Gamma'(y) = \forall \bar{\gamma}. \tau_{va} \setminus \mathcal{D}_{va} \quad \theta \Vdash \{\tau_{va}, \mathcal{D}_{va}\}$$

- i. $\frac{\text{dom}(\theta') = \{\bar{\gamma}\}}{\theta' \langle \mathcal{D}_{va} \rangle; \Gamma'; \Sigma \vdash y : \theta' \langle \tau_{va} \rangle}$
- ii. $C' \Vdash \{\theta' \langle \mathcal{D}_{va} \rangle; \Gamma'; \Sigma \vdash y : \theta' \langle \tau_{va} \rangle\}$
- (m) From case (1.l.i), we have $\theta' \Vdash \{\tau_{va}, \mathcal{D}_{va}\}$, which implies by weakening, $\theta' \Vdash_{\text{cst}} \{\tau_{va}, \mathcal{D}_{va}\}$, and further due to case (1.h.vii), $\theta' \Vdash_{\text{cst}} \{\Gamma, \Sigma, \tau_{va}, \mathcal{D}_{va}\}$. Case (1.h) implies $\Vdash \{\mathcal{D}_{va}; \Gamma; \Sigma \vdash v : \tau_{va}\}$. Using these facts with Lemma 8, we conclude that $\theta' \Vdash_{\text{cst}} \{\mathcal{D}_{va}; \Gamma; \Sigma \vdash v : \tau_{va}\}$.
- (n) i. From case (1.l.i), similar to case (1.d), we conclude that $\exists C'' \supseteq C' \mid C'' \Vdash \theta' \{\tau_{va}, \mathcal{D}_{va}\}$. Due to idempotence of the substitution θ' , we can write this as $\theta' \langle C'' \rangle \Vdash \theta' \{\tau_{va}, \mathcal{D}_{va}\}$.
- ii. By weakening on case (1.k), we obtain $C''; \mathcal{D}_{va}; \Gamma; \Sigma \vdash v : \tau_{va}$. This implies $C'' \Vdash \{\mathcal{D}_{va}; \Gamma; \Sigma \vdash v : \tau_{va}\}$
- iii. Evidently, $\{\tau_{va}, \mathcal{D}_{va}\} \subseteq \{\mathcal{D}_{va}; \Gamma; \Sigma \vdash v : \tau_{va}\}$
- iv. From cases (1.c.iv and 1.h.ii) we conclude that $\text{dom}(\theta') \subseteq \text{ftv}(\tau_{va}, \mathcal{D}_{va})$. This implies $\text{dom}(\theta') \cap \text{ftv}(\{\mathcal{D}_{va}; \Gamma; \Sigma \vdash v : \tau_{va}\}) \subseteq \text{ftv}(\{\tau_{va}, \mathcal{D}_{va}\})$
- v. From cases (1.n.ii, 1.m, 1.n.iii, 1.n.i and 1.n.iv) and Lemma 7 (strengthening), we conclude that $\theta' \langle C'' \rangle \Vdash \theta' \{\mathcal{D}_{va}; \Gamma; \Sigma \vdash v : \tau_{va}\}$.
- (o) From cases (1.n.ii, 1.m and 1.n.v) and Lemma 10 (type substitution), we conclude that $\theta' \langle C'' \rangle; \theta' \langle \mathcal{D}_{va} \rangle; \theta' \langle \Gamma \rangle; \theta' \langle \Sigma \rangle \vdash v : \theta' \langle \tau_{va} \rangle$. From case (1.h.vii), this can be written as $\theta' \langle C'' \rangle; \mathcal{D}; \Gamma; \Sigma \vdash v : \tau$.
- (p) Now, $C'' = C \cup C_{rest}$. From case (1.i.iv), we have $\theta' \langle C'' \rangle = C \cup \theta' \langle C_{rest} \rangle$. Therefore, $\theta' \langle C'' \rangle \supseteq C$.

2. Case T-Lambda: We have

$$\frac{\mathcal{D}; \Gamma, x \mapsto \forall \bar{\alpha}. \tau_v, y \mapsto \tau_1; \Sigma \vdash e : \tau_2 \quad \tau_1 \stackrel{\nabla}{=} \tau'_1 \quad \tau_2 \stackrel{\nabla}{=} \tau'_2}{\mathcal{D}; \Gamma, x \mapsto \forall \bar{\alpha}. \tau_v; \Sigma \vdash \lambda x. e : \tau'_1 \rightarrow \tau'_2}$$

We can assume that $x \neq y$. Now, the result follows from the induction hypothesis, and the T-Lambda rule.

3. Case T-App:

(a) In this case, writing $\Gamma' = \Gamma, x \mapsto \forall \bar{\alpha}. \tau_v$, we have:

$$\text{i. } \frac{\mathcal{D}_1; \Gamma'; \Sigma \vdash e_1 : \tau_1 \quad \tau_1 \preceq: \tau_a \rightarrow \tau_r \quad \mathcal{D}_2; \Gamma'; \Sigma \vdash e_2 : \tau_2 \quad \tau_2 \preceq: \nabla(\tau_a) \quad \Delta(\tau_r) \preceq: \tau}{\mathcal{D}; \Gamma'; \Sigma \vdash e_1 e_2 : \tau}$$

ii. $C \Vdash \{\mathcal{D}; \Gamma'; \Sigma \vdash e_1 e_2 : \tau\}$

iii. $\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2$

(b) From case (3.i) and weakening on case (3.ii), we have $C; \mathcal{D}_1; \Gamma'; \Sigma \vdash e_1 : \tau_1$. Now, by induction hypothesis, we know that $\exists C' \supseteq C \mid C'; \mathcal{D}_1; \Gamma; \Sigma \vdash e_1[v/x] : \tau_1$.

(c) Similarly, we have $C; \mathcal{D}_2; \Gamma'; \Sigma \vdash e_2 : \tau_2$. By Lemma 2 (weakening), we obtain $C'; \mathcal{D}_2; \Gamma'; \Sigma \vdash e_2 : \tau_2$. Again, by induction hypothesis, we know that $\exists C'' \supseteq C' \mid C''; \mathcal{D}_2; \Gamma; \Sigma \vdash e_2[v/x] : \tau_2$.

(d) From case (3.b) and Lemma 2 (weakening), we obtain $C''; \mathcal{D}_1; \Gamma; \Sigma \vdash e_1[v/x] : \tau_1$.

(e) Again by weakening wrt case (3.a.ii), we have $C'' \Vdash \{\tau_a, \tau_r, \tau\}$.

(f) From cases (3.c and 3.d) and copy-coersion relations in case (3.a.i), using the T-App rule we obtain $\mathcal{D}; \Gamma; \Sigma \vdash e_1[v/x] e_2[v/x] : \tau$. Using cases (3.c, 3.d and 3.e) and Definition 3, we conclude that $C''; \mathcal{D}; \Gamma; \Sigma \vdash e_1[v/x] e_2[v/x] : \tau$.

4. T-Set case is to T-App. Premise (3) guarantees that the substitution cannot happen on the LHS of an assignment except within a dereferenced expression.

5. Other cases are similar. □

Lemma 19 (Corollary to Value Substitution). *If*

$$1. C; \mathcal{D}; \Gamma, x : \forall \bar{\alpha}. \tau_v \setminus \mathcal{D}_v \cup \star_x^\forall(\tau_v); \Sigma \vdash e : \tau$$

$$2. C; \mathcal{D}_v; \Gamma; \Sigma \vdash v : \tau_v$$

$$3. \Vdash \mathcal{D}$$

Then, $\exists C' \supseteq C$ such that $C'; \mathcal{D}; \Gamma; \Sigma \vdash e[v/x] : \tau$

Proof. Similar to Lemma 11. The premise $\Vdash \mathcal{D}$ guarantees that $\forall \star_x^\forall(\tau_v) \in \mathcal{D}$, $\text{Immut}(\tau_v)$, which ensures that all instantiations of x are immutable. □

Lemma 20 (Location Substitution). *If $C; \mathcal{D}; \Gamma, x : \tau_x; \Sigma \vdash e : \tau$, then $C; \mathcal{D}; \Gamma; \Sigma, l \mapsto \tau_x \vdash e[l/x] : \tau$.*

Proof. By induction on the type derivation of $\vdash \Gamma, x : \tau; \Sigma$, similar to Lemma 11. □

Lemma 21 (Corollary to Location Substitution). *If $C; \mathcal{D}; \Gamma, x : \forall \bar{\alpha}. \tau_v \setminus \mathcal{D}_v \cup \star_x^\forall(\tau_v); \Sigma \vdash e : \tau$ and $\Vdash \mathcal{D}$, then $C; \mathcal{D}; \Gamma; \Sigma, l \mapsto \tau_x \vdash e[l/x] : \tau$.*

Proof. Similar to Lemma 13. □

Lemma 22 (Stack and Heap Assignment Safety). *1. If $C; \mathcal{D}; \Gamma; \Sigma \vdash_s H, \ell \mapsto v + S$, and $C; \mathcal{D}; \Gamma; \Sigma \vdash_s v' : \tau$, and $\Sigma(\ell) \stackrel{\forall}{=} \tau$, then $C; \mathcal{D}; \Gamma; \Sigma \vdash_s H, \ell \mapsto v' + S$.*

2. If $C; \mathcal{D}; \Gamma; \Sigma \vdash_s H + S, l \mapsto v$, and $C; \mathcal{D}; \Gamma; \Sigma \vdash_s v' : \tau$, and $\Sigma(l) \stackrel{\forall}{=} \tau$, then $C; \mathcal{D}; \Gamma; \Sigma \vdash_s H + S, l \mapsto v'$.

Proof. Immediate from the definition of stack and heap typing. □

Theorem 2 (Subject Reduction). *For any canonical expression e , if $C; \mathcal{D}; \Gamma; \Sigma \vdash_s e : \tau$, $C; \mathcal{D}; \Gamma; \Sigma \vdash_s H + S$ and $\models \mathcal{D}$, then,*

1. *If $S; H; e \Rightarrow S'; H'; e'$, then, $\exists \Sigma' \supseteq \Sigma$ and $C' \supseteq C$ such that $C'; \mathcal{D}; \Gamma; \Sigma' \vdash_s e' : \tau$ and $C'; \mathcal{D}; \Gamma; \Sigma' \vdash_s H' + S'$.*
2. *If $S; H; e \Rightarrow S'; H'; e'$, then, $\exists \Sigma' \supseteq \Sigma$ and $C' \supseteq C$ such that $C'; \mathcal{D}; \Gamma; \Sigma' \vdash_s e' : \tau'$ and $C'; \mathcal{D}; \Gamma; \Sigma' \vdash_s H' + S'$ and $\tau \stackrel{\nabla}{=} \tau'$.*

Proof. By induction on the derivation of $C; \mathcal{D}; \Gamma; \Sigma \vdash e : \tau$. We proceed by the case analysis of the final step.

1. Case T-Id, T-Bool, T-Hloc, T-Lambda cannot happen.

2. Case T-Sloc: $e = 1$. We have, for some $\mathcal{D}' \subseteq \mathcal{D}$,

$$\frac{\Sigma(1) = \tau}{\mathcal{D}'; \Gamma; \Sigma \vdash 1 : \tau}$$

(a) \Rightarrow E-Rval:

$$\frac{S(1) = v}{S; H; 1 \Rightarrow S; H; v}$$

i. $e' = S(1) = v$, $H' = H$, and $S' = S$

ii. From $C; \mathcal{D}; \Gamma; \Sigma \vdash_s H + S$ and Definition 5 (Stack and Heap Typing), we have $C; \mathcal{D}; \Gamma; \Sigma \vdash_s S(1) : \tau' \mid S(1) \stackrel{\nabla}{=} \tau'$. That is, $C; \mathcal{D}; \Gamma; \Sigma \vdash_s v : \tau'$, where $\tau \stackrel{\nabla}{=} \tau'$.

3. Case T-App: $e = e_1 e_2$. We have for some $\mathcal{D}_1 \subseteq \mathcal{D}$ and $\mathcal{D}_2 \subseteq \mathcal{D}$,

$$\frac{\mathcal{D}_1; \Gamma; \Sigma \vdash e_1 : \tau_1 \quad \tau_1 \triangleleft: \tau_a \rightarrow \tau_r \quad \mathcal{D}_2; \Gamma; \Sigma \vdash e_2 : \tau_2 \quad \tau_2 \triangleleft: \nabla(\tau_a) \quad \Delta(\tau_r) \triangleleft: \tau}{\mathcal{D}_1 \cup \mathcal{D}_2; \Gamma; \Sigma \vdash e_1 e_2 : \tau}$$

(a) \Rightarrow E-# (1):

$$\frac{S; H; e_1 \Rightarrow S'; H'; e'_1}{S; H; e_1 e_2 \Rightarrow S'; H'; e'_1 e_2}$$

i. By using Lemma 2 (weakening) on the assumptions of T-App rule, we obtain $C; \mathcal{D}; \Gamma; \Sigma \vdash_s e_1 : \tau_1$. By induction hypothesis, we conclude that $\exists \Sigma' \supseteq \Sigma$ and $C' \supseteq C$ such that $C'; \mathcal{D}; \Gamma; \Sigma' \vdash_s e_1 : \tau'_1$ and $C'; \mathcal{D}; \Gamma; \Sigma' \vdash_s H' + S'$ and $\tau_1 \stackrel{\nabla}{=} \tau'_1$.

ii. From case (3), we have $\tau_1 \triangleleft: \tau_a \rightarrow \tau_r$. From case (3.a.i) we have $\tau_1 \stackrel{\nabla}{=} \tau'_1$. Since $\tau_a \rightarrow \tau_r = \nabla(\tau_a \rightarrow \tau_r)$, using Lemma 5, we conclude that $\tau'_1 \triangleleft: \tau_a \rightarrow \tau_r$.

iii. Using weakening lemmas Lemma 4 and Lemma 2, we obtain $C'; \mathcal{D}; \Gamma; \Sigma' \vdash_s e_2 : \tau_2$.

iv. From case (3), we already have $\tau_2 \triangleleft: \nabla(\tau_a)$ and $\Delta(\tau_r) \triangleleft: \tau$. Clearly, these types are consistent with respect to the weakened constraints set C' .

v. Now, using the T-App rule and Definition 3, we obtain $C'; \mathcal{D}; \Gamma; \Sigma' \vdash_s e'_1 e_2 : \tau$.

(b) \Rightarrow E-# (2):

$$\frac{S; H; e_2 \Rightarrow S'; H'; e'_2}{S; H; v_1 e_2 \Rightarrow S'; H'; v_1 e'_2}$$

Similar to E-# (1).

(c) \Rightarrow E-App:

$$\frac{1 \notin \text{dom}(S)}{S; H; \lambda x. e_x v \Rightarrow S, 1 \mapsto v; H; e_x[v/x]}$$

i. Re-writing case (3) for $e_1 = \lambda x. e_x$ and $e_2 = v$ and using Lemma 1 (inversion of typing), we have, for some $\mathcal{D}_1 \subseteq \mathcal{D}$ and $\mathcal{D}_2 \subseteq \mathcal{D}$,

$$\frac{\frac{\mathcal{D}_1, \Gamma, x \mapsto \tau'_a; \Sigma \vdash e_x : \tau'_r}{\tau_a \stackrel{\nabla}{=} \tau'_a \quad \tau_r \stackrel{\nabla}{=} \tau'_r} \quad \mathcal{D}_2; \Gamma; \Sigma \vdash v : \tau_2}{\mathcal{D}_1; \Gamma; \Sigma \vdash \lambda x. e_x : \tau_a \rightarrow \tau_r} \quad \frac{\tau_1 \preceq: \tau_a \rightarrow \tau_r \quad \tau_2 \preceq: \nabla(\tau_a) \quad \Delta(\tau_r) \preceq: \tau}{\mathcal{D}_1 \cup \mathcal{D}_2; \Gamma; \Sigma \vdash \lambda x. e_x v : \tau}$$

We also have $C \Vdash \{\mathcal{D}_1 \cup \mathcal{D}_2; \Gamma; \Sigma \vdash \lambda x. e_x v : \tau\}$.

- ii. From case (3.c.i), using weakening, we obtain $C; \mathcal{D}_1; \Gamma, x \mapsto \tau'_a; \Sigma \vdash e_x : \tau'_r$
- iii. Using case (3.c.ii) and Lemma 13 (location substitution), we obtain $C; \mathcal{D}_1; \Gamma; \Sigma, l \mapsto \tau'_a \vdash e_x[l/x] : \tau'_r$ and therefore, $C; \mathcal{D}; \Gamma; \Sigma, l \mapsto \tau'_a \vdash e_x[l/x] : \tau'_r$.
- iv. From $\Delta(\tau_r) \preceq: \tau$, using Lemma 5, we conclude that $\tau_r \stackrel{\nabla}{=} \tau$. We already have $\tau_r \stackrel{\nabla}{=} \tau'_r$. Therefore, we conclude that $\tau'_r \stackrel{\nabla}{=} \tau$.
- v. We have $C; \mathcal{D}; \Gamma; \Sigma \vdash v : \tau_2$. From $\tau_2 \preceq: \nabla(\tau_a)$, we obtain $\tau_a \stackrel{\nabla}{=} \tau_2$. From case (3.c.i), we have $\tau_a \stackrel{\nabla}{=} \tau'_a$. Therefore, we obtain $\tau'_a \stackrel{\nabla}{=} \tau_2$. Using these facts along with the premise $C; \mathcal{D}; \Gamma; \Sigma \vdash H + S$ and Definition 5, we conclude that $C; \mathcal{D}; \Gamma; \Sigma, l \mapsto \tau'_a \vdash H + S, l \mapsto v$.

4. Case T-Set: $e = e_1 := e_2$

$$\frac{\mathcal{D}_1; \Gamma; \Sigma \vdash l : \tau_1 \quad \tau_1 \preceq: \Psi \rho}{\mathcal{D}_2; \Gamma; \Sigma \vdash e : \tau_2 \quad \tau_2 \preceq: \rho} \quad \frac{}{\mathcal{D}_1 \cup \mathcal{D}_2; \Gamma; \Sigma \vdash l := e : \text{unit}}$$

Cases E-L# and E-# follow from induction hypothesis, similar to the T-App case. Cases E-:=Stack, E-:=Heap, E-:=S.p and E-:=H.p follow from Lemma 15 (stack and heap assignment safety).

5. Case T-Let-M: $e = \text{let}^\psi x = e_1 \text{ in } e_2$

$$\frac{\mathcal{D}_1; \Gamma; \Sigma \vdash e_1 : \tau_1 \quad \tau_1 \preceq: \tau' \quad \tau \preceq: \tau'}{\mathcal{D}_2, \Gamma, x \mapsto \tau; \Sigma \vdash e_2 : \tau_2} \quad \frac{}{\mathcal{D}_1 \cup \mathcal{D}_2; \Gamma; \Sigma \vdash (\text{let}^\psi x = e_1 \text{ in } e_2) : \tau_2}$$

The case E-# follows from induction hypothesis using the T-Let-M rule. The case E-Let-M follows from Lemma 13 (location substitution) similar to T-App.

6. Case T-Let-MP: $e = \text{let}^\forall x = v \text{ in } e$

$$\frac{\mathcal{D}_1; \Gamma; \Sigma \vdash v : \tau_v \quad \tau_v \preceq: \tau_1 \quad \tau \preceq: \tau_1}{\mathcal{D}' = \mathcal{D}_1 \cup \{\star_x^\varkappa(\tau)\} \quad \{\bar{\alpha}\} = \text{ftv}(\tau, \mathcal{D}') \setminus \text{ftv}(\Gamma, \Sigma)} \quad \frac{\mathcal{D}_2, \Gamma, x \mapsto \forall \bar{\alpha}. \tau \setminus \mathcal{D}'; \Sigma \vdash e : \tau'}{\mathcal{D}[\beta/\bar{\alpha}] \cup \mathcal{D}_2; \Gamma; \Sigma \vdash (\text{let}^\varkappa x = v \text{ in } e) : \tau_2}$$

The case E-# follows from induction hypothesis using the T-Let-P rule. This is because v can only take E-# steps that only have leaf derivations of E-Rval step E-Rval steps, which does not change the constraint set \mathcal{D}_1 in the type derivation. If $\varkappa = \forall$, the case E-Let-P follows from Lemma 12 (value substitution). Otherwise, if $\varkappa = \psi$, the case E-Let-M follows from Lemma 14 (location substitution). The case $\varkappa = \kappa$ cannot happen since e is a canonical expression. □

Theorem 3 (Preservation). *For any canonical expression e , if $\mathcal{D}; \Gamma; \Sigma \vdash_* e : \tau$, $\mathcal{D}; \Gamma; \Sigma \vdash_* H + S$ and $\models \mathcal{D}$ then,*

1. *If $S; H; e \Rightarrow S'; H'; e'$, then, $\exists \Sigma' \supseteq \Sigma$ such that $\mathcal{D}; \Gamma; \Sigma' \vdash_* e' : \tau$ and $\mathcal{D}; \Gamma; \Sigma' \vdash_* H' + S'$.*
2. *If $S; H; e \Rightarrow S'; H'; e'$, then, $\exists \Sigma' \supseteq \Sigma$ such that $\mathcal{D}; \Gamma; \Sigma' \vdash_* e' : \tau'$ and $\mathcal{D}; \Gamma; \Sigma' \vdash_* H' + S'$ and $\tau \stackrel{\nabla}{=} \tau'$.*

Proof. From $\mathcal{D}; \Gamma; \Sigma \vdash_* e : \tau$ and $\mathcal{D}; \Gamma; \Sigma \vdash_* H + S$, using Definition 4 and Definition 11, we conclude that $\exists C_d$ and C_{sh} such that $C_d; \mathcal{D}; \Gamma; \Sigma \vdash e : \tau$ and $C_{sh}; \mathcal{D}; \Gamma; \Sigma \vdash H + S$. Since the only type variables that need to be used in common in both derivations are those that are present in Γ and Σ , we can construct a C such that $C \supseteq C_d$, $C \supseteq C_{sh}$, and $\mathfrak{R}(C)$. Now, by weakening, we have $C; \mathcal{D}; \Gamma; \Sigma \vdash_* e : \tau$ and $C; \mathcal{D}; \Gamma; \Sigma \vdash_* H + S$. The result now follows from Theorem 2 (subject reduction). □

Definition 19 (Stuck State). A system state $S; H; e$ is said to be **stuck** if $e \neq v$ and there are no S', H' , and e' such that $S; H; e \Rightarrow S'; H'; e'$.

Theorem 4 (Type Soundness). Let \Rightarrow^* denote the reflexive-transitive-closure of \Rightarrow . For any canonical expression e , if $\mathcal{D}; \emptyset; \Sigma \vdash_* e : \tau$, $\mathcal{D}; \emptyset; \Sigma \vdash_* H + S$, $\models \mathcal{D}$, and $S; H; e \Rightarrow^* S'; H'; e'$ then $S'; H'; e'$ is not stuck. That is, execution of a closed, canonical, well typed expression cannot lead to a stuck state.

Proof. By straightforward induction on the length of \Rightarrow^* . If $e = v$, proof is immediate. Otherwise, from Lemma 1 (Progress), we know that we can take at least one step forward. Further, from Lemma 3 (Preservation), we know that a (left/right) execution of a well typed expression in with respect to a well typed stack and heap will always result in another well typed expression, stack and heap. Proof now follows from induction hypothesis. \square

C Type Inference

Definition 20 (Type Inference). Type inference is a program transformation that accepts a program in which let expressions are not annotated with their kinds, and returns the same programs in which let expressions are annotated with their kinds and all expressions are annotated with their types.

Definition 21 (Constraint Collection over Inference Derivation). Similar to Definition 3, we write $\{\Gamma; \Sigma \vdash e : \tau \mid \mathcal{C}\}$ to denote the set of all constrained types and unconstrained type variables used in the derivation of $\Gamma; \Sigma \vdash e : \tau \mid \mathcal{C}$.

$$\begin{aligned} \{\Gamma; \Sigma \vdash () : \text{unit} \mid \emptyset\} &= \{\Gamma, \Sigma\} \\ \{\Gamma; \Sigma \vdash x : \tau \mid \mathcal{C}\} &= \{\Gamma, \Sigma, \tau, \mathcal{C}\} \end{aligned}$$

Other cases are similar.

Definition 22 (Notational Derivations). We define the following derivations for notational convenience:

$$\frac{\Gamma; \Sigma \vdash e : \tau \mid \mathcal{C} \quad \theta \vdash_{\text{sol}} \mathcal{C} \rightsquigarrow \mathcal{D} \quad \theta \Vdash \{\Gamma, \Sigma, \tau, \mathcal{C}\}}{\theta; \Gamma; \Sigma \vdash e : \tau \mid \mathcal{D}}$$

$$\frac{\theta; \Gamma; \Sigma \vdash e : \tau \mid \mathcal{D} \quad \mathcal{C} \Vdash \{\theta\langle \Gamma; \Sigma \vdash e : \tau \mid \mathcal{C} \rangle\}}{\mathcal{C}; \theta; \Gamma; \Sigma \vdash e : \tau \mid \mathcal{D}}$$

$$\frac{\theta\langle \mathcal{D} \rangle; \theta\langle \Gamma \rangle; \theta\langle \Sigma \rangle \vdash \theta\langle e \rangle : \theta\langle \tau \rangle}{\theta; \mathcal{D}; \Gamma; \Sigma \vdash e : \tau}$$

$$\frac{\theta\langle \mathcal{D} \rangle; \theta\langle \Gamma \rangle; \theta\langle \Sigma \rangle \vdash_* \theta\langle e \rangle : \theta\langle \tau \rangle \quad \theta \Vdash \{\Gamma, \Sigma, \tau, \mathcal{D}\}}{\theta; \mathcal{D}; \Gamma; \Sigma \vdash_* e : \tau}$$

$$\frac{\mathcal{C}; \theta\langle \mathcal{D} \rangle; \theta\langle \Gamma \rangle; \theta\langle \Sigma \rangle \vdash \theta\langle e \rangle : \theta\langle \tau \rangle \quad \theta \Vdash \{\Gamma, \Sigma, \tau, \mathcal{D}\}}{\mathcal{C}; \theta; \mathcal{D}; \Gamma; \Sigma \vdash e : \tau}$$

Theorem 5 (Correctness of Unification). If $\mathcal{U}(\mathcal{C}) = (\mathcal{D}, \theta)$, then $\theta \vdash_{\text{sol}} \mathcal{C} \rightsquigarrow \mathcal{D}$

Proof. By straightforward induction on the derivation of $\mathcal{U}(\mathcal{C})$. \square

Lemma 23 (Consistency of Unification). If $\Vdash \{\mathcal{C}\}$ and $\mathcal{U}(\mathcal{C}) = (\mathcal{D}, \theta)$, then $\Vdash \theta\{\mathcal{C}\}$

Proof. By straightforward induction on the derivation of $\mathcal{U}(\mathcal{C})$. \square

Theorem 6 (Satisfiability of Unified Constraints). If $\mathcal{U}(\mathcal{C}) = (\mathcal{D}, \theta_u)$, then $\exists \theta_s$ such that $\theta_{u.s} \vdash_{\text{sat}} \mathcal{C} \rightsquigarrow \mathcal{D}$.

Proof. By induction on the derivation of $\mathcal{U}(\mathcal{C})$. \square

Theorem 7 (Principality of Unified Types). If $\mathcal{U}(\mathcal{C}) = (\mathcal{D}, \theta_u)$, where \mathcal{C} is a set of constraints obtained from the type inference algorithm, then $\forall \theta_s$ such that $\theta_s \vdash_{\text{sol}} \mathcal{C} \rightsquigarrow \mathcal{D}'$, we have $\theta_u \sqsubseteq \theta_s$.

Proof. By induction on the derivation of $\mathcal{U}(\mathcal{C})$. The interesting case is $\mathcal{U}(\varsigma \downarrow \rho = \varrho)$, in particular $\mathcal{U}(\varsigma \downarrow \rho = \alpha \downarrow \rho')$. This case is handled by noting that the inference algorithm only produces $\alpha \downarrow \rho'$ types in the pair-selection rule, where ρ' is of the form $\alpha_1 \downarrow \rho_1 \times \alpha_2 \downarrow \rho_2$. \square

Theorem 8 (Decidability of Unification). *A canonical derivation of $\mathcal{U}(\mathcal{C})$, where no two applications of the reflexive rule happen consecutively, halts for all \mathcal{C} . That is, $\forall \mathcal{C}, \mathcal{U}(\mathcal{C})$ decidably either succeeds with (\mathcal{D}, θ) or fails with \perp .*

Proof. Let the *degree* of unification be defined as the tuple (number of α or κ variables, number of MPC constants in \mathcal{C} , the size of types within the equality constraints in \mathcal{C}). Now, the proof is by induction on the derivation of $\mathcal{U}(\mathcal{C})$, where we show that the degree of unification reduces in every recursive call. \square

Lemma 24 (Consistency of Inferred Types). *If $\Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{C}$ and $\Vdash \{\Gamma, \Sigma\}$, then $\Vdash \{\Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{C}\}$.*

Proof. By straightforward induction on the derivation of $\Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{C}$, using Lemma 1. \square

Lemma 25 (Substitution Consistency over Inference Derivation). *If $\Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{C}$ and $\theta \Vdash \{\Gamma, \Sigma, \tau, \mathcal{C}\}$, then $\theta \Vdash \{\Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{C}\}$.*

Proof. By straightforward induction on the derivation of $\Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{C}$, using Lemma 1, and using the fact that we can assume $\text{ftv}(\theta) \cap \text{ftv}(\{\Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{C}\}) = \text{ftv}(\Gamma, \Sigma, \tau, \mathcal{C})$. \square

Theorem 9 (Soundness of Type Inference). *If $\mathcal{C}; \theta; \Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{D}$ then $\mathcal{C}; \theta; \mathcal{D}; \Gamma; \Sigma \vdash e : \tau$.*

Proof. By induction on the derivation of $\Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{C}$.

1. From the premise $\mathcal{C}; \theta; \Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{C}$ of the theorem, using Definition 3 and Definition 12, we obtain:

- (a) $\Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{C}$
- (b) $\theta \vdash_{\text{sol}} \mathcal{C} \rightsquigarrow \mathcal{D}$
- (c) $\theta \Vdash \{\Gamma, \Sigma, \tau, \mathcal{C}\}$. Using Lemma 3, this can be written as $\theta \Vdash \{\Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{C}\}$
- (d) $\theta \Vdash_{\text{cst}} \{\Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{C}\}$
- (e) $\mathcal{C} \Vdash \{\theta(\Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{C})\}$

Now, we proceed by case analysis on the last step of the derivation.

2. Cases I-unit, I-Bool, I-Hloc and I-Sloc are trivial

3. Case I-Id: In this case, we have

$$\frac{\Gamma(x) = \forall \bar{\alpha}. \tau_x \setminus \mathcal{D}_x \quad \theta_x = \overline{[\alpha \mapsto \beta]} \quad \Vdash_{\text{new}} \bar{\beta}}{\Gamma; \Sigma \vdash_{\bar{\tau}} x : \theta_x \langle \tau_x \rangle \mid \theta_x \langle \mathcal{D}_x \rangle}$$

Proof follows from cases (1.b and 1.c) using T-ID rule and the fact that variable replacement is always consistent.

4. Case I-Lambda:

(a) In this case, we have

$$\frac{\Gamma, x \mapsto \beta \downarrow \alpha; \Sigma \vdash_{\bar{\tau}} e_r : \tau_r \mid \mathcal{C} \quad \Vdash_{\text{new}} \alpha \beta \beta' \gamma \gamma' \delta}{\Gamma; \Sigma \vdash_{\bar{\tau}} \lambda x. e_r : \beta' \downarrow \alpha \rightarrow \gamma' \downarrow \delta \mid \mathcal{C} \cup \{\tau_r = \gamma \downarrow \delta\}}$$

(b) From cases (1 and 4.a), by weakening, we have $\mathcal{C}; \theta; \Gamma, x \mapsto \beta \downarrow \alpha; \Sigma \vdash_{\bar{\tau}} e_r : \tau_r \mid \mathcal{C}$. Now, from induction hypothesis, we have $\mathcal{C}; \theta; \mathcal{D}; \Gamma, x \mapsto \beta \downarrow \alpha; \Sigma \vdash e_r : \tau_r$

(c) From case (1.d), using weakening, we obtain $\theta \Vdash_{\text{cst}} \{\beta \downarrow \alpha, \beta' \downarrow \alpha\}$. Now, from Lemma 6 (property 3), we conclude that $\theta \langle \beta \downarrow \alpha \rangle \stackrel{\nabla}{=} \theta \langle \beta' \downarrow \alpha \rangle$. Similarly, we conclude that $\theta \langle \gamma \downarrow \delta \rangle \stackrel{\nabla}{=} \theta \langle \gamma' \downarrow \delta \rangle$.

(d) From case (1.b) and the constraint $\tau_r = \gamma \downarrow \delta$, we obtain $\theta \langle \tau_r \rangle = \theta \langle \gamma \downarrow \delta \rangle$. Now, using case (4.c), we obtain $\theta \langle \tau_r \rangle \stackrel{\nabla}{=} \theta \langle \gamma' \downarrow \delta \rangle$.

- (e) From cases (4.b, 4.c and 4.d) and the T-Lambda rule, we obtain $\theta; \mathcal{D}; \Gamma; \Sigma \vdash \lambda x.e_r : \beta' \downarrow \alpha \rightarrow \gamma' \downarrow \delta$. Finally, using this with case (1), we conclude that $\mathcal{C}; \theta; \mathcal{D}; \Gamma; \Sigma \vdash \lambda x.e_r : \beta' \downarrow \alpha \rightarrow \gamma' \downarrow \delta$.

5. Case I-App:

- (a) In this case, we have

$$\frac{\Gamma; \Sigma \vdash e_1 : \tau_1 \mid \mathcal{C}_1 \quad \Gamma; \Sigma \vdash e_2 : \tau_2 \mid \mathcal{C}_2 \quad \frac{}{\text{new}} \alpha \beta \beta' \gamma \gamma' \delta \varepsilon}{\Gamma; \Sigma \vdash e_1 e_2 : \varepsilon \downarrow \gamma \mid \mathcal{C}}$$

where,

$$\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 = \alpha \downarrow (\beta' \downarrow \beta \rightarrow \gamma' \downarrow \gamma), \tau_2 = \delta \downarrow \beta\}.$$

- (b) Using weakening on case (1), we obtain

- i. $\theta \vdash_{\text{sol}} \mathcal{C}_1 \rightsquigarrow \mathcal{D}_1$ and $\theta \vdash_{\text{sol}} \mathcal{C}_2 \rightsquigarrow \mathcal{D}_2$
- ii. $\theta \Vdash \{\Gamma; \Sigma \vdash e_1 : \tau_1 \mid \mathcal{C}_1\}$ and $\theta \Vdash \{\Gamma; \Sigma \vdash e_2 : \tau_2 \mid \mathcal{C}_2\}$
- iii. $\mathcal{C} \Vdash \{\theta \langle \Gamma; \Sigma \vdash e_1 : \tau_1 \mid \mathcal{C}_1 \rangle\}$ and $\mathcal{C} \Vdash \{\theta \langle \Gamma; \Sigma \vdash e_2 : \tau_2 \mid \mathcal{C}_2 \rangle\}$.

- (c) From cases (5.a and 5.b), we obtain $\mathcal{C}; \theta; \Gamma; \Sigma \vdash e_1 : \tau_1 \mid \mathcal{C}_1$ and $\mathcal{C}; \theta; \Gamma; \Sigma \vdash e_2 : \tau_2 \mid \mathcal{C}_2$.

- (d) From case (5.c) and induction hypothesis, we obtain $\mathcal{C}; \theta; \mathcal{D}_1; \Gamma; \Sigma \vdash e_1 : \tau_1$ and $\mathcal{C}; \theta; \mathcal{D}_2; \Gamma; \Sigma \vdash e_2 : \tau_2$.

- (e) Note that since $\theta \vdash_{\text{sol}} \mathcal{C} \rightsquigarrow \mathcal{D}$ and $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 = \alpha \downarrow (\beta' \downarrow \beta \rightarrow \gamma' \downarrow \gamma), \tau_2 = \delta \downarrow \beta\}$, we have $\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2$.

- (f) From case (1.b) and the constraint $\tau_1 = \alpha \downarrow (\beta' \downarrow \beta \rightarrow \gamma' \downarrow \gamma)$, we have $\theta \langle \tau_1 \rangle = \theta \langle \alpha \downarrow (\beta' \downarrow \beta \rightarrow \gamma' \downarrow \gamma) \rangle$.

- (g) Since $\theta \langle \tau_1 \rangle \preceq \nabla(\theta \langle \tau_1 \rangle)$, we have $\theta \langle \tau_1 \rangle \preceq \nabla(\theta \langle \alpha \downarrow (\beta' \downarrow \beta \rightarrow \gamma' \downarrow \gamma) \rangle)$. From case (1.d) and Lemma 3, we have $\nabla(\theta \langle \alpha \downarrow (\beta' \downarrow \beta \rightarrow \gamma' \downarrow \gamma) \rangle) = \nabla(\theta \langle \nabla(\alpha \downarrow (\beta' \downarrow \beta \rightarrow \gamma' \downarrow \gamma)) \rangle)$. Further, we have $\nabla(\theta \langle \nabla(\alpha \downarrow (\beta' \downarrow \beta \rightarrow \gamma' \downarrow \gamma)) \rangle) = \nabla(\theta \langle \beta' \downarrow \beta \rightarrow \gamma' \downarrow \gamma \rangle) = \theta \langle \beta' \downarrow \beta \rangle \rightarrow \theta \langle \gamma' \downarrow \gamma \rangle$.

- (h) From cases (5.f and 5.g), we have $\theta \langle \tau_1 \rangle \preceq \theta \langle \beta' \downarrow \beta \rangle \rightarrow \theta \langle \gamma' \downarrow \gamma \rangle$.

- (i) Similarly, using the constraint $\tau_2 = \delta \downarrow \beta$, we obtain $\theta \langle \tau_2 \rangle \preceq \nabla(\theta \langle \delta \downarrow \beta \rangle)$.

- (j) Similarly, using the constraint $\tau_2 = \delta \downarrow \beta$, we obtain $\theta \langle \tau_2 \rangle \preceq \nabla(\theta \langle \delta \downarrow \beta \rangle)$.

- (k) From case (1.d), using weakening, we obtain $\theta \Vdash_{\text{cst}} \{\gamma' \downarrow \gamma, \varepsilon \downarrow \gamma\}$. Now, from Lemma 6 (property 4), we conclude that $\Delta(\theta \langle \gamma' \downarrow \gamma \rangle) \preceq \theta \langle \varepsilon \downarrow \gamma \rangle$.

- (l) From cases (5.d, 5.h, 5.i and 5.k) and the T-App rule, we conclude that $\theta; \mathcal{D}; \Gamma; \Sigma \vdash e_1 e_2 : \varepsilon \downarrow \gamma$. Finally, using this with case (1), we conclude that $\mathcal{C}; \theta; \mathcal{D}; \Gamma; \Sigma \vdash e_1 e_2 : \varepsilon \downarrow \gamma$.

6. Case I-Let-Exp: In this case, we have:

$$\frac{\Gamma; \Sigma \vdash e_1 : \tau_1 \mid \mathcal{C}_1 \quad e_1 \neq v \quad \Gamma, x \mapsto \alpha \downarrow \beta; \Sigma \vdash e_2 : \tau_2 \mid \mathcal{C}_2 \quad \frac{}{\text{new}} \alpha \beta \gamma \kappa}{\Gamma; \Sigma \vdash \text{let}^\kappa x = e_1 \text{ in } e_2 : \tau_2 \mid \mathcal{C}}$$

where $\mathcal{C} = \mathcal{C}_1 \cup \{\tau_1 = \gamma \downarrow \beta, \kappa = \psi\} \cup \mathcal{C}_2$.

Proof is similar to the I-App case, proof follows from induction hypothesis using the T-Let-M rule, case (1) and Lemma 6 (property 4).

7. Case I-Let-Val:

- (a) In this case, we have:

$$\frac{\Gamma; \Sigma \vdash v : \tau_1 \mid \mathcal{C}_1 \quad \mathcal{C}'_1 = \mathcal{C}_1 \cup \{\tau_1 = \gamma \downarrow \beta\} \quad \mathcal{U}(\mathcal{C}'_1) = (\mathcal{D}_u, \theta_u) \quad \mathcal{D}_s = \mathcal{D}_u \cup \{\star_x^\kappa(\tau_s)\} \quad \tau_s = \theta_u \langle \delta \downarrow \beta \rangle \quad \{\bar{\alpha}\} = \text{ftv}(\tau_s, \mathcal{D}_s) \setminus \text{ftv}(\theta_u \langle \Gamma \rangle, \theta_u \langle \Sigma \rangle) \quad \Gamma, x \mapsto \forall \bar{\alpha}. \tau_s \setminus \mathcal{D}_s; \Sigma \vdash e : \tau_2 \mid \mathcal{C}_2 \quad \frac{}{\text{new}} \beta \gamma \delta \bar{\varepsilon} \kappa}{\Gamma; \Sigma \vdash \text{let}^\kappa x = e_1 \text{ in } e_2 : \tau_2 \mid \mathcal{C}'_1[\bar{\varepsilon}/\bar{\alpha}] \cup \mathcal{C}_2}$$

- (b) From case (7.a) and Theorem 1, we have $\theta_u \vdash_{sol} C' \rightsquigarrow \mathcal{D}_u$
- (c) By weakening on case (1.b), we have $\theta \vdash_{sol} C'_1[\overline{\alpha}] \rightsquigarrow \mathcal{D}_1$ and $\theta \vdash_{sol} C_2 \rightsquigarrow \mathcal{D}_2$ for some \mathcal{D}_1 and \mathcal{D}_2 . Further, $\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2$
- (d) From cases (7.c and 7.d), Theorem 3 and the fact that the variables $\overline{\alpha}$ are purely local to the derivation $\Gamma; \Sigma \vdash_{\overline{\alpha}} v : \tau_1 \mid \mathcal{C}_1$, we conclude that $\exists \theta'$ such that
 - i. $\theta_u \sqsubseteq \theta'$
 - ii. $\theta' \vdash_{sol} C' \rightsquigarrow \theta' \langle \mathcal{D}_u \rangle$
 - iii. $C; \theta'; \Gamma; \Sigma \vdash_{\overline{\alpha}} v : \tau_1 \mid \mathcal{C}_1$
 - iv. $\theta' \langle \Gamma \rangle = \theta \langle \Gamma \rangle$, $\theta' \langle \Sigma \rangle = \theta \langle \Sigma \rangle$, $\theta' \langle \tau_1 \rangle = \theta \langle \tau_1 \rangle$, $\theta' \langle \tau_s \rangle = \theta \langle \tau_s \rangle$, and $\theta' \langle \mathcal{D}_s \rangle = \theta \langle \mathcal{D}_s \rangle$.
 - v. $\{\overline{\alpha}\} = \text{ftv}(\theta \langle \tau_s \rangle, \theta \langle \mathcal{D}_s \rangle) \setminus \text{ftv}(\theta' \langle \Gamma \rangle, \theta' \langle \Sigma \rangle)$.
- (e) From case (7.d.iii) and induction hypothesis, we obtain $C; \theta'; \mathcal{D}_u; \Gamma; \Sigma \vdash v : \tau_1$. Due to case (7.d.iv), this can be written as $C; \theta; \mathcal{D}_u; \Gamma; \Sigma \vdash v : \tau_1$.
- (f) Let $\tau'_1 = \nabla \langle \theta \langle \tau_1 \rangle \rangle$. Evidently, $\theta \langle \tau_1 \rangle \preceq \tau'_1$. using this in case (7.e), we obtain $C; \theta; \mathcal{D}_u; \Gamma; \Sigma \vdash v \preceq \tau'_1$.
- (g) Similar to the I-App case, using Lemma 6 (property 4), we obtain $\theta \langle \tau_s \rangle \preceq \tau'_1$.
- (h) Evidently, $\theta \langle \mathcal{D} \rangle = \theta \langle \mathcal{D}_u \rangle \cup \{\star_x^{\kappa}(\theta \langle \tau \rangle)\}$.
- (i) From cases (7.d.v and 7.d.iv), we have $\{\overline{\alpha}\} = \text{ftv}(\theta \langle \tau_s \rangle, \theta \langle \mathcal{D}_s \rangle) \setminus \text{ftv}(\theta \langle \Gamma \rangle, \theta \langle \Sigma \rangle)$.
- (j) From weakening on case (1) and induction hypothesis, we have $C; \theta; \mathcal{D}_2; \Gamma, x \mapsto \nabla \overline{\alpha}. \tau_s \setminus \mathcal{D}_s; \Sigma \vdash e : \tau_2$.
- (k) Finally, from cases (7.f, 7.g, 7.h, 7.i, 7.j and 1), using the T-Let-MP rule, we conclude that $C; \theta; \mathcal{D}; \Gamma; \Sigma \vdash \text{let } x = v \text{ in } e : \tau_2$.

□

Theorem 10 (Completeness of Type Inference). *If $C; \theta; \mathcal{D}; \Gamma; \Sigma \vdash e : \tau$, then $\exists \theta' \supseteq \theta$ and $C' \supseteq C$ such that $C'; \theta'; \Gamma; \Sigma \vdash_{\overline{\alpha}} e : \tau \mid \mathcal{D}$ and $\text{dom}(\theta') \setminus \text{dom}(\theta) = \text{dom}(C') \setminus \text{dom}(C) = \{\overline{\alpha} \mid \alpha \text{ is a fresh variable used in } \Gamma; \Sigma \vdash_{\overline{\alpha}} e : \tau \mid \mathcal{C}\}$.*

Proof. By induction on the derivation of $C; \theta; \mathcal{D}; \Gamma; \Sigma \vdash e : \tau$.

1. From the premise $C; \theta; \mathcal{D}; \Gamma; \Sigma \vdash e : \tau$ of the theorem, using using Definition 3 and Definition 4, we obtain:

- (a) $\theta; \mathcal{D}; \Gamma; \Sigma \vdash e : \tau$
- (b) $\theta \Vdash \{\Gamma, \Sigma, \tau, \mathcal{D}\}$
- (c) $C \Vdash \{\theta \langle \mathcal{D} \rangle; \theta \langle \Gamma \rangle; \theta \langle \Sigma \rangle \vdash \theta \langle e \rangle : \theta \langle \tau \rangle\}$

We need to show that for some \mathcal{C} , $\theta' \supseteq \theta$ and $C' \supseteq C$,

- (a) $\Gamma; \Sigma \vdash_{\overline{\alpha}} e : \tau \mid \mathcal{C}$
- (b) $\theta' \vdash_{sol} \mathcal{C} \rightsquigarrow \mathcal{D}$
- (c) $\theta' \Vdash \{\Gamma, \Sigma, \tau, \mathcal{C}\}$
- (d) $C' \Vdash \{\theta' \langle \Gamma \rangle; \Sigma \vdash_{\overline{\alpha}} e : \tau \mid \mathcal{C}\}$

Now, we proceed by case analysis on the last step of the derivation.

2. Cases T-Unit, T-Bool, T-Hloc, and T-Sloc are trivial.

3. Case T-ID: In this case, we have:

$$\frac{\theta \langle \Gamma \rangle(x) = \nabla \overline{\alpha}. \theta \langle \tau \rangle \setminus \theta \langle \mathcal{D} \rangle \quad \theta_x \Vdash \{\theta \langle \tau \rangle, \theta \langle \mathcal{D} \rangle\} \quad \text{dom}(\theta_x) = \{\overline{\alpha}\}}{\theta \circ \theta_x \langle \mathcal{D} \rangle; \Gamma; \Sigma \vdash x : \theta \circ \theta_x \langle \tau \rangle}$$

Without change in meaning, we can write $\Gamma(x) = \nabla \overline{\gamma}. \tau \setminus \mathcal{D}$. Now, we can write:

$$\frac{\Gamma(x) = \nabla \overline{\gamma}. \tau \setminus \mathcal{D} \quad \theta = \overline{[\gamma \mapsto \alpha]} \quad \overline{\text{new}} \overline{\alpha}}{\Gamma; \Sigma \vdash_{\overline{\alpha}} x : \theta_y \langle \tau \rangle \mid \theta_y \langle \mathcal{D} \rangle}$$

Let $\theta' = \theta \circ \theta_x$. The result now follows from I-Id rule and case (1.c).

4. Case T-Lambda:

(a) In this case, we have

$$\frac{\theta\langle\mathcal{D}\rangle; \theta\langle\Gamma\rangle, x \mapsto \theta\langle\tau_1\rangle; \Sigma \vdash e : \theta\langle\tau_2\rangle \quad \frac{\theta\langle\tau_1\rangle \stackrel{\nabla}{=} \theta\langle\tau'_1\rangle \quad \theta\langle\tau_2\rangle \stackrel{\nabla}{=} \theta\langle\tau'_2\rangle}{\theta\langle\mathcal{D}\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash \lambda x.e : \theta\langle\tau'_1 \rightarrow \tau'_2\rangle}}{\theta\langle\mathcal{D}\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash \lambda x.e : \theta\langle\tau'_1 \rightarrow \tau'_2\rangle}$$

(b) By induction hypothesis, we have, for some θ'' and C'' such that $C \subseteq C''$ and $\theta \sqsupseteq \theta''$, we have

- i. $C'; \theta'; \Gamma, x \mapsto \tau_1; \Sigma \vdash_i e : \tau_2 \mid C$
- ii. $\theta \vdash_{sat} C \rightsquigarrow \mathcal{D}$.

(c) We pick θ' such that $\theta' = \theta'' \circ \theta_{new}$, where

- i. $dom(\theta_{new}) = \{\alpha, \beta, \beta', \gamma, \gamma', \delta\}$
- ii. $\theta' \langle \beta \downarrow \alpha \rangle = \theta \langle \tau_1 \rangle$
- iii. $\theta' \langle \gamma \downarrow \alpha \rangle = \theta \langle \tau_2 \rangle$
- iv. $\theta' \langle \beta' \downarrow \alpha \rangle = \theta \langle \tau'_1 \rangle$, possible due to $\theta \langle \tau_1 \rangle \stackrel{\nabla}{=} \theta \langle \tau'_1 \rangle$, which continues to hold under the consistent substitution θ' , due to Lemma 6 (property 2).
- v. $\theta' \langle \gamma \downarrow \alpha \rangle = \theta \langle \tau_2 \rangle$, possible due to $\theta \langle \tau_2 \rangle \stackrel{\nabla}{=} \theta \langle \tau'_2 \rangle$, which continues to hold under θ' , due to Lemma 6 (property 2).

(d) Evidently, $\theta' \vdash_{sat} C \cup \{\tau = \gamma \downarrow \delta\} \rightsquigarrow \mathcal{D}$.

(e) We also extend C'' to C' with entries for new type variables.

(f) Now, using I-Lambda rule and case (1.c), we obtain $C'; \theta'; \Gamma; \Sigma \vdash_i \lambda x.e : \beta' \downarrow \alpha \rightarrow \gamma' \downarrow \delta \mid \mathcal{D}$

5. Case T-App:

(a) In this case, we have:

$$\frac{\theta\langle\mathcal{D}_1\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e_1 : \theta\langle\tau_1\rangle \quad \theta\langle\mathcal{D}_2\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e_2 : \theta\langle\tau_2\rangle \quad \frac{\theta\langle\tau_1\rangle \triangleleft: \tau_a \rightarrow \tau_r \quad \theta\langle\tau_2\rangle \triangleleft: \nabla(\tau_a) \quad \Delta(\tau_r) \triangleleft: \theta\langle\tau\rangle}{\theta\langle\mathcal{D}_1\rangle \cup \theta\langle\mathcal{D}_2\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e_1 e_2 : \tau}}{\theta\langle\mathcal{D}_1\rangle \cup \theta\langle\mathcal{D}_2\rangle; \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e_1 e_2 : \tau}$$

(b) From induction hypothesis with respect to e_1 , we have, for some $\mathcal{C}_1, \theta_1 \sqsupseteq \theta$, and $C_1 \supseteq C$

- i. $\Gamma; \Sigma \vdash_i e_1 : \tau_1 \mid \mathcal{C}_1$
- ii. $\theta_1 \vdash_{sol} \mathcal{C}_1 \rightsquigarrow \mathcal{D}_1$
- iii. $\theta_1 \Vdash \{\Gamma, \Sigma, \tau_1, \mathcal{C}_1\}$
- iv. $C_1 \Vdash \{\theta_1 \langle \Gamma; \Sigma \vdash_i e_1 : \tau_1 \mid \mathcal{C}_1 \rangle\}$

(c) From induction hypothesis with respect to e_2 , we have, for some $\mathcal{C}_2, \theta_2 \sqsupseteq \theta$, and $C_2 \supseteq C$

- i. $\Gamma; \Sigma \vdash_i e_2 : \tau_2 \mid \mathcal{C}_2$
- ii. $\theta_2 \vdash_{sol} \mathcal{C}_2 \rightsquigarrow \mathcal{D}_2$
- iii. $\theta_2 \Vdash \{\Gamma, \Sigma, \tau_2, \mathcal{C}_2\}$
- iv. $C_2 \Vdash \{\theta_2 \langle \Gamma; \Sigma \vdash_i e_2 : \tau_2 \mid \mathcal{C}_2 \rangle\}$

(d) Since the new type variables used in the two derivations do not collide, and since θ_1 / C_1 and θ_2 / C_2 extend θ / C with entries for new type variables only, we can write:

- i. Let $\theta_{12} = \theta_1 \circ \theta_2$ and $C_{12} = C_1 \cup C_2$.
- ii. $\theta_{12} \vdash_{sol} C_1 \cup C_2 \rightsquigarrow \mathcal{D}_1 \cup \mathcal{D}_2$
- iii. $\theta_{12} \Vdash \{\Gamma, \Sigma, \tau_1, \tau_2, \mathcal{C}_1, \mathcal{C}_2\}$
- iv. $C_{12} \Vdash \{\theta_{12} \langle \Gamma; \Sigma \vdash_i e_1 : \tau_1 \mid \mathcal{C}_1 \rangle, \theta_{12} \langle \Gamma; \Sigma \vdash_i e_2 : \tau_2 \mid \mathcal{C}_2 \rangle\}$

(e) We pick θ' such that $\theta' = \theta_{12} \circ \theta_{new}$, where

- i. $dom(\theta_{new}) = \{\alpha, \beta, \beta', \gamma, \gamma', \delta\}$

- ii. $\theta' \langle \beta' \downarrow \beta \rightarrow \gamma' \downarrow \gamma \rangle = \theta_{12} \langle \tau_a \rightarrow \tau_r \rangle$.
 - iii. $\theta' \langle \alpha \downarrow (\beta' \downarrow \beta \rightarrow \gamma' \downarrow \gamma) \rangle = \theta_{12} \langle \tau_1 \rangle$, possible due to $\theta \langle \tau_1 \rangle \preceq: \tau_a \rightarrow \tau_r$, which continues to hold under the consistent substitution θ' due to Lemma 6 (property 1).
 - iv. $\theta' \langle \delta \downarrow \beta \rangle = \theta_{12} \langle \tau_2 \rangle$, possible due to $\theta \langle \tau_2 \rangle \preceq: \nabla(\tau_a)$, which continues to hold under θ' .
 - v. $\theta' \langle \varepsilon \downarrow \gamma \rangle = \theta_{12} \langle \tau \rangle$, possible due to $\Delta(\tau_r) \preceq: \theta \langle \tau \rangle$, which continues to hold under θ' .
- (f) From cases (5.b.i and 5.b.ii) and the I-App rule, we obtain $\Gamma; \Sigma \vdash_{\bar{\tau}} e_1 e_2 : \varepsilon \downarrow \gamma \mid \mathcal{C}$, where $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 = \alpha \downarrow (\beta' \downarrow \beta \rightarrow \gamma' \downarrow \gamma), \tau_2 = \delta \downarrow \beta\}$.
- (g) From cases (5.d.ii, 5.e.ii, 5.e.iii, 5.e.iv, 5.e.v and 5.f), we conclude that $\theta' \vdash_{sol} \mathcal{C} \rightsquigarrow \mathcal{D}_1 \cup \mathcal{D}_2$
- (h) From cases (5.d.iii and 5.e and 5.f), we obtain $\theta' \Vdash \{\Gamma, \Sigma, \tau, \mathcal{C}\}$
- (i) By extending C_{12} to C' with appropriate entries for $\alpha, \beta, \beta', \gamma, \gamma'$, and δ , and using case (5.d.iv), we obtain $C' \Vdash \{\theta' \langle \Gamma; \Sigma \vdash_{\bar{\tau}} e_1 e_2 : \tau \mid \mathcal{C} \rangle\}$.
- (j) Finally, from cases (5.f, 5.g, 5.h and 5.i), we conclude that $C'; \theta'; \Gamma; \Sigma \vdash_{\bar{\tau}} e_1 e_2 : \tau \mid \mathcal{D}_1 \cup \mathcal{D}_2$.

6. Other cases are similar. □

Theorem 11 (Soundness of Type Inference). *If $\theta; \Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{D}$ then $\theta; \mathcal{D}; \Gamma; \Sigma \vdash_* e : \tau$.*

Proof. Follows from Theorem 5 and Lemma 1 (property 3). □

Theorem 12 (Type Checkability). *If $\Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{C}$ and $\mathcal{U}(\mathcal{C}) = (\mathcal{D}_u, \theta_u)$, then, $\exists \theta_s$ such that $\theta_s \vdash_{sat} \mathcal{D}_u \rightsquigarrow \mathcal{D}$, $\theta_{u.s} \langle e \rangle$ is canonical, and $\theta_{u.s}; \mathcal{D}; \Gamma; \Sigma \vdash_* e : \tau$.*

Proof. Follows from Theorem 7 and Theorem 2. □

Theorem 13 (Completeness of Type Inference). *If $\theta; \mathcal{D}; \Gamma; \Sigma \vdash_* e : \tau$, then $\exists \theta' \supseteq \theta$ such that $\theta'; \Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \mid \mathcal{D}$.*

Proof. Follows from Theorem 6 and Lemma 1 (property 4). □