

Heuristic Type Inference in BitC

SRL Technical Report SRL2008-01

Swaroop Sridhar Jonathan S. Shapiro Scott F. Smith
swarop@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 that is designed for safe systems programming. A key feature of the type system is a new mutability model that combines explicitly 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. The paper identifies the challenges posed by this type system for type inference, and proposes a heuristic approach to solving this problem.

1 Introduction

Systems programs rely on fine-grain control of data representation and use of state to achieve performance, conformance to hardware specification, and temporal predictability. Robustness and reuse of systems codes can be greatly improved by leveraging modern programming language features — such as static type safety, type inference, higher order functions, and polymorphism — but only if these features can be provided without sacrificing the above mentioned requirements. No existing language supports both of these feature sets simultaneously. Therefore, systems programmers continue to use Ada [8], C [10], C++ [9] or resort to domain specific and assembly level languages. We first discuss the support for these features in existing languages, identify the challenges in combining these feature sets and describe our approach toward solving this problem.

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 [18] or Haskell [15]) combines the advantages of static type safety with much of the convenience provided by dynamically typed languages like Python [20]. Automatic inference of polymorphism simplifies generic programming, and therefore increases the reuse and reliability of code. Safe languages like Java [19], C# [16], or Vault [2] do not support type inference. Cyclone [11] supports polymorphism only for functions that are explicitly annotated with a polymorphic type.

Representation Control: A systems programming language must be expressive enough to specify details of data-structure layout (boxed or unboxed), alignment and allocation (stack or heap). This feature is essential for systems programming for reasons of performance (ex: to control cache and paging behavior), conformance to hardware specification (ex: page table entries), and interfacing with external C or assembly code and data. A careful implementation of the standard TCP/IP protocol stack in Standard ML incurs 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 in systems programs, data structure representation is as important as, or even more important than high level algorithms.

The philosophy of ML-like languages is that programs specify semantics and not realization (implementation). However, in systems programs, statements about representation and location are *prescriptive*, not *descriptive*. Compilers like TIL [25] implement unboxed representation as a discretionary optimization. Prescriptive requirements are not discretionary. First class treatment of representation is required in systems programming languages.

Mutability Support: One of the key features essential for systems programming is support for first class mutability (c.f. C). The support for mutability must be first class in the sense that any location (stack or heap) can be mutable, and we should be able to specify mutability at field level granularity. Many modern programming languages — particularly those that support type inference — do not support this first class notion of mutability. ML does not support first class mutability; all mutable cells must reside in the heap. In Haskell, all state must be encapsulated within Monads [14]. In contrast to the C-like languages, these functional languages *do* have a mathematically sound notion of immutability.

We have identified three features — unboxed representation, mutability and polymorphic type inference — that are desirable in a systems programming language. While several existing languages support two of these features, none combine all three of them elegantly. In this paper, we endeavor to bridge this gap between systems programming and modern language designs through the BitC [22] programming language. BitC is a direct expression of typed lambda calculus with side effects, extended to be able to reflect the semantics of explicit representation. It supports polymorphic type inference and a new model of mutability which is expressive and has sound semantics.

BitC is a call-by-value expression language. BitC’s support for unboxed mutability makes it desirable to allow some freedom in the compatibility of types with respect to their mutability at copy boundaries. This kind of compatibility has ramifications for type inference since there is no longer a unique way to type an expression. However, usability constraints require that we minimize the amount of type annotations required from the programmer. In this paper, we discuss some of these issues, and present a solution based on a simple extension to the Hindley-Milner inference algorithm [17] that applies certain heuristics to subjectively infer the *appropriate* type for all expressions.

2 The Language

BitC is a safe, systems programming language. A detailed description of the language and its support for systems programming (including a rich set of primitive data types and bit-fields, type classes [12] for overloading, representation of explicitly boxed and unboxed data-structures, discriminated unions, tagged-unions with inlined discriminators, *etc.*) can be found in the language definition [22]. In this paper, we limit our presentation to a core calculus of BitC called \mathbb{B} in the interest of brevity. The user visible part of \mathbb{B} is defined below:

Patterns	$p ::= x \mid p : \tau \mid (p, p) \mid [p^*] \mid x : p$
Values	$v ::= () \mid true \mid false \mid \lambda p.e \mid (v, v) \mid [\bar{v}]$
Expressions	$e ::= x \mid () \mid true \mid false \mid \lambda p.e \mid e e \mid e : \tau \mid e := e \mid \text{dup}(e) \mid e^\wedge$ $\mid (e, e) \mid e.1 \mid e.2 \mid [\bar{e}] \mid e : e \mid \text{match } e \text{ with } p \text{ in } e \mid \text{else } e$ $\mid \text{if } e \text{ then } e \text{ else } e \mid \text{let } x : \sigma = e \text{ in } e$
Types	$\tau ::= \alpha \mid \text{unit} \mid \text{bool} \mid \tau \rightarrow \tau \mid \uparrow\tau \mid \Psi\tau \mid \tau \times \tau \mid [\tau] \mid \tau \setminus \mathcal{C}$
Type Scheme	$\sigma ::= \tau \mid \forall \bar{\alpha}.\sigma$
Constraints	$\mathcal{C} ::= \emptyset \mid \{\tau = \tau \mid \tau \cong \tau\}$

Program variables are denoted by $x, y, \text{etc.}$ and type variables by $\alpha, \beta, \text{etc.}$ \bar{X} denotes zero or more X s separated by a semicolon. The $[]$ brackets denote optional parts of syntax. $\uparrow\tau$ represents a reference (pointer) type and $\Psi\tau$ represents a mutable type. The type $\tau \setminus \mathcal{C}$ is a constrained type with the set of constraints \mathcal{C} . The relation \cong will be defined in section 4. The `let` construct can be used for allocating (possibly mutable) stack variables and to create let-polymorphic bindings. The expression `dup(e)`, where e is of 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. The $:=$ operator updates (mutates) both stack and heap locations. Unlike ML, $:=$ does not dereference its target. Pairs $(,)$ are *unboxed* structures whose constituent elements contiguously allocated on the stack, or in their containing data-structure. $e.1$ and $e.2$ perform selection from pair values. Lists $[\dots]$ have a *boxed* representation.

3 The Mutability Model

Traditionally, there are two models of mutability studied in the case of imperative languages. One of them is the ML model, where there is a clear separation between name bindings and updatable locations. All updatable (mutable) locations live in the heap within “ref cells”. Fetching the value inside a ref cell requires an explicit dereferencing operation. The major advantage of this approach is that types are definitive about the mutability of every location, across

all aliases. In this sense, we can say that the support for mutability is mathematically “well-founded.” This model benefits tools that perform static analysis or model checking because conclusions drawn about location immutability need never be conservative. This model of mutability also increases the amount of optimization the compiler can safely perform without complex alias analysis.

The other well known model of mutability is the C model, wherein the support for mutability is “first-class” in the sense that *locations* with mutable type can be passed as arguments and stored in data-structures. This model permits mutation of stack variables and unboxed values. There is a notion of *lvalues* which are expressions that can be the target of an assignment, and *rvalues*, that are otherwise used in computations. The extraction of the value from a (mutable) location is implicit, and does not require dereferencing. However, in this model, types cannot distinguish mutable values from immutable ones. 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 compiler or other analytical engines are not entitled to believe that certain locations or fields are constant even if so declared.

\mathbb{B} supports well-founded first class mutability. Similar to ML, we impose the “one location, one type” rule.

```
let cp :  $\uparrow$ bool = dup(true) in let p :  $\uparrow\Psi$ bool = cp (* Error *)
```

We see that `cp` has the type reference to `bool` (\uparrow bool). This type is incompatible with that of `p`, reference to mutable-bool ($\uparrow\Psi$ bool). In order to support unboxed mutability, we still need to have a notion of lvalues. This not only preserves the programmer’s mental model of the relationship between locations storage, but also ensures that compiler transformations are semantics preserving. In \mathbb{B} , the legal lvalues are defined by: $L_v ::= x \mid L_v . 1 \mid L_v . 2 \mid L_v \wedge \mid L_v : \tau$

4 Copy Compatibility

Since \mathbb{B} is a call-by-value language, it is desirable that we allow some freedom in the compatibility of types with respect to their mutability at a copy boundary. For example, in the following expression:

```
let f $\lambda$ xn =  $\lambda$ x.(let __ = x := _some-fxn_ x in _fnx-returning-unit_ x) in
let y : bool = true in f $\lambda$ xn y
```

the type of `f λ xn` is $(\Psi$ bool) \rightarrow unit, whereas that of the actual argument `y` is `bool`. Since the formal argument `x` is a *copy* of `y` and occupies a different location, this expression is type safe. We refer to this notion of compatibility of types as **copy compatibility**, denoted by \cong . In this example, `bool` \cong Ψ bool.

Copy compatibility need not be restricted to the outermost mutability compatibility, but must not extend past a reference boundary in order ensure that every location has unique type. We define copy compatibility for \mathbb{B} as:

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

Copy compatibility can be permitted at argument passing, new variable binding, assignment, and basically at argument/return positions of all expressions, except where an lvalue is expected or returned. For example, the expression $(x : \tau) : \Psi\tau$ is illegal since identifiers are lvalues, but the branches of a conditional can have different but copy compatible types, as in: `if true then a : τ else b : $\Psi\tau$.`

5 Type Inference

We would like to employ an inference algorithm with the following properties:

1. The inference algorithm must be decidable without programmer annotations. The problem with programmer annotations is pragmatic rather than ideological: ease of prototyping requires that these annotations be minimized.
2. The inference algorithm must ideally be complete. In the absence of principal types, it must minimize programmer annotations in the common case, and must be capable of inferring all sound types at least when guided by explicit annotation. This requirement is orthogonal to requirement (1).

3. The inference algorithm must automatically infer polymorphism (without programmer annotations) in order to maximize code reuse and encourage good software engineering.
4. The inference algorithm must not require whole program analysis. Whole program inference precludes separate compilation, poses scalability problems for large projects, and can result in surprising behaviour if the inferred types for a program change due to modifications in a distant part of the code base.

With these considerations in mind, we chose a variation of the Hindley-Milner algorithm [17] in \mathbb{B} . We now describe challenges for type inference due to copy compatibility, explore the design choices, and finally present our solution.

Challenges Due to Copy Compatibility: When an exact type compatibility requirement is replaced in the language design by copy compatibility, it is no longer possible to infer a unique (simple) type for the expression. 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, we give `p` the so-called “maybe” type $\alpha\downarrow\text{bool}$, which is a shorthand for the constrained type $\alpha \setminus \{\alpha \cong \text{bool}\}$. The actual type will later get resolved to either `bool` or Ψbool . Similarly, in the expression (inferred types are shown beside `||`).

```
let pr = (true, false) in ... || pr :  $\alpha\downarrow(\beta\downarrow\text{bool} \times \gamma\downarrow\text{bool})$ 
```

copy compatibility is introduced at both the pair construction (arguments passed to the pair constructor), and the formation of a new binding.

Why Should We Infer Mutability? 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? 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 = [(fst (m, false) :  $\Psi\text{bool} \times \text{bool}$ )] : [ $\Psi\text{bool}$ ] in ...
```

Therefore, if mutability is not inferred, it results in a substantial increase in the number of programmer annotations, and type inference becomes ineffective.

Incompleteness of Inference The key idea of maybe types $\alpha\downarrow\tau$ is to defer commitments about the mutability status of types, and thus infer most-general types wherever possible. \mathbb{B} is a let-polymorphic language and enforces the value restriction [27]. This means that the decision about the mutability of types cannot — in general — be deferred past their let bindings. For example, in the case of the expression: `let p = [] in ...`, we cannot give `p` the type $\forall \alpha, \beta. \beta\downarrow[\alpha]$ (or $\forall \alpha. \Psi[\alpha]$ or $\forall \alpha. \beta\downarrow[\alpha]$). We must instead choose one of the polymorphic type $\forall \alpha. [\alpha]$ or the monomorphic type $\beta\downarrow[\alpha]$. That is, there is no principal type that can be inferred for `p`. Given this, we must fix these maybe types to be either mutable or immutable at a let-boundary. That is, the language definition must pick some solution for unsolved copy compatibility constraints *before* type generalization. In \mathbb{B} , we trade completeness of inference to obtain a more expressive language without making major changes to the core type system.

Inference Considerations First, we consider how to resolve copy compatibility constraints at a let boundary. One possibility is to fix all unresolved maybe types to immutable versions. For example:

```
let pf = (n :  $\Psi\text{bool}$ ,  $\lambda x. x$ ) in ... || pf :  $\forall \alpha. \text{bool} \times (\alpha \rightarrow \alpha)$ 
```

This scheme will preserve all polymorphism possible, but will mandate a programmer annotation for every mutable value. If mutable variants are chosen instead, no polymorphism can be inferred by default. Therefore, neither of these solutions are satisfactory. Further, from the standpoint of good programming ideology and static analysis, the inferred types must not be promiscuous with respect to mutability.

The previous section argued that we “lose” precision of inferred types (with respect to mutability) by the introduction of copy compatibility. Therefore, we can choose whether (or not) to introduce copy compatibility at various constructs like new bindings, function application/return, constructors, conditional expressions, *etc.* Another dimension of trade-off is whether to permit copy compatibility to the maximum permissible limit (as defined in section 4), or restrict it to top-level shallow mutability compatibility only. A further option is to require that all polymorphism be contained within function types, since function types can be polymorphic even if they abstract over mutable or maybe types.

Unless handled with care, full use of copy compatibility can result in the inferred types that are counter-intuitive to the programmer. For example:

```
let copyList =  $\lambda lst. \text{match } lst \text{ with } [] \text{ in } [] \text{ else}$   
                $\text{match } lst \text{ with } x :: rest \text{ in } x :: \text{copyList } rest \text{ in } ...$ 
```

For a naïve reader, the type of `copyList` appears to be $\forall \alpha. [\alpha] \rightarrow [\alpha]$ but is actually the more general type: $\forall \alpha, \beta.$

$[\alpha] \rightarrow [\beta] \setminus \{\alpha \cong \beta\}$. Even though both the argument and return types of *copyList* are of (boxed) list type, they are only required to be copy compatible because *copyList* copies the constituent elements, thereby using new locations. Now, if we default maybe types that are ultimately unresolved to immutable, in the following definition we obtain:

```
let mutLst2 = copyList mutLst : [Ψbool] in ... || mutLst : [bool] !!
```

which is a correct typing, but is most likely not what the programmer expects.

6 Type Inference in \mathbb{B}

Having identified the various issues and trade-offs involved in type inference, we now describe the particular design choices made in BitC/ \mathbb{B} for handling copy compatibility. They have been driven in part by our experience writing BitC programs. In \mathbb{B} , we allow copy compatibility to the full extent, up to a reference boundary. We allow copy compatibility to be invoked at arguments and return positions of all expressions that do not expect a location (lvalue).

At a let boundary, an unresolved maybe type $\alpha \downarrow \tau$ is resolved to τ . Intuitively, this means that we will default maybe types to the types of their original copies unless a better type is inferred or specified. The type τ of the original value is remembered as a “hint” to fix unresolved maybe types at a let boundary. Here, we are approximating the user’s intent to the lexical “flow” of type information. For example, in the following expression,

```
let mb : Ψbool = true in let l1 = [mb] in let l2 : Ψ[bool] = [mb]
```

the type $\beta \downarrow [\alpha \downarrow \Psi\text{bool}]$ is first inferred for both *l1* and *l2* (due to copies at list construction and binding). In the case of *l1*, the final type is resolved to $[\Psi\text{bool}]$ in accordance to the hints since we have no further information. The type of *l2* is $\Psi[\text{bool}]$, since we have a (consistent) annotation – there are no unresolved maybe types. Further, under this scheme, the *listCopy* example described in the previous section gets the more intuitive type $\forall \alpha. [\alpha] \rightarrow [\alpha]$. In the case of conflicting hints, we pick the most immutable type obtainable from all hints. This ensures that inferred types are always deterministic. For example:

```
let bp = if true then (true, false) : Ψbool × bool
         else (false, true) : bool × Ψbool in ... || bp : bool × bool
```

Here, the hints provided by the two branches of the *if* do not agree, and we resolve the conflict by picking $\text{bool} \times \text{bool}$ as the effective hint.

In the case of locally defined identifiers, the top-most mutability is inferred by studying the syntactic usage of the identifier. That is, if the identifier is used as the target of an assignment ($:=$), it is given a shallowly mutable type. This is an *ad hoc* rule that reduces the need for explicit programmer annotations in the common case like iterators. In a full language like BitC, this rule must not be used for globals in order to keep inference deterministic. Further, this “infer mutability first” rule cannot, in general, be extended beyond shallow mutability. For example, in the expression `let fPtr = dup(λx.x) in (useF1 fPtr, useF2 fPtr)`, we do not have enough local information *before* type inference to determine whether *useF1* and *useF2* set the contents of the cell pointed to by *fPtr*, or use it polymorphically.

Due to copy compatibility, two function types are *equal* regardless of the shallow mutability of the argument and return types. Therefore, we enforce a syntactic restriction that all function types must be written with immutable types at copy compatible positions. The intuition here is that type of a function must be described in the interface form, and must hide the “internal” mutability information. For example, for the function definition `let f = λx.x := true in ...`, the external type is $f : \text{bool} \rightarrow \text{unit}$, and the internal type is $f : \Psi\text{bool} \rightarrow \text{unit}$.

We must ensure that the internal types of a function do not influence the result type of applications, but the effect of arguments on the return types must be preserved. This ensures that the implementation of an abstraction can be changed (ex: from a pure recursive computation to loop involving mutation) unbeknownst to its callers. For example:

```
let p : Ψbool = true in           || p : Ψbool
let f = λx.p in let g = λx.x in   || f : ∀α.α → bool ; g : ∀α.α → α
let ff = f p in let gg = g p in ... || ff : bool ; gg : Ψbool
```

The function *f* returns $p : \Psi\text{bool}$, regardless of its input. The external type of *f* abstracts away the mutability of *p*, and thus, *ff* gets the type bool . *g* is an identity function which returns its argument. The external type of *g* preserves the mutability of the actual argument *p*, and thus *gg* gets the type Ψbool .

Mutability Polymorphism: A type is said to be mutability-polymorphic if it ranges over all [im]mutability variants of

a particular type. For example, $\forall\alpha.\alpha \downarrow \text{bool}$ is mutability polymorphic over bool . By far the most useful case of this feature is to define functions that are mutability-polymorphic over concrete types, since functions with a polymorphic type such as $\forall\alpha.\alpha \rightarrow \alpha$ are polymorphic over all types, mutable or immutable. The above inference algorithm tries to heuristically fix the mutability of all expressions (including functions). However, this behavior can be overridden by explicit annotation. For example, the definition:

$\text{let } allTrue : \forall\alpha.[\alpha \downarrow \text{bool}] \rightarrow \text{bool} = \dots \text{ in } \dots \parallel allTrue : \forall\alpha.[\alpha \downarrow \text{bool}] \rightarrow \text{bool}$
 defines $allTrue$, a function whose argument is mutability-polymorphic over $[\text{bool}]$.

7 Formalization

We now formalize our type system and inference algorithm. Due to space limitations, we will limit our presentation to the following subset of \mathbb{B} , called \mathbb{B}' .

Syntax

Identifiers	$x ::= y \mid z \mid \dots$
Stack Locations	$l ::= l_1 \mid l_2 \mid \dots$
Heap Locations	$\ell ::= \ell_1 \mid \ell_2 \mid \dots$
Locations	$L ::= l \mid \ell$
Values	$v ::= () \mid true \mid false \mid \ell \mid \lambda x.e$
Ivalues	$\mathcal{L} ::= l \mid \ell^\wedge$
Expressions	$e ::= v \mid l \mid e \ e \mid e : \tau \mid e := e$ $\quad \mid \text{dup}(e) \mid e^\wedge$ $\quad \mid \text{if } e \text{ then } e \text{ else } e$ $\quad \mid \text{let}^\kappa x[\tau] = e \text{ in } e$
Let-kinds	$\varkappa ::= - \mid \kappa \mid \psi \mid \forall$
Types	$\tau ::= \alpha \mid \text{unit} \mid \text{bool} \mid \tau \rightarrow \tau$
<i>ref / pointer</i>	$\mid \uparrow\tau$
<i>Mutable type</i>	$\mid \Psi\tau$
Type Scheme	$\sigma ::= \tau \mid \forall\alpha.\sigma$

All the above syntactic forms can be parenthesized without change in meaning. The let-kind “-” is a placeholder for the unkinded (input) let form. A substitution is of Z for Y in X is written using the standard notation: $X[Z/Y]$.

The calculus of \mathbb{B}' defines two kinds of locations: stack locations holding unboxed values, and heap locations holding boxed values. Heap locations are first class values, but stack locations are not. This distinction is sufficient to illustrate the two cases that must be handled by the type system: the types of values passed / copied by value (only) need to be copy compatible, but the types of values passed by reference must be strictly equal. More complex rules can be constructed based on these primitive cases.

In this calculus, we make a distinction between two “kinds” of let expressions — let^ψ : monomorphic, possibly mutable definition, and let^\forall : polymorphic definitions. The two kinds of let expressions have different execution semantics. This distinction is similar to Smith and Volpano’s Polymorphic-C [23]. 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. The kind “-” is a placeholder for the unkinded input let expression. We write let to range over let^ψ and let^\forall .

7.1 Dynamic Semantics of \mathbb{B}

Syntax

Stack	$S ::= \emptyset \mid S, l \mapsto v$
Heap	$H ::= \emptyset \mid H, \ell \mapsto v$

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 relationship $S; H; e \Rightarrow S'; H'; e'$ that denotes transformation in the system state due to a single step of execution. Figure 1 shows the evaluation rules for our core language. We assume that

Rule	Pre-conditions	Evaluation Step
E-Rval	$S(l) = v$	$S; H; l \Rightarrow S; H; v$
EL-Tq		$S; H; e : \tau \Rightarrow S; H; e$
E-Tq		$S; H; e : \tau \Rightarrow S; H; e$
E-Let-Tq		$S; H; \text{let } x : \tau = e_1 \text{ in } e_2 \Rightarrow S; H; \text{let } x = e_1 \text{ in } e_2$
E-App1#	$S; H; e_1 \Rightarrow S'; H'; e'_1$	$S; H; e_1 e_2 \Rightarrow S'; H'; e'_1 e'_2$
E-App2#	$S; H; e_2 \Rightarrow S'; H'; e'_2$	$S; H; v_1 e_2 \Rightarrow S'; H'; v_1 e'_2$
E-App	$l \notin \text{dom}(S)$	$S; H; \lambda x. e v \Rightarrow S, l \mapsto v; H; e[l/x]$
E-If#	$S; H; e \Rightarrow S'; H'; e'$	$S; H; \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow S'; H'; \text{if } e'_1 \text{ then } e_2 \text{ else } e_3$
E-If-True		$S; H; \text{if } \text{true} \text{ then } e_1 \text{ else } e_2 \Rightarrow S; H; e_1$
E-If-False		$S; H; \text{if } \text{false} \text{ then } e_1 \text{ else } e_2 \Rightarrow S; H; e_2$
E-Dup#	$S; H; e \Rightarrow S'; H'; e'$	$S; H; \text{dup}(e) \Rightarrow S'; H'; \text{dup}(e')$
E-Dup	$l \notin \text{dom}(H)$	$S; H; \text{dup}(v) \Rightarrow S; H, l \mapsto v; l$
EL- \wedge #	$S; H; e \Rightarrow S'; H'; e'$	$S; H; e^\wedge \Rightarrow S'; H'; e'^\wedge$
E- \wedge #	$S; H; e \Rightarrow S'; H'; e'$	$S; H; e^\wedge \Rightarrow S'; H'; e'^\wedge$
E- \wedge	$H(l) = v$	$S; H; l^\wedge \Rightarrow S; H; v$
E-:=lhs#	$S; H; e_1 \Rightarrow S'; H'; e'_1$	$S; H; e_1 := e_2 \Rightarrow S'; H'; e'_1 := e_2$
E-:=rhs#	$S; H; e_2 \Rightarrow S'; H'; e'_2$	$S; H; \mathcal{L} := e_2 \Rightarrow S'; H'; \mathcal{L} := e'_2$
E-:=Stack		$S, l \mapsto v_1; H; l := v_2 \Rightarrow S, l \mapsto v_2; H; ()$
E-:=Heap		$S; H, l \mapsto v_1; l^\wedge := v_2 \Rightarrow S; H, l \mapsto v_2; ()$
E-Let#	$S; H; e_1 \Rightarrow S'; H'; e'_1$	$S; H; \text{let } x = e_1 \text{ in } e_2 \Rightarrow S'; H'; \text{let } x = e'_1 \text{ in } e_2$
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 1: Dynamic Semantics

L-Id	L-Hloc	L-Sloc	L-Deref	L-Tq
$\frac{}{\vdash_{\text{val}} x}$	$\frac{}{\vdash_{\text{val}} \ell}$	$\frac{}{\vdash_{\text{val}} l}$	$\frac{}{\vdash_{\text{val}} e^\wedge}$	$\frac{\vdash_{\text{val}} e}{\vdash_{\text{val}} e : \tau}$

Figure 2: Location Semantics

the program is alpha-converted so that there are no name collisions due to inner bindings. Following the theoretical development in [5], we give separate execution semantics for left and right execution (evaluation of expressions that appear on the LHS and RHS of an assignment $e_l := e_r$) denoted by \Rightarrow and \Rightarrow respectively.

Since the E-Dup and E- \wedge 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-:=Stack and E-:=Heap). 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 (they can later be reified independently).

7.2 Static Semantics of \mathbb{B}

Syntax

Types	$\tau ::= \alpha \mid \text{unit} \mid \text{bool} \mid \tau \rightarrow \tau$ $\mid \uparrow\tau \mid \Psi\tau$
Constr. Type	$\rho ::= \tau \mid \tau \setminus \mathcal{C}$
Type Scheme	$\sigma ::= \rho \mid \forall\alpha.\sigma$
Constraints	$c ::= \alpha \preceq : \tau$
Constraint Sets	$\mathcal{C} ::= \emptyset \mid \{\bar{c}\} \mid \mathcal{C} \cup \mathcal{C}$
Binding Environment	$\Gamma ::= \emptyset \mid \Gamma, x \mapsto \sigma$
Store Typing	$\Sigma ::= \emptyset \mid \Sigma, L \mapsto \tau$
Logical Relations	$\Omega ::= \text{true} \mid \text{false} \mid \Omega \wedge \Omega \mid \Omega \vee \Omega$ $\mid \neg\Omega \mid \text{Predicate}(\bar{\Omega})$
Solvable Entities	$\omega ::= \tau \mid \Gamma \mid \Sigma \mid e$

T-Id $\frac{\Gamma(x) = \forall \bar{\alpha}. \tau}{\Gamma; \Sigma \vdash x : \tau[\bar{\tau}_n / \bar{\alpha}]}$	T-Unit $\frac{}{\Gamma; \Sigma \vdash () : \text{unit}}$	T-True $\frac{}{\Gamma; \Sigma \vdash \text{true} : \text{bool}}$	T-False $\frac{}{\Gamma; \Sigma \vdash \text{false} : \text{bool}}$
	T-Hloc $\frac{\Sigma(\ell) = \tau}{\Gamma; \Sigma \vdash \ell : \uparrow \tau}$	T-Sloc $\frac{\Sigma(l) = \tau}{\Gamma; \Sigma \vdash l : \tau}$	T-Lambda $\frac{\Gamma, x \mapsto \tau_1; \Sigma \vdash e : \tau_2}{\Gamma; \Sigma \vdash \lambda x. e : \nabla(\tau_1) \rightarrow \Delta(\tau_2)}$
T-App $\frac{\Gamma; \Sigma \vdash e_1 \preccurlyeq : \tau_2 \rightarrow \tau_0 \quad \Gamma; \Sigma \vdash e_2 \preccurlyeq : \tau_2 \quad \tau_0 \preccurlyeq : \tau'_0}{\Gamma; \Sigma \vdash e_1 e_2 : \tau'_0}$			T-Set $\frac{\Gamma; \Sigma \vdash e_1 \preccurlyeq : \Psi \tau \quad \Gamma; \Sigma \vdash e_2 \preccurlyeq : \tau \quad \vdash_{\text{val}} e_1}{\Gamma; \Sigma \vdash e_1 := e_2 : \text{unit}}$
T-TqExpr $\frac{\Gamma; \Sigma \vdash e : \tau}{\Gamma; \Sigma \vdash (e : \tau) : \tau}$	T-Dup $\frac{\Gamma; \Sigma \vdash e \preccurlyeq : \tau \quad \tau' \preccurlyeq : \tau}{\Gamma; \Sigma \vdash \text{dup}(e) : \uparrow \tau'}$	T-Deref $\frac{\Gamma; \Sigma \vdash e \preccurlyeq : \uparrow \tau}{\Gamma; \Sigma \vdash e^\wedge : \tau}$	
	T-If $\frac{\Gamma; \Sigma \vdash e_1 \preccurlyeq : \text{bool} \quad \Gamma; \Sigma \vdash e_2 \preccurlyeq : \tau \quad \Gamma; \Sigma \vdash e_3 \preccurlyeq : \tau \quad \tau' \preccurlyeq : \tau}{\Gamma; \Sigma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau'}$		
T-Let-M [Tq] $\frac{\Gamma; \Sigma \vdash e_1 \preccurlyeq : \tau_1 \quad \tau \preccurlyeq : \tau_1 \quad \Gamma, x \mapsto \tau; \Sigma \vdash e_2 : \tau_2}{\Gamma; \Sigma \vdash (\text{let}^\psi x[\tau] = e_1 \text{ in } e_2) : \tau_2}$		T-Let-P [Tq] $\frac{\Gamma; \Sigma \vdash e_1 \preccurlyeq : \tau_1 \quad \tau \preccurlyeq : \tau_1 \quad \Gamma; \Sigma; e_1 \vdash_{\text{gen}} \tau \triangleleft \sigma \quad \vdash_{\text{term}} x : \sigma \quad \Gamma, x \mapsto \sigma; \Sigma \vdash e_2 : \tau_2}{\Gamma; \Sigma \vdash (\text{let}^\vee x[\tau] = e_1 \text{ in } e_2) : \tau_2}$	
G-Value $\frac{\text{Value}(e) \quad \text{Immut}(\tau) \quad \bar{\alpha} = \text{ftv}(\tau) \setminus \text{ftv}(\Gamma) \setminus \text{ftv}(\Sigma)}{\Gamma; \Sigma; e \vdash_{\text{gen}} \tau \triangleleft \forall \bar{\alpha}. \tau}$	G-Expansive $\frac{}{\Gamma; \Sigma; e \vdash_{\text{gen}} \tau \triangleleft \tau}$	Q-Loc $\frac{\sigma = \tau}{\vdash_{\text{loc}} x : \sigma}$	Q-Term $\frac{\sigma = \forall \bar{\alpha}. \tau \quad \sigma \neq \tau}{\vdash_{\text{term}} x : \sigma}$

Figure 3: Declarative Type Rules

S-Mut1 $\frac{}{\Psi \tau \preccurlyeq : \tau}$	S-Mut2 $\frac{\tau \preccurlyeq : \tau'}{\Psi \tau \preccurlyeq : \Psi \tau'}$	S-Ref $\frac{\tau = \tau'}{\uparrow \tau \preccurlyeq : \uparrow \tau'}$	S-Fn $\frac{\tau_1 = \tau'_1 \quad \tau_2 = \tau'_2}{\tau_1 \rightarrow \tau_2 \preccurlyeq : \tau'_1 \rightarrow \tau'_2}$
	S-Ref1 $\frac{}{\tau \preccurlyeq : \tau}$	S-Trans $\frac{\tau_0 \preccurlyeq : \tau_1 \quad \tau_1 \preccurlyeq : \tau_2}{\tau_0 \preccurlyeq : \tau_2}$	

Figure 4: Copy Coercion Rules

We represent mathematical properties as: assumption $\stackrel{\text{property}}{=}$ subject.

Figure 4 shows the copy coercion rules. The location semantics (lvalue) rules are shown in figure 2. Figure 3 shows the declarative type rules, rules for type generalization. The standard type judgment $\Gamma; \Sigma \vdash e : \tau$ is understood as: given a binding environment Γ and store typing Σ , the expression e has type τ .

We use the following shorthand notation:

$$\llbracket \Gamma; \Sigma \vdash e \preccurlyeq : \tau \rrbracket \rightsquigarrow \llbracket \Gamma; \Sigma \vdash e : \tau', \tau \preccurlyeq : \tau' \rrbracket$$

Figure 3 shows a generalized declarative type system and associated rules for type generalization. The type judgment $\Gamma; \Sigma \vdash e : \tau$ is understood as: given a binding environment Γ and store typing Σ , the expression e has type τ .

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

Definition 2 (Copy Compatibility). *We define the copy compatibility relationship (\cong) as follows:*

$$\frac{}{\tau \cong \tau} \quad \frac{\tau \cong \tau'}{\tau \cong \Psi\tau'} \quad \frac{\tau \cong \tau'}{\Psi\tau \cong \tau'}$$

In terms of the copy-coercion rules shown in Figure 3, we can define copy compatibility as:

$$\tau_1 \cong \tau_2 \equiv \tau_1 \preccurlyeq : \nabla(\tau_2)$$

The S-Ref rule ensures that copy compatibility does not extend beyond a ref-boundary. Since two function types are equal regardless of the shallow mutability of the argument and return positions, we write all function types in normalized form. The (contravariant) argument type is written in the maximally immutable form (devoid of shallow mutability), and the (covariant) return type is written in the maximally mutable form. This ensures that the “outer” type of a function is maximally permissive with respect to mutability. The S-Fn rule therefore is invariant in terms of its arguments and return types. [This normalization is different from the type displayed to the user, which is discussed in section 6.]

Definition 3 (Max and Min Mutability). *The operators Δ and ∇ increase or decrease the mutability of a type, and are defined as:*

$$\begin{aligned} \Delta(\Psi\tau) &= \Psi\tau \text{ and } \Delta(\tau) = \Psi\tau, \text{ where } \tau \neq \Psi\tau' \\ \nabla(\Psi\tau) &= \tau \text{ and } \nabla(\tau) = \tau, \text{ where } \tau \neq \Psi\tau' \end{aligned}$$

It is obvious that $\forall \tau. \nabla(\tau) \cong \tau \cong \Delta(\tau)$, and $\forall \tau, \tau'. \tau \cong \tau'$ iff $\nabla(\tau) = \nabla(\tau')$ iff $\Delta(\tau) = \Delta(\tau')$.

Definition 4 (Structural Containment). *We define a structural containment relation $\tau \in \omega$ as follows. $\tau \in \tau'$ if τ is structurally present as a part of τ' . $\tau \in e$ if τ is structurally present as a part of e , as a type annotation. $\tau \in \Gamma$ if $\exists x \mapsto \tau' \in \Gamma$ such that $\tau \in \tau'$. $\tau \in \Sigma$ if $\exists L \mapsto \tau' \in \Sigma$ such that $\tau \in \tau'$. We write $\tau \in \overline{\omega}$ if $\tau \in \omega$, for any $\omega \in \{\overline{\omega}\}$.*

Definition 5 (Free Type Variables). *We denote the set of free type variables in a type τ as $\text{ftv}(\tau)$.*

$$\begin{aligned} \text{ftv}(\alpha) &= \alpha \\ \text{ftv}(\text{unit}) &= \{\} \\ \text{ftv}(\text{bool}) &= \{\} \\ \text{ftv}(\uparrow\tau) &= \text{ftv}(\tau) \\ \text{ftv}(\Psi\tau) &= \text{ftv}(\tau) \\ \text{ftv}(\tau_1 \rightarrow \tau_2) &= \text{ftv}(\tau_1) \cup \text{ftv}(\tau_2) \\ \text{ftv}(\overline{\tau_i}) &= \bigcup \text{ftv}(\tau_i) \\ \text{ftv}(\sigma) &= \text{ftv}(\overline{\alpha}) \cup \text{ftv}(\tau), \text{ where } \sigma = \nabla\overline{\alpha}.\tau \\ \text{ftv}(\Gamma) &= \bigcup \text{ftv}(\sigma_i), \forall x \mapsto \sigma_i \in \Gamma \\ \text{ftv}(\Sigma) &= \bigcup \text{ftv}(\tau_i), \forall L \mapsto \tau_i \in \Sigma \\ \text{ftv}(e) &= \bigcup \text{ftv}(\tau_i), \forall \tau_i \in e \\ \text{ftv}(\overline{\omega}) &= \bigcup \text{ftv}(\omega). \end{aligned}$$

Definition 6 (Value Restriction). *We define some definitions used in the enforcement of value restriction in Figure 3.*

$$\begin{aligned}
\text{Value}(v) &= \text{true} \\
\text{Value}(x) &= \text{true} \\
\text{Value}(\ell) &= \text{true} \\
\text{Value}(l) &= \text{true} \\
\text{Value}(e : \tau) &= \text{Value}(e) \\
\text{Value}(\text{dup}(e)) &= \text{Value}(e) \\
\text{Value}(e^\wedge) &= \text{Value}(e) \\
\text{Value}(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \\
&= \text{Value}(e_1) \wedge \text{Value}(e_2) \wedge \text{Value}(e_3) \\
\text{Value}(\text{let } x = e_1 \text{ in } e_2) &= \text{Value}(e_1) \wedge \text{Value}(e_2) \\
\text{Value}(e \text{ (otherwise)}) &= \text{false}
\end{aligned}$$

$$\text{Expansive}(e) = \neg \text{Value}(e)$$

$$\begin{aligned}
\text{Immutable}(\text{unit}) &= \text{true} \\
\text{Immutable}(\text{bool}) &= \text{true} \\
\text{Immutable}(\tau_1 \rightarrow \tau_2) &= \text{true} \\
\text{Immutable}(\uparrow\tau) &= \text{Immutable}(\tau) \\
\text{Immutable}(\tau \text{ (otherwise)}) &= \text{false}
\end{aligned}$$

$$\begin{aligned}
\text{Immut}(\text{unit}) &= \text{true} \\
\text{Immut}(\text{bool}) &= \text{true} \\
\text{Immut}(\alpha) &= \text{true} \\
\text{Immut}(\tau_1 \rightarrow \tau_2) &= \text{true} \\
\text{Immut}(\uparrow\tau) &= \text{Immut}(\tau) \\
\text{Immut}(\forall \bar{\alpha}. \tau) &= \text{Immut}(\tau) \\
\text{Immut}(\tau \text{ (otherwise)}) &= \text{false}
\end{aligned}$$

$$\text{Mut}(\tau) = \neg \text{Immut}(\tau)$$

Definition 7 (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 $\Gamma; \Sigma \vdash H + S$ if*

1. $\text{dom}(\Sigma) = \text{dom}(H) \cup \text{dom}(S)$
2. $\forall \ell \in \text{dom}(H), \Gamma; \Sigma \vdash H(\ell) \preceq: \Sigma(\ell)$
3. $\forall l \in \text{dom}(S), \Gamma; \Sigma \vdash S(l) \preceq: \Sigma(l)$

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

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

Proof. Immediate from the definition of typing relation. □

Lemma 2 (Inversion of Copy Coercion). *1. If $\tau \preceq: \text{bool}$ then $\tau = \text{bool}$ or $\tau = \Psi\text{bool}$.*

2. If $\tau \preceq: \text{unit}$ then $\tau = \text{unit}$ or $\tau = \Psi \text{bool}$.
3. If $\tau \preceq: \tau_1 \rightarrow \tau_2$ then $\tau = \tau_1 \rightarrow \tau_2$ or $\tau = \Psi(\tau_1 \rightarrow \tau_2)$.
4. If $\tau \preceq: \uparrow\tau'$ then $\tau = \uparrow\tau'$ or $\tau = \Psi\uparrow\tau'$.
5. If $\tau \preceq: \Psi\tau'$ then $\tau = \Psi\tau''$ such that $\tau'' \preceq: \tau'$.
6. If $\tau \preceq: \Psi\tau'$ then $\tau \preceq: \tau'$.

Proof. By induction on the copy coercion derivation. □

Lemma 3 (Canonical Forms). 1. If v is a value, and $\Gamma; \Sigma \vdash v \preceq: \text{unit}$, then, v is ().

2. If v is a value, and $\Gamma; \Sigma \vdash v \preceq: \text{bool}$, then, v is either true or false.
3. If v is a value, and $\Gamma; \Sigma \vdash v \preceq: \uparrow\tau$, then, v is ℓ , $\ell \in \text{dom}(\Sigma)$.
4. If v is a value, and $\Gamma; \Sigma \vdash v \preceq: \tau_1 \rightarrow \tau_2$, then, v is $\lambda x.e$.

Proof. By induction on the derivation of $\Gamma; \Sigma \vdash v \preceq: \tau$.

If $\Gamma; \Sigma \vdash v \preceq: \text{bool}$, we have $\Gamma; \Sigma \vdash v : \tau$ and $\tau \preceq: \text{bool}$ by Inversion of copy coercion relation, $\tau = \text{bool}$ or $\tau = \Psi \text{bool}$. If $\tau = \text{bool}$, it is clear that the final rule in the derivation must be T-True, or T-False, in which case the result is immediate. The case $\tau = \Psi \text{bool}$ cannot happen because there is no rule that derives a mutable type for a value, and we assume that the induction hypothesis $\Gamma; \Sigma \vdash v : \tau$ holds.

Other cases of the lemma are similar. □

Lemma 4 (Progress). If e is a closed, well typed term, that is, $\emptyset; \Sigma \vdash e : \tau$ for some τ and Σ , given any heap H and stack S such that $\Gamma; \Sigma \vdash H + S$,

1. If $\vdash_{\text{val}} e$, then e is either a valid lvalue \mathcal{L} (that is, $\mathcal{L} = l$, $l \in \text{dom}(S)$ or $\mathcal{L} = \ell^\wedge$, $\ell \in \text{dom}(H)$) 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 typing derivation.

1. Case T-Unit, T-True, T-False, T-Hloc, T-Lambda: (Values) Result is immediate for right execution, and cannot happen for right execution
2. Case T-Id: cannot happen, there is no execution rule for variables.
3. Case T-Sloc: Immediate for left execution. Right execution and can always continue with E-Rval rule as the stack is well typed ($\Gamma; \Sigma \vdash H + S$).
4. Case T-App: Only right execution is possible, no application is well typed as an lvalue. We have: $e = e_1 e_2$, $e_1 \preceq: \tau_1 \rightarrow \tau_2$, and $e_2 \preceq: \tau_1$. If e_1 or e_2 is not a value, we can take E-App1# or E-App2#. Otherwise, when both e_1 and e_2 are values, by canonical forms lemma, e_1 is of the form $\lambda x.e'$, and 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: $e = e_1 := e_2$, $e_1 \preceq: \tau_1 \rightarrow \tau_2$, $\Gamma; \Sigma \vdash e_1 \preceq: \Psi\tau$, $\Gamma; \Sigma \vdash e_2 \preceq: \tau$, and $\vdash_{\text{val}} e_1$. If e_1 not an lvalue, since we have $\vdash_{\text{val}} e_1$ we can take E-:=lhs# by induction hypothesis. Similarly, if e_2 is not a value, we can take E-:=rhs#. Finally, if $e_1 = \mathcal{L}$ and $e_1 = v$, we can take the step E-:=Stack or E-:=Heap as applicable.
7. Case T-Dup: Only right execution is permitted, and can take E-Dup# or E-Dup as applicable.

8. Case T-Deref: We have: $e = e_1^\wedge$, and $\Gamma; \Sigma \vdash e_1 \preceq: \uparrow\tau$. Execution can take $EL^\wedge\#$ or $E^\wedge\#$ as applicable if e_1 is not a value. If $e_1 \preceq: \uparrow\tau$ is a value, then, from the canonical forms lemma, $e_1 = \ell$, $\ell \in \text{dom}(\Sigma)$. 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^\wedge since the heap is well typed ($\Gamma; \Sigma \vdash H + S$).
9. Case T-Let-M: Only right execution is applicable. We have: $e = (\text{let}^\psi x = e_1 \text{ in } e_2)$, $\tau \preceq: \tau_1$, $\Gamma; \Sigma; e_1 \vdash_{\text{gen}} \tau < \sigma$, and $\vdash_{\text{loc}} x : \sigma$, and $\Gamma, x \mapsto \sigma; \Sigma \vdash e_2 : \tau_2$. If e_1 is not a value, we can take $E\text{-Let}\#$. Otherwise, we can take $E\text{-Let-M}$.
10. Case T-Let-P: Only right execution is applicable. We have: $e = (\text{let}^\psi x = e_1 \text{ in } e_2)$, $\tau \preceq: \tau_1$, $\Gamma; \Sigma; e_1 \vdash_{\text{gen}} \tau < \sigma$, and $\vdash_{\text{term}} x : \sigma$, and $\Gamma, x \mapsto \sigma; \Sigma \vdash e_2 : \tau_2$. If e_1 is not a value, we can take $E\text{-Let}\#$. Otherwise, can take $E\text{-Let-P}$.
11. Case T-TqExpr and Case T-Let-M-Tq, T-Let-P-Tq are similar.

□

Lemma 5 (Weakening). *We will write $\Gamma; \Sigma \vdash e : \tau < \sigma$ as a shorthand for $\Gamma; \Sigma \vdash e : \tau$, and $\Gamma; \Sigma; e \vdash_{\text{gen}} \tau < \sigma$.*

1. If $\Gamma; \Sigma \vdash e : \tau$ then,
 - (a) If $\Gamma' \supseteq \Gamma$ then $\Gamma'; \Sigma \vdash e : \tau$.
 - (b) If $\Sigma' \supseteq \Sigma$ then $\Gamma; \Sigma' \vdash e : \tau$.
2. If $\Gamma; \Sigma \vdash e : \tau < \sigma$, $\sigma = \forall \bar{\alpha}. \tau$
 - (a) If $\Gamma' \supseteq \Gamma$ and $\text{ftv}(\Gamma') \cap \text{ftv}(\bar{\alpha}) = \emptyset$ then $\Gamma'; \Sigma \vdash e : \tau < \sigma$.
 - (b) If $\Sigma' \supseteq \Sigma$ and $\text{ftv}(\Sigma') \cap \text{ftv}(\bar{\alpha}) = \emptyset$ then $\Gamma; \Sigma' \vdash e : \tau < \sigma$.

Proof. Straightforward induction on the typing derivation. □

Lemma 6 (Value Substitution). *If $\Gamma, x : \sigma; \Sigma \vdash e : \tau$, $\text{Immut}(\sigma)$ and $\Gamma; \Sigma \vdash v : \tau_v$, and $\Gamma; \Sigma; e \vdash_{\text{gen}} \tau_v < \sigma$ then $\Gamma; \Sigma \vdash e[v/x] : \tau$*

Proof. By induction on the typing derivation of $\Gamma, x : \sigma; \Sigma \vdash e : \tau$. We proceed by case analysis on the final step of the derivation.

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

There are two sub cases to consider. If $x = y$, then, $y[v/x] = v$, and the result type τ is an instantiation of the type scheme σ . One of the assumptions of the lemma states that $\Gamma; \Sigma \vdash v : \tau_v < \sigma$. That is, $\tau \sqsubseteq \sigma$, and we can infer any more-specific type (and in particular the type being instantiated at the T-Id rule) instead for this substitution of the expression v . Therefore, we have $\Gamma; \Sigma \vdash e[v/x] : \tau$.

If $x \neq y$, then $y[v/x] = y$, and the result is immediate.
2. Case T-Lambda: We have $e = \lambda y. e'$, and $\tau = \tau_1 \rightarrow \tau_2$, and $\Gamma, x : \sigma, y : \tau_1; \Sigma \vdash e' : \tau_2$.

We can assume that $x \neq y$. Since it is clear that the type τ_1 of y can either use variables already in Γ or fresh type variables, we know that $\text{ftv}(\Gamma, y : \tau_1) \cap \text{ftv}(\sigma) = \emptyset$. Thus, by weakening lemma, we have: $\Gamma, y : \tau_1; \Sigma \vdash v : \tau_v < \sigma$, and, by induction hypothesis, $\Gamma, y : \tau_1; \Sigma \vdash e'[v/x] : \tau_2$. Finally, by the T-Lambda rule, we have: $\Gamma; \Sigma \vdash \lambda x_y. (e'[v/x]) : \tau_1 \rightarrow \tau_2$, and thus $\Gamma; \Sigma \vdash \lambda x_y. e'[v/x] : \tau_1 \rightarrow \tau_2$, which is the desired result.
3. T-Set case is similar, except that the substitution cannot happen on the LHS of an assignment, since we do not perform substitution of mutable values.
4. Other cases are similar.

□

Lemma 7 (Location Substitution). *If $\Gamma, x : \tau_0; \Sigma \vdash e : \tau$, and for some $\Sigma' \supseteq \Sigma$, $\Sigma(l) = \tau_0$, then $\Gamma; \Sigma' \vdash e[l/x] : \tau$.*

Proof. By induction on the typing derivation of $\Gamma, x : \tau; \Sigma \vdash e : \tau$, similar to lemma 6. □

Lemma 8 (Stack and Heap Assignment). 1. *If $\Gamma; \Sigma \vdash H, \ell \mapsto v + S$, and $\Sigma(\ell) \preceq \tau$, and $\Gamma; \Sigma \vdash v' : \tau$, then, $\Gamma; \Sigma \vdash H, \ell \mapsto v' + S$.*

2. *Similarly, if $\Gamma; \Sigma \vdash H + S, l \mapsto v$ and $\Sigma(l) \preceq \tau$, and $\Gamma; \Sigma \vdash v' : \tau$, then, $\Gamma; \Sigma \vdash H + S, l \mapsto v'$.*

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

Lemma 9 (Preservation). *If $\Gamma; \Sigma \vdash e : \tau$ and $\Gamma; \Sigma \vdash H + S$ then,*

1. *If $S; H; e \Rightarrow S'; H'; e'$, then, there exists a $\Sigma' \supseteq \Sigma$ such that $\Gamma; \Sigma' \vdash e' : \tau$ and $\Gamma; \Sigma' \vdash H' + S'$.*

2. *If $S; H; e \Rightarrow S'; H'; e'$, there exists a $\Sigma' \supseteq \Sigma$ such that $\Gamma; \Sigma' \vdash e' \preceq \tau'$, $\Gamma; \Sigma' \vdash H' + S'$ and $\nabla(\tau) = \nabla(\tau')$.*

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

1. Case T-Id, T-True, T-False, T-Hloc, T-Lambda cannot happen.

2. Case T-Sloc: Only right execution is applicable. We have: $e = l$ and $\Sigma(l) : \tau$. The only applicable step is E-Rval, and we have $e' = S(l)$. From the definition of stack typing, we have: $S(l) \preceq \Sigma(l)$ and thus $e' \preceq \tau$ which implies $\nabla(\tau) = \nabla(\tau')$.

3. Case T-App: $e = e_1 e_2$, and $\Gamma; \Sigma \vdash e_1 \preceq \tau_2 \rightarrow \tau_0$, and $\Gamma; \Sigma \vdash e_2 \preceq \tau_2$, and $\tau_0 \preceq \tau$, and $e : \tau$ where $\tau_2 = \nabla(\tau'_2)\tau_0 = \Delta(\tau'_0)$ and .

This cannot happen for left execution. For right execution, we proceed by further case analysis of the applicable execution rules for $S; H; e \Rightarrow S'; H'; e'$.

(a) Case E-App1#: We have: $S; H; e_1 \Rightarrow S'; H'; e'_1$ and $e' = e'_1 e_2$. By induction hypothesis, we have: $\Gamma; \Sigma' \vdash e'_1 \preceq \tau_2 \rightarrow \tau_0$ for some $\Sigma' \supseteq \Sigma$. One of the assumptions of the T-App rule states that $\Gamma; \Sigma \vdash e_2 \preceq \tau_2$, and by weakening lemma, we have, $\Gamma; \Sigma' \vdash e_2 \preceq \tau_2$. Finally, by the T-App rule, we conclude that $(e'_1 e_2) : \tau$.

(b) Case E-App2#: Similar to the previous sub-case.

(c) Case E-App: We have: $e_1 = \lambda x. e_0$ and $e_2 = v$ and $e' = e_0[l/x]$ and $S, l \mapsto v; H; e_1 \Rightarrow S'; H'; e'_1$.

By the inversion lemma for $\lambda x. e_0$ we have $\Gamma, \Sigma \vdash e_0 : \tau'_0$.

Further from location substitution lemma, we have $\Gamma; \Sigma' \vdash e_0[l/x] : \tau'_0$ where $\Sigma' \supseteq \Sigma$ and $\Sigma(l) : \tau'_2$.

Thus, we have $\tau_0 \preceq \tau'_0$ and $\tau_0 \preceq \tau$. Therefore, it is clear that $\nabla(\tau'_0) = \nabla(\tau)$.

4. Case T-Set: $e = e_1 := e_2$, and $\Gamma; \Sigma \vdash e_1 \preceq \Psi\tau$, and $\Gamma; \Sigma \vdash e_2 \preceq \tau \upharpoonright_{\text{val}} e_1$

If the step taken is E-:=#lhs or E-:=#rhs, the result follows from the induction hypothesis and T-Set rule (as in the case of T-App). If the step taken is E-:=Stack or E-:=Heap, the result follows from the stack and heap assignment lemma.

5. Case T-Deref: We have: $e = e' \hat{\ }^{\#}$ and $\Gamma; \Sigma \vdash e' \preceq \uparrow\tau$. If the step taken is EL- $\hat{\ }^{\#}$ or E- $\hat{\ }^{\#}$, the result follows from induction hypothesis and T-Deref rule. If the step taken is E- $\hat{\ }^{\#}$ (right execution only) e' is a value, and from canonical forms lemma, we know that $e' = \ell$ and $\ell \in \text{dom}(\Sigma)$ and the result follows from the fact that $\Gamma; \Sigma \vdash H + S$.

6. Case T-Let-P: Right execution only. We have: $e = (\text{let}^{\forall} x = e_1 \text{ in } e_2)$ and $\Gamma; \Sigma \vdash e_1 \preceq \tau_1$ and $\tau \preceq \tau_1$ and $\Gamma; \Sigma; e_1 \upharpoonright_{\text{gen}} \tau \preceq \sigma$ and $\Gamma, x \mapsto \sigma; \Sigma \vdash e_2 : \tau_2$. There are two sub-cases to consider:

- (a) If we take step E-Let#, $S; H; e_1 \Rightarrow S'; H'; e'_1$ and $e' = (\text{let}^\forall x = e'_1 \text{ in } e_2)$. If $e = v$ It is clear that $\text{Value}(e_1)$ implies $\text{Value}(e'_1)$. Now, the result follows from the induction hypothesis and the E-Let-P rule.
 - (b) If we take the step E-Let-P, $e = (\text{let}^\forall x = v \text{ in } e_2)$ Since $x : \sigma$ has a polymorphic type, (that is, $\sigma = \forall \bar{\alpha}. \tau$) we know that $\text{Immut}(\tau)$. Also, from canonical forms lemma, all values have an immutable type. Therefore, $\tau = \tau_1$. Now, the result follows from value substitution lemma.
7. Case T-Let-M: Similar to T-Let-P, except that we should always use the GEN-EXPANSIVE rule during generalization, and use the location substitution lemma instead of the value substitution lemma
8. Cases T-If, T-Dup, T-TqExpr, and T-Let-M-Tq T-Let-P-Tq are similar.

□

Definition 8 (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 1 (Type Soundness). If $\emptyset; \Sigma \vdash e : \tau$ and $\Gamma; \Sigma \vdash H + S$ and $S; H; e \xrightarrow{*} S'; H'; e'$ then $S'; H'; e'$ is not stuck. That is, execution of a well typed expression cannot lead to a stuck state. Here, $\xrightarrow{*}$ represents the reflexive-transitive-closure of \Rightarrow .

Proof. By straightforward induction on the length of $\xrightarrow{*}$. If $e = v$, proof is immediate. Otherwise, from Lemma 4 (Progress), we know that we can take at least one step forward. Further, from Lemma 9 (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. □

7.3 Heuristic Type Inference

Syntax

Types	$\tau ::= \alpha \mid \text{unit} \mid \text{bool} \mid \lfloor \tau \rfloor \rightarrow \lceil \tau \rceil$ $\mid \uparrow \tau \mid \Psi \tau$
Maybe Type	$\mid \tau \downarrow \tau'$
Uunct. Types	$\varsigma ::= \alpha \mid \text{unit} \mid \text{bool} \mid \lfloor \varsigma \rfloor \rightarrow \lceil \varsigma \rceil$ $\mid \uparrow \varsigma \mid \Psi \varsigma$
Type Scheme	$\sigma ::= \tau \mid \forall \alpha. \sigma$
Constraints	$c ::= \tau = \tau \mid \tau \cong \tau \mid \tau \preceq \tau$
Constraint Sets	$\mathcal{C} ::= \emptyset \mid \{\bar{c}\} \mid \mathcal{C} \cup \mathcal{C}$
Substitutions	$\theta ::= \langle \rangle \mid [\alpha \mapsto \tau] \mid [\bar{k} \mapsto \bar{\kappa}] \mid \theta \circ \theta$

The application of a substitution θ on X is written as $\theta\langle X \rangle$. As a matter of notational convenience, we write: $\theta_{a,b}$ to mean $\theta_a \circ \theta_b$. Note that $\theta_{a,b}\langle x \rangle = \theta_a \circ \theta_b\langle x \rangle = \theta_a\langle \theta_b\langle x \rangle \rangle = \theta_b\langle \theta_a\langle x \rangle \rangle$.

Definition 9 (Meta Constructors). $\lfloor \tau \rfloor$ and $\lceil \tau \rceil$ are "meta-constructors" which (respectively) minimize and maximize the mutability of a type, but are interpreted lazily. The meta-constructors are idempotent.

Note that in our type system, we have restricted meta types to be syntactically present only as part of function types.

Definition 10 (FTVs (Extension)). We enhance the definition of $\text{ftv}(\dots)$ in Definition 5 as follows:

$$\begin{aligned}
 & \dots \\
 \text{ftv}(\lfloor \tau_1 \rfloor \rightarrow \lceil \tau_2 \rceil) &= \text{ftv}(\tau_1) \cup \text{ftv}(\tau_2) \\
 \text{ftv}(\tau_1 = \tau_2) &= \text{ftv}(\tau_1) \cup \text{ftv}(\tau_2) \\
 \text{ftv}(\tau_1 \cong \tau_2) &= \text{ftv}(\tau_1) \cup \text{ftv}(\tau_2) \\
 \text{ftv}(\tau_1 \preceq \tau_2) &= \text{ftv}(\tau_1) \cup \text{ftv}(\tau_2) \\
 \text{ftv}(\mathcal{C}) &= \bigcup \text{ftv}(c_i), \forall c_i \in \mathcal{C}
 \end{aligned}$$

M-Mut1 $\frac{}{\Psi\tau \triangleleft: \tau}$	M-Mut2 $\frac{\tau \triangleleft: \tau'}{\Psi\tau \triangleleft: \Psi\tau'}$	M-May1 $\frac{\Im(\tau) \triangleleft: \nabla(\tau')}{\alpha \downarrow \tau \triangleleft: \tau'}$	M-May2 $\frac{\tau \neq \alpha \quad \tau \triangleleft: \tau'}{\tau \downarrow \tau' \triangleleft: \tau''}$	M-May3 $\frac{\tau' \neq \alpha \quad \tau \triangleleft: \tau'}{\tau \triangleleft: \tau' \downarrow \tau''}$
M-Ref $\frac{\tau = \tau'}{\uparrow\tau \triangleleft: \uparrow\tau'}$	M-Fn $\frac{\tau_1 \triangleleft: \tau_1'' \quad \tau_1' \triangleleft: \tau_1'' \quad \tau_2 \triangleleft: \tau_2'' \quad \tau_2' \triangleleft: \tau_2''}{[\tau_1] \rightarrow [\tau_2] \triangleleft: [\tau_1'] \rightarrow [\tau_2']}$		M-Refl $\frac{}{\tau \triangleleft: \tau}$	M-Trans $\frac{\tau_0 \triangleleft: \tau_1 \quad \tau_1 \triangleleft: \tau_2}{\tau_0 \triangleleft: \tau_2}$
J-Id $\frac{\Gamma(x) = \nabla\bar{\alpha}.\tau \quad \Vdash \{\Gamma, \Sigma, \tau[\bar{\tau}_n/\bar{\alpha}]\}}{\Gamma; \Sigma \Vdash x : \tau[\bar{\tau}_n/\bar{\alpha}]}$	J-Unit $\frac{\Vdash \{\Gamma, \Sigma\}}{\Gamma; \Sigma \Vdash () : \text{unit}}$	J-True $\frac{\Vdash \{\Gamma, \Sigma\}}{\Gamma; \Sigma \Vdash \text{true} : \text{bool}}$	J-False $\frac{\Vdash \{\Gamma, \Sigma\}}{\Gamma; \Sigma \Vdash \text{false} : \text{bool}}$	
	J-Hloc $\frac{\Sigma(\ell) = \tau \quad \Vdash \{\Gamma, \Sigma\}}{\Gamma; \Sigma \Vdash \ell : \uparrow\tau}$	J-Sloc $\frac{\Sigma(l) = \tau \quad \Vdash \{\Gamma, \Sigma\}}{\Gamma; \Sigma \Vdash l : \tau}$	J-Lambda $\frac{\Gamma, x \mapsto \tau_1; \Sigma \Vdash e : \tau_2 \quad \Vdash \{\Gamma, \Sigma, e, \tau_1, \tau_2\}}{\Gamma; \Sigma \Vdash \lambda x.e : [\tau_1] \rightarrow [\tau_2]}$	
J-App $\frac{\Gamma; \Sigma \Vdash e_1 : \tau_1 \quad \tau_1 \triangleleft: [\tau_a] \rightarrow [\tau_r] \quad \Gamma; \Sigma \Vdash e_2 : \tau_2 \quad \tau_2 \triangleleft: \Im(\tau_a) \quad \tau_r \triangleleft: \Im(\tau) \quad \Vdash \{\Gamma, \Sigma, e_1, e_2, \tau, \tau_1, \tau_2, [\tau_a] \rightarrow [\tau_r]\}}{\Gamma; \Sigma \Vdash e_1 e_2 : \tau}$				
J-Set $\frac{\Gamma; \Sigma \vdash e_1 : \Psi\tau_1 \quad \tau_1 \triangleleft: \tau \quad \Gamma; \Sigma \Vdash e_2 : \tau_2 \quad \tau_2 \triangleleft: \tau \quad \Vdash_{\text{flat}} e_1 \quad \Vdash \{\Gamma, \Sigma, e_1, e_2, \tau, \tau_1, \tau_2\}}{\Gamma; \Sigma \Vdash e_1 := e_2 : \text{unit}}$			J-TqExpr $\frac{\Gamma; \Sigma \Vdash e : \tau \quad \Vdash \{\Gamma, \Sigma, e, \tau\}}{\Gamma; \Sigma \Vdash (e : \tau) : \tau}$	
J-If $\frac{\Gamma; \Sigma \Vdash e_1 : \tau_1 \quad \tau_1 \triangleleft: \text{bool} \quad \Gamma; \Sigma \Vdash e_2 : \tau_2 \quad \tau_2 \triangleleft: \tau \quad \Gamma; \Sigma \Vdash e_3 : \tau_3 \quad \tau_3 \triangleleft: \tau \quad \tau' \triangleleft: \tau \quad \Vdash \{\Gamma, \Sigma, e_1, e_2, e_3, \tau, \tau', \tau_1, \tau_2, \tau_3\}}{\Gamma; \Sigma \Vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau'}$				
J-Dup $\frac{\Gamma; \Sigma \Vdash e : \tau' \quad \tau' \triangleleft: \tau \quad \tau'' \triangleleft: \tau \quad \Vdash \{\Gamma, \Sigma, e, \tau, \tau', \tau''\}}{\Gamma; \Sigma \Vdash \text{dup}(e) : \uparrow\tau''}$		J-Let-M [Tq] $\frac{\Gamma; \Sigma \Vdash e_1 : \tau_1 \quad \tau_1 \triangleleft: \tau \quad \triangleleft: \tau \quad \Gamma, x \mapsto ; \Sigma \Vdash e_2 : \tau_2 \quad \Vdash \{\Gamma, \Sigma, e_1, e_2, \tau, \tau_1, \tau_2\}}{\Gamma; \Sigma \Vdash (\text{let}^\Psi x[:] = e_1 \text{ in } e_2) : \tau_2}$		
J-Deref $\frac{\Gamma; \Sigma \Vdash e : \tau' \quad \tau' \triangleleft: \uparrow\tau \quad \Vdash \{\Gamma, \Sigma, e, \tau, \tau'\}}{\Gamma; \Sigma \Vdash e^\wedge : \tau}$		J-Let-P [Tq] $\frac{\Gamma; \Sigma \Vdash e_1 : \tau_1 \quad \tau_1 \triangleleft: \tau \quad \triangleleft: \tau \quad \Gamma; \Sigma; e_1 \Vdash_{\text{gen}} \triangleleft \sigma \quad \Vdash_{\text{term}} x : \sigma \quad \Gamma, x \mapsto \sigma; \Sigma \Vdash e_2 : \tau_2 \quad \Vdash \{\Gamma, \Sigma, e_1, e_2, \tau, \tau_1, \tau_2\}}{\Gamma; \Sigma \Vdash (\text{let}^\forall x[:] = e_1 \text{ in } e_2) : \tau_2}$		

Figure 5: Intermediate Declarative System

I-Unit $\frac{}{\Gamma; \Sigma \vdash_{\bar{i}} () : \text{unit}}$	I-Hloc $\frac{\Sigma(\ell) = \tau}{\Gamma; \Sigma \vdash_{\bar{i}} \ell : \uparrow\tau}$	I-Sloc $\frac{\Sigma(l) = \tau}{\Gamma; \Sigma \vdash_{\bar{i}} l : \tau}$	I-Id $\frac{\Gamma(x) = \forall \bar{\alpha}. \tau \quad \frac{}{\bar{i}} \bar{\beta}}{\Gamma; \Sigma \vdash_{\bar{i}} x : \tau[\bar{\beta}/\bar{\alpha}]}$
I-True $\frac{}{\Gamma; \Sigma \vdash_{\bar{i}} \text{true} : \text{bool}}$	I-False $\frac{}{\Gamma; \Sigma \vdash_{\bar{i}} \text{false} : \text{bool}}$	I-Lambda $\frac{\Gamma, x \mapsto \alpha; \Sigma \vdash_{\bar{i}} e : \tau \quad \theta \quad \frac{}{\bar{i}} \alpha}{\Gamma; \Sigma \vdash_{\bar{i}} \lambda x. e : [\theta\langle\alpha\rangle] \rightarrow [\tau]}$	
I-TqExpr $\frac{\Gamma; \Sigma \vdash_{\bar{i}} e : \tau' \parallel \theta_i \quad \theta_u \vdash_{\text{unf}} \tau' = \theta_i\langle\tau\rangle}{\Gamma; \Sigma \vdash_{\bar{i}} (e : \tau) : \theta_u\langle\tau'\rangle \parallel \theta_{i,u}}$		I-Dup $\frac{\Gamma; \Sigma \vdash_{\bar{i}} e : \tau \parallel \theta \quad \tau' = \alpha \downarrow \tau \quad \frac{}{\bar{i}} \alpha}{\Gamma; \Sigma \vdash_{\bar{i}} \text{dup}(e) : \uparrow\tau' \parallel \theta}$	
I-App $\frac{\Gamma; \Sigma \vdash_{\bar{i}} e_1 : \tau_1 \parallel \theta_1 \quad \theta_1 \parallel \Gamma; \Sigma \vdash_{\bar{i}} e_2 : \tau_2 \parallel \theta_2 \quad \theta_f \vdash_{\text{unf}} \theta_2\langle\tau_1\rangle = \beta \downarrow ([\delta] \rightarrow [\alpha]) \quad \theta_a \vdash_{\text{unf}} \tau_2 = \gamma \downarrow \nabla(\theta_f\langle\delta\rangle) \quad \tau = \varepsilon \downarrow \nabla(\theta_{a,f}\langle\alpha\rangle) \quad \frac{}{\bar{i}} \alpha \beta \gamma \delta \varepsilon}{\Gamma; \Sigma \vdash_{\bar{i}} e_1 e_2 : \tau \parallel \theta_{1.2.f.a}}$			
I-If $\frac{\Gamma; \Sigma \vdash_{\bar{i}} e_1 : \tau_1 \parallel \theta_1 \quad \theta_1 \parallel \Gamma; \Sigma \vdash_{\bar{i}} e_2 : \tau_2 \parallel \theta_2 \quad \theta_{1.2} \parallel \Gamma; \Sigma \vdash_{\bar{i}} e_3 : \tau_3 \parallel \theta_3 \quad \frac{}{\bar{i}} \alpha \beta \gamma \delta \varepsilon \quad \theta_t \vdash_{\text{unf}} \theta_3\langle\tau_2\rangle = \alpha \downarrow \beta \quad \theta_f \vdash_{\text{unf}} \theta_t\langle\tau_3\rangle = \gamma \downarrow \theta_c\langle\beta\rangle \quad \theta_c \vdash_{\text{unf}} \theta_{2.3.t.f}\langle\tau_1\rangle = \delta \downarrow \text{bool} \quad \theta = \theta_{1.2.3.c.t.f}}{\Gamma; \Sigma \vdash_{\bar{i}} \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \varepsilon \downarrow \text{join}(\theta\langle\tau_2\rangle, \theta\langle\tau_3\rangle) \parallel \theta}$			
I-Set $\frac{\Gamma; \Sigma \vdash_{\bar{i}} e_1 : \tau_1 \parallel \theta_1 \quad \theta_1 \parallel \Gamma; \Sigma \vdash_{\bar{i}} e_2 : \tau_2 \parallel \theta_2 \quad \frac{}{\bar{i}} \alpha \quad \theta_l \vdash_{\text{unf}} \theta_2\langle\tau_1\rangle = \Psi \alpha \quad \theta_r \vdash_{\text{unf}} \theta_l\langle\tau_2\rangle = \beta \downarrow \theta_{2.l}\langle\tau_1\rangle \quad \frac{}{\text{ival}} e_1}{\Gamma; \Sigma \vdash_{\bar{i}} e_1 := e_2 : \text{unit} \parallel \theta_{1.2.l.r}}$		I-Deref $\frac{\Gamma; \Sigma \vdash_{\bar{i}} e : \tau \parallel \theta_i \quad \frac{}{\bar{i}} \alpha \beta \quad \theta_u \vdash_{\text{unf}} \tau = \beta \downarrow \uparrow \alpha}{\Gamma; \Sigma \vdash_{\bar{i}} e^{\wedge} : \theta_{i,u}\langle\alpha\rangle \parallel \theta_{i,u}}$	
I-Let [Tq] $\frac{\Gamma; \Sigma \vdash_{\bar{i}} e_1 : \tau_1 \parallel \theta_1 \quad \theta_u \vdash_{\text{unf}} \theta_1\langle\tau\rangle = \alpha \downarrow \tau_1 \quad \theta_s; x; e_2 \vdash_{\text{sol}} \theta_{1.u}\langle\tau\rangle : \gg \tau' \quad \theta = \theta_{1.u.s} \quad \theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle; \theta\langle e_1\rangle \vdash_{\text{gen}} \tau' \leq \sigma \quad x : \sigma \vdash_{\text{ch}} \kappa = \varkappa \quad \theta \parallel \Gamma, x \mapsto \sigma; \Sigma \vdash_{\bar{i}} e_2 : \tau_2 \parallel \theta_2 \quad \frac{}{\bar{i}} \alpha}{\Gamma; \Sigma \vdash_{\bar{i}} \text{let}^{\psi} x : \tau = e_1 \text{ in } e_2 : \tau_2 \parallel \theta \circ [\kappa \mapsto \varkappa] \circ \theta_2}$			
$\frac{\frac{}{\text{loc}} x : \sigma}{x : \sigma \vdash_{\text{ch}} \kappa = \psi}$		$\frac{\frac{}{\text{term}} x : \sigma}{x : \sigma \vdash_{\text{ch}} \kappa = \forall}$	

Figure 6: Type Inference Rules.

U-Refl $\frac{}{\emptyset \vdash_{\text{unf}} \tau = \tau}$	U-Commut $\frac{\theta \vdash_{\text{unf}} \tau = \tau'}{\theta \vdash_{\text{unf}} \tau' = \tau}$	U-Tvar $\frac{\alpha \neq \tau}{[\alpha \mapsto \tau] \vdash_{\text{unf}} \alpha = \tau}$	U-Mut $\frac{\theta \vdash_{\text{unf}} \tau_1 = \tau_2}{\theta \vdash_{\text{unf}} \Psi \tau_1 = \Psi \tau_2}$
U-Ref $\frac{\theta \vdash_{\text{unf}} \tau_1 = \tau_2}{\theta \vdash_{\text{unf}} \uparrow\tau_1 = \uparrow\tau_2}$	U-Fn $\frac{\theta \vdash_{\text{unf}} \nabla\langle\tau_1\rangle = \nabla\langle\tau'_1\rangle \quad \theta' \vdash_{\text{unf}} \Delta\langle\tau_2\rangle = \Delta\langle\tau'_2\rangle \quad \frac{}{\text{acl}} \theta \circ \theta'}{\theta \circ \theta' \vdash_{\text{unf}} [\tau_1] \rightarrow [\tau_2] = [\tau'_1] \rightarrow [\tau'_2]}$		
U-Mb-Mb $\frac{\theta \vdash_{\text{unf}} \mathfrak{S}\langle\tau'_1\rangle = \mathfrak{S}\langle\tau'_2\rangle \quad \theta' \vdash_{\text{unf}} \theta\langle\tau_1\rangle = \theta\langle\tau_2\rangle \quad \frac{}{\text{acl}} \theta \circ \theta'}{\theta \circ \theta' \vdash_{\text{unf}} \tau_1 \downarrow \tau'_1 = \tau_2 \downarrow \tau'_2}$		$\frac{\tau = \tau_1 \downarrow \tau_2 \quad \mathfrak{S}\langle\tau_2\rangle = \tau'_2}{\mathfrak{S}\langle\tau\rangle = \tau'_2}$	
U-Mb-Oth $\frac{\tau_2 \neq \tau_3 \downarrow \tau'_3 \quad \tau_2 \neq \alpha \quad \theta \vdash_{\text{unf}} \mathfrak{S}\langle\tau'_1\rangle = \nabla\langle\tau_2\rangle \quad \theta' \vdash_{\text{unf}} \theta\langle\tau_1\rangle = \theta\langle\tau_2\rangle \quad \frac{}{\text{acl}} \theta \circ \theta'}{\theta \circ \theta' \vdash_{\text{unf}} \tau_1 \downarrow \tau'_1 = \tau_2}$			$\frac{\tau \neq \tau_1 \downarrow \tau_2}{\mathfrak{S}\langle\tau\rangle = \nabla\langle\tau\rangle}$

Figure 7: Unification Rules.

$$\begin{array}{c}
\textbf{Solve-Known} \\
\frac{\tau \neq \tau_1 \downarrow \tau_2 \quad \theta \vdash_s \tau : \gg \tau'}{\theta; x; e \vdash_{\text{solve}} \tau : \gg \tau'} \\
\textbf{Sol-Unit} \\
\frac{}{\emptyset \vdash_s \text{unit} : \gg \text{unit}} \\
\textbf{Sol-Fn} \\
\frac{\theta \vdash_s \tau_1 : \gg \tau'_1 \quad \theta' \vdash_s \theta \langle \tau_2 \rangle : \gg \tau'_2}{\theta \circ \theta' \vdash_s [\tau_1] \rightarrow [\tau_2] : \gg [\theta' \langle \tau'_1 \rangle] \rightarrow [\tau'_2]} \\
\textbf{Sol-Ct-Var} \\
\frac{\tau_1 = \alpha \quad \theta \vdash_s \tau_2 : \gg \tau'_2 \quad \theta' = [\alpha \mapsto \tau'_2]}{\theta \circ \theta' \vdash_s \tau_1 \downarrow \tau_2 : \gg \tau'_2} \\
\textbf{Solve-Mut} \\
\frac{\theta \vdash_s \tau_1 \downarrow \tau_2 : \gg \tau \quad \text{mut}(x, e)}{\theta; x; e \vdash_{\text{solve}} \tau_1 \downarrow \tau_2 : \gg \Delta(\tau)} \\
\textbf{Sol-Bool} \\
\frac{}{\emptyset \vdash_s \text{bool} : \gg \text{bool}} \\
\textbf{Sol-Ct-Const} \\
\frac{\tau_1 \neq \alpha \quad \vdash_{\text{cst}} \{\tau_1 \downarrow \tau_2\} \quad \theta \vdash_s \tau_1 : \gg \tau'_1}{\theta \vdash_s \tau_1 \downarrow \tau_2 : \gg \tau'_1} \\
\textbf{Solve-Immut} \\
\frac{\theta \vdash_s \tau_1 \downarrow \tau_2 : \gg \tau \quad \neg \text{mut}(x, e)}{\theta; x; e \vdash_{\text{solve}} \tau_1 \downarrow \tau_2 : \gg \nabla(\tau)} \\
\textbf{Sol-Mut} \\
\frac{\theta \vdash_s \tau : \gg \tau'}{\theta \vdash_s \Psi \tau : \gg \Psi \tau'} \\
\textbf{Sol-Ref} \\
\frac{\theta \vdash_s \tau : \gg \tau'}{\theta \vdash_s \uparrow \tau : \gg \uparrow \tau'}
\end{array}$$

Figure 8: Solving Copy Compatibility Constraints.

$$\begin{aligned}
\text{ftv}(\rho) &= \text{ftv}(\tau) \cup \text{ftv}(\mathcal{C}), \text{ where } \rho = \tau \setminus \mathcal{C} \\
\text{ftv}(\sigma) &= \text{ftv}(\bar{\alpha}) \cup \text{ftv}(\rho), \text{ where } \sigma = \forall \bar{\alpha}. \rho \\
\text{ftv}(\tau_1 \downarrow \tau_2) &= \text{ftv}(\tau_1) \cup \text{ftv}(\mathfrak{S}(\tau_2)).
\end{aligned}$$

Definition 11 (MTVs). The function $\text{mtv}(\omega) = \bar{\alpha}$ is defined to be the set of all type variables within the solvable entity ω . That is, it returns the set of all type-variables $\bar{\alpha}$ where α occurs within a maybe type as $\alpha \downarrow \tau_h$.

$$\begin{aligned}
\text{mtv}(\alpha) &= \{\} \\
\text{mtv}(\text{unit}) &= \{\} \\
\text{mtv}(\text{bool}) &= \{\} \\
\text{mtv}(\alpha \downarrow \tau_h) &= \alpha \cup \text{mtv}(\tau_h) \\
\text{mtv}(\tau \downarrow \tau_h) &= \text{mtv}(\tau), \text{ where } \tau \neq \alpha \\
\text{mtv}(\uparrow \tau) &= \text{mtv}(\tau) \\
\text{mtv}(\Psi \tau) &= \text{mtv}(\tau) \\
\text{mtv}(\tau_1 \rightarrow \tau_2) &= \text{mtv}(\tau_1) \cup \text{mtv}(\tau_2) \\
\text{mtv}(\Gamma) &= \bigcup \text{mtv}(\tau_i), \forall x_i \mapsto \tau_i \in \Gamma \\
\text{mtv}(\Sigma) &= \bigcup \text{mtv}(\tau_i), \forall L_i \mapsto \tau_i \in \Sigma \\
\text{mtv}(e) &= \bigcup \text{mtv}(\tau_i), \forall \tau_i \in e \\
\text{mtv}(\bar{\omega}) &= \bigcup \text{mtv}(\omega)
\end{aligned}$$

Definition 12 (NTVs). The set of unconstrained variables is defined as:

$$\begin{aligned}
\text{ntv}(\alpha) &= \{\alpha\} \\
\text{ntv}(\text{unit}) &= \{\} \\
\text{ntv}(\text{bool}) &= \{\} \\
\text{ntv}(\alpha \downarrow \tau_h) &= \text{ntv}(\tau_h) \\
\text{ntv}(\tau \downarrow \tau_h) &= \text{ntv}(\tau), \text{ where } \tau \neq \alpha \\
\text{ntv}(\uparrow \tau) &= \text{ntv}(\tau) \\
\text{ntv}(\Psi \tau) &= \text{ntv}(\tau) \\
\text{ntv}(\tau_1 \rightarrow \tau_2) &= \text{ntv}(\tau_1) \cup \text{ntv}(\tau_2) \\
\text{ntv}(\Gamma) &= \bigcup \text{ntv}(\tau_i), \forall x_i \mapsto \tau_i \in \Gamma \\
\text{ntv}(\Sigma) &= \bigcup \text{ntv}(\tau_i), \forall L_i \mapsto \tau_i \in \Sigma \\
\text{ntv}(e) &= \bigcup \text{ntv}(\tau_i), \forall \tau_i \in e \\
\text{ntv}(\bar{\omega}) &= \bigcup \text{ntv}(\omega)
\end{aligned}$$

Definition 13 (TVs). The set of all type variables in a solvable entity is given by $\text{TV}()$ function, defined as follows:

$$\begin{aligned}
\text{tv}(\omega) &= \text{mtv}(\omega) \cup \text{ntv}(\omega) \\
\text{tv}(\bar{\omega}) &= \bigcup \text{tv}(\omega)
\end{aligned}$$

Note that this function is different from $\text{FTV}()$, defined in Definition 10.

$$\text{Definition 14 (Constraint Set Extraction). } \frac{}{\{\{\emptyset\}\} = \emptyset} \quad \frac{\forall i = 1 \dots n \ \{\{\omega_i\}\} = \mathcal{C}_i}{\{\{\omega_1, \dots, \omega_n\}\} = \bigcup \mathcal{C}_i}$$

$$\frac{\frac{\{\{\omega_1\}\} = \mathcal{C}_1 \quad \{\{\omega_2\}\} = \mathcal{C}_2}{\{\{\omega_1\}\} \cup \{\{\omega_2\}\} = \mathcal{C}_1 \cup \mathcal{C}_2}}{\frac{\forall \tau_i \downarrow \tau'_i \in \tau, \mathcal{C}_i = \{\tau_i \cong \mathfrak{S}(\tau'_i)\}}{\{\{\tau\}\} = \bigcup \mathcal{C}_i} \quad \frac{\forall \tau_i \in e, \{\{\tau_i\}\} = \mathcal{C}_i}{\{\{e\}\} = \bigcup \mathcal{C}_i}}$$

$$\frac{\frac{\forall x \mapsto \tau_i \in \Gamma, \{\{\tau_i\}\} = \mathcal{C}_i}{\{\{\Gamma\}\} = \bigcup \mathcal{C}_i} \quad \frac{\forall L \mapsto \tau_i \in \Sigma, \{\{\tau_i\}\} = \mathcal{C}_i}{\{\{\Sigma\}\} = \bigcup \mathcal{C}_i}}$$

Definition 15 (Consistency of Maybe types). We identify the following consistency property on maybe types.

$$\frac{\begin{array}{l} \forall \alpha, \tau, \tau' \text{ such that } \tau \downarrow \tau' \in \bar{\omega}, \tau = \alpha \text{ or } \tau \cong \mathfrak{S}(\tau') \\ \forall \alpha \downarrow \tau \text{ and } \alpha \downarrow \tau' \in \bar{\omega}, \tau = \tau' \quad \text{mtv}(\bar{\omega}) \cap \text{ntv}(\bar{\omega}) = \emptyset \end{array}}{\Vdash \{\bar{\omega}\}}$$

Definition 16 (Constraint Set Closure). A closure operation $\text{close}(\mathcal{C})$ on a constraint set \mathcal{C} produces an equivalent set of atomic constraints by using the copy coercion rules defined in Figure 3 (note that this conversion is total); and by explicitly adding all transitive relationships. This conversion does not affect the MPCs in \mathcal{C} .

1. $\text{close}(\{\tau_1 \rightarrow \tau_2 \preccurlyeq: \tau'_1 \rightarrow \tau'_2\} \cup \mathcal{C}) = \text{close}(\mathcal{C} \cup \{\tau_1 \preccurlyeq: \tau'_1, \tau_2 \preccurlyeq: \tau'_2, \tau'_1 \preccurlyeq: \tau_1, \tau'_2 \preccurlyeq: \tau_2\})$
2. $\text{close}(\{\uparrow \tau \preccurlyeq: \uparrow \tau'\} \cup \mathcal{C}) = \text{close}(\mathcal{C} \cup \{\tau \preccurlyeq: \tau', \tau' \preccurlyeq: \tau\})$
3. $\text{close}(\{\Psi \tau \preccurlyeq: \Psi \tau'\} \cup \mathcal{C}) = \text{close}(\mathcal{C} \cup \{\tau \preccurlyeq: \tau', \tau' \preccurlyeq: \tau\})$
4. $\text{close}(\{\tau_1 \preccurlyeq: \tau_2, \tau_2 \preccurlyeq: \tau_3\} \cup \mathcal{C})$ where $\tau_1 \preccurlyeq: \tau_3 \notin \mathcal{C}$
 $= \text{close}(\{\tau_1 \preccurlyeq: \tau_2, \tau_2 \preccurlyeq: \tau_3, \tau_1 \preccurlyeq: \tau_3\} \cup \mathcal{C})$
5. $\text{close}(\mathcal{C})$ where none of the above cases are applicable $= \mathcal{C}$

Definition 17 (Normalization of Constraint Sets). The normalization function $\mathbf{N}(\mathcal{C})$ is used to obtain a normalized form of a constraint set \mathcal{C} by re-writing it only in terms of closed subtype constraints.

$\mathbf{N}(\mathcal{C}) = \text{close}(\mathcal{C}')$ where \mathcal{C}' is obtained from \mathcal{C} using the normalizing translations:

- $\llbracket \tau_1 = \tau_2 \rrbracket \rightsquigarrow \llbracket \tau_1 \preccurlyeq: \tau_2, \tau_2 \preccurlyeq: \tau_1 \rrbracket$
- $\llbracket \tau_1 \cong \tau_2 \rrbracket \rightsquigarrow \llbracket \tau_1 \preccurlyeq: \alpha, \tau_2 \preccurlyeq: \alpha \rrbracket$, where $\Vdash_{\text{new}} \alpha$.

Definition 18 (Canonical forms of Solvable Entities). For any ω such that $\langle \rangle \Vdash_{\text{sat}} \{\{\omega\}\}$, we write $\underline{\omega}$ to represent the equivalent entity in which (1) all meta-constructors in are fully interpreted, and (2) the (tautological) constraints embedded within ω are removed to obtain an equivalent unconstrained entity.

For example, if $\tau = \Psi \text{bool} \downarrow \text{bool}$, then $\underline{\tau} = \Psi \text{bool}$.

Definition 19 (Constraint Satisfiability). A Substitution θ satisfies a constraint set \mathcal{C} , written $\theta \Vdash_{\text{sat}} \mathcal{C}$ iff $\mathbf{N}(\theta(\mathcal{C}))$ consists only of tautologies (trivially).

$$\text{Definition 20 (Consistency and Acyclicity). } \frac{\exists \theta \text{ such that } \theta \Vdash_{\text{sat}} \mathcal{C}}{\Vdash_{\text{cst}} \mathcal{C}}$$

$$\frac{\forall [\alpha \rightsquigarrow \tau] \text{ and } [\beta \rightsquigarrow \tau'] \text{ in } \theta, \alpha \neq \beta}{\Vdash_{\text{cst}} \theta}$$

$$\frac{\forall \alpha \preccurlyeq: \tau \text{ or } \alpha \preccurlyeq: \tau \in \mathbf{N}(\mathcal{C}), (\tau = \alpha) \vee (\tau = \Psi \alpha) \vee (\alpha \notin \tau)}{\Vdash_{\text{acy}} \mathcal{C}}$$

$$\frac{\frac{\frac{}{\vdash_{acy} \langle \rangle}}{\vdash_{cst} C} \quad \frac{\frac{}{\vdash_{acy} C}}{\vdash_{ca} C}}{\frac{\theta = [\alpha \mapsto \tau] \circ \theta' \quad \alpha \notin \tau \quad \vdash_{acy} [\alpha \mapsto \tau] \langle \theta' \rangle}{\vdash_{acy} \theta}}{\vdash_{cst} \theta} \quad \frac{\frac{}{\vdash_{acy} \theta}}{\vdash_{ca} \theta}}{\vdash_{ca} \theta}}$$

Definition 21 (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.*

The type inference algorithm is as shown in Figure 6. The inference judgment $\Gamma; \Sigma \vdash e : \tau \parallel \theta$ should be understood as: given the binding context Γ and the store typing Σ , we infer the type τ for the expression e . θ is list of substitutions obtained by unifications performed during inference, and must be propagated to further derivations. The judgment $\vdash_{new} \bar{\alpha}$ identifies new type variables.

We use the following shorthand translations as a notational convenience. The translations defined in section 7.2 are repeated here.

1. $\llbracket \Gamma; \Sigma \vdash e \preceq : \tau \rrbracket \rightsquigarrow \llbracket \Gamma; \Sigma \vdash e : \tau', \tau \preceq : \tau' \rrbracket$
2. $\llbracket \Gamma; \Sigma \vdash e : \tau \rrbracket \rightsquigarrow \llbracket \Gamma; \Sigma \vdash e : \tau \parallel \langle \rangle \rrbracket$
3. $\llbracket \theta_0 \parallel \Gamma; \Sigma \vdash e : \tau \parallel \theta \rrbracket \rightsquigarrow \llbracket \theta_0 \langle \Gamma \rangle; \theta_0 \langle \Sigma \rangle \vdash \theta_0 \langle e \rangle : \tau \parallel \theta \rrbracket$
4. $\llbracket \theta \parallel \Gamma; \Sigma \vdash e : \tau \rrbracket \rightsquigarrow \llbracket \theta \langle \Gamma \rangle; \theta \langle \Sigma \rangle \vdash \theta \langle e \rangle : \theta \langle \tau \rangle \rrbracket$
5. $\llbracket \theta \parallel \Gamma; \Sigma \vdash e : \tau \rrbracket \rightsquigarrow \llbracket \theta \langle \Gamma \rangle; \theta \langle \Sigma \rangle \vdash \theta \langle e \rangle : \theta \langle \tau \rangle \rrbracket$
6. $\llbracket \theta \parallel \Gamma; \Sigma \vdash e : \tau \rrbracket \rightsquigarrow \llbracket \underline{\theta \langle \Gamma \rangle}; \underline{\theta \langle \Sigma \rangle} \vdash \underline{\theta \langle e \rangle} : \underline{\theta \langle \tau \rangle} \rrbracket$
7. $\llbracket \theta \{ \dots \} \rrbracket \rightsquigarrow \llbracket \theta \langle \{ \dots \} \rangle \rrbracket$
8. $\llbracket \theta \{ \dots \} \rrbracket \rightsquigarrow \llbracket \theta \langle \{ \dots \} \rangle \rrbracket$

Unification rules are as shown in Figure 8. The unification judgment $\theta \vdash_{unf} \tau_1 = \tau_2$ is understood as: τ_1 unifies with τ_2 under the substitution θ .

A constraint solver for solving copy compatibility constraints at `let`-boundaries is defined in Figure 7. The judgment $\theta; x; e \vdash_{solve} \tau_1 \gg \tau_2$ should be read as: the (possibly) constrained type τ_1 for the identifier x (possibly) used in the expression e is transformed to the unconstrained type τ_2 by solving all the copy compatibility constraints.

We prove the soundness of the inference system, through an intermediate declarative system defined in Figure 5

Lemma 10 (\preceq : implies \trianglelefteq :). *If $\tau \preceq : \tau'$, then $\tau \trianglelefteq : \tau'$.*

Proof. Evident from the definition of \trianglelefteq : in Figure 5. Both \preceq : and \trianglelefteq : are partial functions on types, but \trianglelefteq : clearly covers all cases that \preceq : relates. \square

Lemma 11 (\trianglelefteq : begets \preceq :). *If $\tau \trianglelefteq : \tau'$, then $\exists \theta$ such that $\underline{\theta \langle \tau \rangle} \preceq : \underline{\theta \langle \tau' \rangle}$.*

Proof. By construction of θ . One possible solution is $\theta = \forall \alpha \downarrow \tau_h \in \tau, [\alpha \mapsto \mathfrak{S}(\tau_h)]$. \square

Lemma 12 (Consistency of \trianglelefteq :). *If $\tau \trianglelefteq : \tau'$ and $\Vdash \{ \tau, \tau' \}$, then $\forall \theta$ such that $\theta \vdash_{sat} \{ \tau \}$, we have $\underline{\theta \langle \tau \rangle} \preceq : \underline{\theta \langle \tau' \rangle}$.*

Proof. By straightforward induction over the derivation of $\tau \trianglelefteq : \tau'$, with case analysis over the definition of \trianglelefteq : in Figure 5. A normalizing derivation with no redundant applications of reflexive and transitive rules must be considered. \square

Lemma 13 (Weakening of Satisfiability and Consistency). *Weakening of properties over solvable entities:*

1. *If $\theta \vdash_{sat} C$ and $C' \subseteq C$, then $\theta \vdash_{sat} C'$.*

2. If $\models_{\text{cst}} \mathcal{C}$ and $\mathcal{C}' \subseteq \mathcal{C}$, then $\models_{\text{cst}} \mathcal{C}'$.
3. If $\Vdash \{\overline{\omega}_n\}$, and $\{\overline{\omega}_m\} \subseteq \{\overline{\omega}_n\}$, then $\Vdash \{\overline{\omega}_m\}$.
4. If $\Vdash \{\overline{\omega}\}$, then $\models_{\text{cst}} \{\overline{\omega}\}$.

Proof. Evident from Definition 19, Definition 20, and Definition 15. \square

Lemma 14 (Satisfied constraints are consistent). *If $\theta \models_{\text{sat}} \{\overline{\omega}\}$, then $\Vdash \theta\{\overline{\omega}\}$.*

Proof. Due to assumption (3), since θ satisfies all constraints in $\overline{\omega}$, we must have $\text{mtv}(\theta\langle\overline{\omega}\rangle) = \emptyset$. That is, there exists no $\alpha \downarrow \tau \in \theta\langle\overline{\omega}\rangle$. Further, we must also have $\forall \tau \downarrow \tau' \in \theta\langle\overline{\omega}\rangle, \tau \cong \mathfrak{S}(\tau')$. Now, from Definition 15, we obtain $\Vdash \theta\{\overline{\omega}\}$. \square

Lemma 15 (Substitution on Declarative Derivation). *If $\Gamma; \Sigma \vdash e : \tau$ then $\theta \parallel \Gamma; \Sigma \vdash e : \tau$.*

Proof. Straightforward induction on the derivation of $\Gamma; \Sigma \vdash e : \tau$, except for the fact that we should use appropriate α -renaming on generalized variables $\forall \sigma \in \Gamma$, so that generalized variables do not get substituted. \square

Lemma 16 (Consistency of Intermediate Derivation). *If $\Gamma; \Sigma \vdash e : \tau$, then $\Vdash \{\Gamma, \Sigma, e, \tau\}$.*

Proof. Evident from the definition of the intermediate type system in figure 5 (each rule explicitly includes a consistency constraint), and Lemma 13 (weakening). \square

Lemma 17 (Substitution Consistency of Maybe types). *If*

1. $\Vdash \{\overline{\omega}_n\}$
2. $\{\overline{\omega}_m\} \subseteq \{\overline{\omega}_n\}$
3. θ is a substitution such that $\Vdash \{\theta\langle\overline{\omega}_m\rangle\}$
4. $\text{dom}(\theta) \cap \text{tv}(\overline{\omega}_n) = \text{tv}(\overline{\omega}_m)$

Then, $\Vdash \theta\{\overline{\omega}_n\}$.

Proof. 1. Due to assumption (1), and Definition 15, we have

- (a) $\forall \alpha, \tau, \tau'$ such that $\tau \downarrow \tau' \in \overline{\omega}_n, \tau = \alpha$ or $\tau \cong \mathfrak{S}(\tau')$
- (b) $\forall \alpha \downarrow \tau$ and $\alpha \downarrow \tau' \in \overline{\omega}_n, \tau = \tau'$
- (c) $\text{mtv}(\overline{\omega}_n) \cap \text{ntv}(\overline{\omega}_n) = \emptyset$

2. Due to assumption (3), Definition 15, we have

- (a) $\forall \alpha, \tau, \tau'$ such that $\tau \downarrow \tau' \in \theta\langle\overline{\omega}_m\rangle, \tau = \alpha$ or $\tau \cong \mathfrak{S}(\tau')$
- (b) $\forall \alpha \downarrow \tau$ and $\alpha \downarrow \tau' \in \theta\langle\overline{\omega}_m\rangle, \tau = \tau'$
- (c) $\text{mtv}(\theta\langle\overline{\omega}_m\rangle) \cap \text{ntv}(\theta\langle\overline{\omega}_m\rangle) = \emptyset$

3. Due to assumption (4), we can write $\theta = \theta_1 \circ \theta_2 \circ \theta_3$ such that $\text{dom}(\theta_1) = \text{mtv}(\overline{\omega}_m)$, $\text{dom}(\theta_2) = \text{ntv}(\overline{\omega}_m)$, and $\text{dom}(\theta_3) \cap \text{tv}(\overline{\omega}_n) = \emptyset$.

4. From cases (2.a, 2.b, 2.c, 1.a, and 3), we can conclude that $\forall \alpha, \tau, \tau'$ such that $\tau \downarrow \tau' \in \theta\langle\overline{\omega}_n\rangle, \tau = \alpha$ or $\tau \cong \mathfrak{S}(\tau')$

5. From cases (2.b, 1.b, and 3), we conclude that $\forall \alpha \downarrow \tau$ and $\alpha \downarrow \tau' \in \theta\langle\overline{\omega}_n\rangle, \tau = \tau'$.

6. From cases (2.c, 1.c, and 3), we conclude that $\text{mtv}(\theta\langle\overline{\omega}_n\rangle) \cap \text{ntv}(\theta\langle\overline{\omega}_n\rangle) = \emptyset$

7. From cases (4, 5, 6) and Definition 15, we obtain $\Vdash \theta\{\overline{\omega}_n\}$. □

Lemma 18 (Partial solutions). *If*

1. $\Vdash \{\overline{\omega}_n\}$
2. $\{\overline{\omega}_m\} \subseteq \{\overline{\omega}_n\}$
3. $\theta_m \Vdash_{\text{sat}} \{\overline{\omega}_m\}$
4. $\text{dom}(\theta_m) \cap \text{tv}(\overline{\omega}_n) = \text{tv}(\overline{\omega}_m)$

Then,

1. $\Vdash \theta_m\{\overline{\omega}_n\}$.
2. $\exists \theta_n$ such that $\theta_n \Vdash_{\text{sat}} \theta_m\{\overline{\omega}_n\}$, and $\theta_{n.m}\{\overline{\omega}_m\} = \theta_m\{\overline{\omega}_m\}$.

Proof. 1. From assumption (3) and Lemma 14, we conclude that $\Vdash \theta_m\{\overline{\omega}_m\}$.

2. From assumptions (1 and 2), case (1), assumptions (4) and Lemma 17, we obtain $\Vdash \theta_m\{\overline{\omega}_n\}$, the required conclusion (1).

3. Conclusion (2) can be proved by construction of θ_n . One possibility is $\theta_n = [\alpha \mapsto \Im(\tau)], \forall \alpha \downarrow \tau \in \theta_m\{\overline{\omega}_n\}$.

- (a) Clearly, $\theta_n \Vdash_{\text{sat}} \theta_m\{\overline{\omega}_n\}$.
- (b) We know that $\text{dom}(\theta) = \text{mtv}(\theta_m\{\overline{\omega}_n\})$.
- (c) Due to assumption (3), we must have $\text{mtv}(\theta_m\{\overline{\omega}_m\}) = \emptyset$.
- (d) By construction, $\text{dom}(\theta) \cap \text{ntv}(\theta_m\{\overline{\omega}_n\}) = \emptyset$, and consequently, $\text{dom}(\theta) \cap \text{ntv}(\theta_m\{\overline{\omega}_m\}) = \emptyset$.
- (e) From cases (3.c and 3.d), we conclude that $\theta_{n.m}\{\overline{\omega}_m\} = \theta_m\{\overline{\omega}_m\}$

□

Lemma 19 (Additivity of Consistent Entities). *If*

1. $\Vdash \{\overline{\omega}, \overline{\omega}_m\}$
2. $\Vdash \{\overline{\omega}, \overline{\omega}_n\}$
3. $\text{tv}(\overline{\omega}_m) \cap \text{tv}(\overline{\omega}_n) = \text{tv}(\overline{\omega})$

Then, $\Vdash \{\overline{\omega}, \overline{\omega}_m, \overline{\omega}_n\}$.

Proof. 1. $\forall \tau \downarrow \tau' \in \overline{\omega}, \overline{\omega}_m$, or $\overline{\omega}_n$, if $\tau \neq \alpha$ for some α , then from assumptions (1 and 2), we must have $\tau \cong \Im(\tau')$.

2. $\forall \alpha \downarrow \tau_i \in \overline{\omega}_m$, and $\overline{\omega}_n$, from assumptions (1, 2, and 3), and Definition 15, we have $\exists \alpha \downarrow \tau_0 \in \overline{\omega}$, such that $\overline{\tau}_i = \tau_0$.

3. Due to assumption (1), we must have $\text{mtv}(\overline{\omega}, \overline{\omega}_m) \cap \text{ntv}(\overline{\omega}, \overline{\omega}_m) = \emptyset$. Due to assumption (2), we must have $\text{mtv}(\overline{\omega}, \overline{\omega}_n) \cap \text{ntv}(\overline{\omega}, \overline{\omega}_n) = \emptyset$. This, along with assumption (3) gives us $\text{mtv}(\overline{\omega}, \overline{\omega}_m, \overline{\omega}_n) \cap \text{ntv}(\overline{\omega}, \overline{\omega}_m, \overline{\omega}_n) = \emptyset$.

4. From cases (1, 2, and 3), and Definition 15, we obtain $\Vdash \{\overline{\omega}, \overline{\omega}_m, \overline{\omega}_n\}$.

□

Theorem 2 (Soundness of Intermediate System). *If:*

1. $\Gamma; \Sigma \Vdash e : \tau$
2. $\theta \models_{sat} \{\Gamma, \Sigma, e, \tau\}$.

Then, $\theta \parallel \Gamma; \Sigma \vdash e : \tau$.

Proof. By induction on the derivation of $\Gamma; \Sigma \Vdash e : \tau$. We proceed by case analysis on the last step, assuming α -reduction vacuously.

1. Cases J-Unit, J-True, J-False, J-Id, J-Hloc, J-Sloc are trivial.

2. Case J-Lambda:

(a) In this case, we have:

- i.
$$\frac{\Gamma, x \mapsto \tau_1; \Sigma \Vdash e : \tau_2 \quad \Vdash \{\Gamma, \Sigma, e, \tau_1, \tau_2\}}{\Gamma; \Sigma \Vdash \lambda x. e : [\tau_1] \rightarrow [\tau_2]}}$$
- ii. $\theta \models_{sat} \{\Gamma, \Sigma, \lambda x. e_i, [\tau_1] \rightarrow [\tau_2]\}$

and we need to show that $\theta \parallel \Gamma; \Sigma \vdash \lambda x. e_i : [\tau_1] \rightarrow [\tau_2]$.

- (b) It is evident (from the syntactic structure) that $\{\lambda x. e_i\} = \{e_i\}$ and $\{[\tau_1] \rightarrow [\tau_2]\} = \{\tau_1, \tau_2\}$. Therefore, we can re-write case (2.a.ii) as $\theta \models_{sat} \{\Gamma, \Sigma, e_i, \tau_1, \tau_2\}$. Similarly, since $\{\Gamma, \tau_1\} = \{(\Gamma, x \mapsto \tau_1)\}$, we can write $\theta \models_{sat} \{\Gamma, x \mapsto \tau_1, \Sigma, e_i, \tau_2\}$.
- (c) Since we have $\Gamma, x \mapsto \tau_1; \Sigma \vdash e_i : \tau_2$, and $\theta \models_{sat} \{\Gamma, x \mapsto \tau_1, \Sigma, e_i, \tau_2\}$, from induction hypothesis, we obtain $\theta \parallel \Gamma, x \mapsto \tau_1; \Sigma \vdash e_i : \tau_2$.
- (d) $\theta \parallel \Gamma, x \mapsto \tau_1; \Sigma \vdash e_i : \tau_2$ is a shorthand for $\underline{\theta(\Gamma, x \mapsto \tau_1)}; \underline{\theta(\Sigma)} \vdash \underline{\theta(e_i)} : \underline{\theta(\tau_2)}$, which can be re-written as $\underline{\theta(\Gamma)}, x \mapsto \underline{\theta(\tau_1)}; \underline{\theta(\Sigma)} \vdash \underline{\theta(e_i)} : \underline{\theta(\tau_2)}$.
- (e) From case (2.d) and T-Lambda rule in figure 3, we obtain $\underline{\theta(\Gamma)}; \underline{\theta(\Sigma)} \vdash \lambda x. \underline{\theta(e_i)} : \nabla(\underline{\theta(\tau_1)}) \rightarrow \Delta(\underline{\theta(\tau_2)})$.
- (f) From Definition 18 and Definition 9, it is clear that $|\underline{\theta(\tau_1)}| = \nabla(\underline{\theta(\tau_1)})$, and $[\underline{\theta(\tau_2)}] = \Delta(\underline{\theta(\tau_2)})$. Therefore, we can write $\nabla(\underline{\theta(\tau_1)}) \rightarrow \Delta(\underline{\theta(\tau_2)}) = |\underline{\theta(\tau_1)}| \rightarrow [\underline{\theta(\tau_2)}] = |\underline{\theta(\tau_1)}| \rightarrow [\underline{\theta(\tau_2)}] = \underline{\theta([\tau_1] \rightarrow [\tau_2])}$. It is further evident that $\lambda x. \underline{\theta(e_i)} = \underline{\theta(\lambda x. e_i)}$.
- (g) Substituting the equivalencies in case (2.f) into case (2.e), we obtain: $\underline{\theta(\Gamma)}; \underline{\theta(\Sigma)} \vdash \underline{\theta(\lambda x. e_i)} : \underline{\theta([\tau_1] \rightarrow [\tau_2])}$. That is, $\theta \parallel \Gamma; \Sigma \vdash \lambda x. e_i : [\tau_1] \rightarrow [\tau_2]$.

3. Case J-App:

(a) In this case, we have:

- (A) $\Gamma; \Sigma \Vdash e_1 : \tau_1$
- (B) $\tau_1 \leq: [\tau_a] \rightarrow [\tau_r]$
- (C) $\Gamma; \Sigma \Vdash e_2 : \tau_2$
- (D) $\tau_2 \leq: \mathfrak{S}(\tau_a)$
- (E) $\tau_r \leq: \mathfrak{S}(\tau)$
- i.
$$\frac{\Vdash \{\Gamma, \Sigma, e_1, e_2, \tau, \tau_1, \tau_2, [\tau_a] \rightarrow [\tau_r]\}}{\Gamma; \Sigma \Vdash e_1 e_2 : \tau}}$$
- ii. $\theta \models_{sat} \{\Gamma, \Sigma, e_1 e_2, \tau\}$.

We need to show that $\theta \parallel \Gamma; \Sigma \vdash e_1 e_2 : \tau$.

(b) From case (3.a.ii), we obtain $\theta \models_{sat} \{\Gamma, \Sigma, e_1, e_2, \tau\}$.

- (c) i. From case (3.a.i.F), we have $\Vdash \{\Gamma, \Sigma, e_1, e_2, \tau, \tau_1, \tau_2, [\tau_a] \rightarrow [\tau_r]\}$.
- ii. Clearly, $\{\Gamma, \Sigma, e_1, e_2, \tau\} \subseteq \{\Gamma, \Sigma, e_1, e_2, \tau, \tau_1, \tau_2, [\tau_a] \rightarrow [\tau_r]\}$.
- iii. From case (3.b), we have $\theta \Vdash_{\text{sat}} \{\Gamma, \Sigma, e_1, e_2, \tau\}$.
- iv. We can assume that $\text{dom}(\theta) \cap \text{tvs}(\{\Gamma, \Sigma, e_1, e_2, \tau, \tau_1, \tau_2, [\tau_a] \rightarrow [\tau_r]\}) = \text{tvs}(\{\Gamma, \Sigma, e_1, e_2, \tau\})$, because this property can be obtained by suitable α -renaming using fresh type variables. The only common type variables that need to be common to both the derivations are those present in Γ or Σ .
- v. Now, from cases (3.c.i, 3.c.ii, 3.c.iii and 3.c.iv) and Lemma 18, we conclude that $\exists \theta''$ such that if $\theta' = \theta'' \circ \theta$, we have
 - A. $\theta'' \Vdash_{\text{sat}} \theta\{\Gamma, \Sigma, e_1, e_2, \tau, \tau_1, \tau_2, [\tau_a] \rightarrow [\tau_r]\}$, and therefore, $\theta' \Vdash_{\text{sat}} \{\Gamma, \Sigma, e_1, e_2, \tau, \tau_1, \tau_2, [\tau_a] \rightarrow [\tau_r]\}$.
 - B. $\theta'\{\Gamma, \Sigma, e_1, e_2, \tau\} = \theta\{\Gamma, \Sigma, e_1, e_2, \tau\}$.
- (d) From case (3.c.v.A) and Lemma 13 (weakening), we obtain
 - i. $\theta' \Vdash_{\text{sat}} \{\Gamma, \Sigma, e_1, \tau_1\}$
 - ii. $\theta' \Vdash_{\text{sat}} \{\Gamma, \Sigma, e_2, \tau_2\}$
- (e) From case (3.d.i), and induction hypothesis with respect to case (3.a.i.A), we conclude that $\theta' \parallel \Gamma; \Sigma \vdash_{\circ} e_1 : \tau_1$.
- (f) From case (3.d.ii), and induction hypothesis with respect to case (3.a.i.C), we conclude that $\theta' \parallel \Gamma; \Sigma \vdash_{\circ} e_2 : \tau_2$.
- (g) From cases (3.a.i.F and 3.c.v.A), and Lemma 13 (weakening), we obtain $\Vdash \{\tau_1, [\tau_a] \rightarrow [\tau_r]\}$ and $\theta' \Vdash_{\text{sat}} \{\tau_1, [\tau_a] \rightarrow [\tau_r]\}$. Now, from case (3.a.i.B) and Lemma 12, we obtain $\theta'\langle\tau_1\rangle \preceq: \theta'\langle[\tau_a] \rightarrow [\tau_r]\rangle$. That is, $\theta'\langle\tau_1\rangle \preceq: \nabla(\theta'\langle\tau_a\rangle) \rightarrow \Delta(\theta'\langle\tau_r\rangle)$.
- (h) Similarly, for case (3.a.i.D), we obtain $\theta'\langle\tau_2\rangle \preceq: \theta'\langle\mathfrak{S}(\tau_a)\rangle$. Given case (3.c.v.A), $\theta'\langle\tau_2\rangle \cong \theta'\langle\mathfrak{S}(\tau_a)\rangle$. For any two types τ' and τ'' such that $\tau' \cong \tau''$, we have $\tau' \preceq: \nabla(\tau')$, and $\tau'' \preceq: \nabla(\tau')$. Therefore, we conclude that $\theta'\langle\mathfrak{S}(\tau_a)\rangle \preceq: \nabla(\theta'\langle\tau_a\rangle)$. We can now write $\theta'\langle\tau_2\rangle \preceq: \nabla(\theta'\langle\tau_a\rangle)$.
- (i) Similar to case (h), from case (3.a.i.E), we obtain $\Delta(\theta'\langle\mathfrak{S}(\tau_r)\rangle) \preceq: \theta'\langle\tau\rangle$.
- (j) From cases (3.e, 3.f, 3.g, 3.h, and 3.i) and the T-App rule in figure 3, we obtain, $\theta' \parallel \Gamma; \Sigma \vdash_{\circ} e_1 e_2 : \tau$.
- (k) From cases (3.j, and 3.c.v.B), we finally conclude that $\theta \parallel \Gamma; \Sigma \vdash_{\circ} e_1 e_2 : \tau$.

4. Cases J-If, J-Dup, J-Deref, J-Set, J-Tqexpr, J-Let-M[Tq], and J-Let-P[Tq] are similar. □

Lemma 20 (Substitution on Intermediate Derivation). *If $\Gamma; \Sigma \vdash_{\circ} e : \tau$ and θ is a substitution such that $\Vdash \theta\{\Gamma, \Sigma, e, \tau\}$, then $\theta \parallel \Gamma; \Sigma \vdash_{\circ} e : \tau$.*

Proof. Straightforward induction on the derivation of $\Gamma; \Sigma \vdash_{\circ} e : \tau$, similar to Lemma 15. □

Theorem 3 (Correctness of Unification). *If $\theta \vdash_{\text{unf}} \tau_1 = \tau_2$, then:*

1. $\theta\langle\tau_1\rangle \preceq: \theta\langle\tau_2\rangle$ and $\theta\langle\tau_2\rangle \preceq: \theta\langle\tau_1\rangle$
2. $\Vdash_{\text{ca}} \{\tau_1\}$, and $\Vdash_{\text{ca}} \{\tau_2\}$ implies $\Vdash_{\text{ca}} \theta\{\tau_1, \tau_2\}$.

Proof. By straightforward induction on the derivation of $\theta \vdash_{\text{unf}} \tau_1 = \tau_2$. □

Lemma 21 (Consistency of Unified Types). *If $\theta \vdash_{\text{unf}} \tau_1 = \tau_2$, then:*

1. $\Vdash \{\bar{\omega}, \tau_1, \tau_2\}$ implies $\Vdash \theta\{\bar{\omega}, \tau_1, \tau_2\}$.

2. $\Vdash \{\bar{\omega}, \tau_1\}$, and $\Vdash \{\bar{\omega}, \tau_2\}$ implies $\Vdash \theta\{\bar{\omega}, \tau_1, \tau_2\}$.

Proof. By straightforward induction on the derivation of $\theta \vdash_{unf} \tau_1 = \tau_2$. \square

Theorem 4 (Correctness of the Constraint Solver). *If $\theta \vdash_s \tau : \gg \tau'$, then $\theta \Vdash_{sat} \{\tau\}$ and $\underline{\theta\langle\tau\rangle} = \underline{\tau'}$*

Proof. By straightforward induction on the derivation of $\theta \vdash \tau : \gg \tau'$, noting that the solver infers compatible types at steps Sol-Ct-Var and that the Sol-Ct-Const explicitly checks for compatibility. \square

Lemma 22 (Corollary to Correctness of the Constraint Solver). *If $\theta \vdash_{solve} \tau : \gg \tau'$, then $\theta \Vdash_{sat} \{\tau\}$ and $\underline{\theta\langle\tau\rangle} = \underline{\tau'}$*

Proof. Follows from Theorem 4 and the definition of \vdash_{solve} figure 7. \square

Theorem 5 (Totality of the Constraint Solver). *If $\Vdash_{cst} \{\tau\}$ then $\exists \theta$ and τ' such that $\theta \vdash_s \tau : \gg \tau'$.*

Proof. Evident from the definition of the solver in figure 7. \square

Theorem 6 (Uniqueness of Solutions Produced by the Solver). *If $\theta' \vdash_s \tau : \gg \tau'$ and $\theta'' \vdash_s \tau : \gg \tau''$, then $\theta' = \theta''$ and $\tau' = \tau''$.*

Proof. Evident from the definition of the solver in figure 7. \square

Theorem 7 (Decidability of Unification). *If $\Vdash_{acy} \{\tau_1\}$ and $\Vdash_{acy} \{\tau_2\}$, then, a normalizing derivation of $\theta \vdash_{unf} \tau_1 = \tau_2$ where no two uses of U-Commut occur consecutively is decidable.*

Proof. The unifier and constraint solver builds a solution tree by always invoking itself types having *smaller* shapes of types (after eliminating redundant uses of the U-Commut rule). Since types are of bounded size and acyclic, and since unification itself does not produce any cycles (Theorem 3), these derivations must be bounded. \square

Theorem 8 (Decidability of the Constraint Solver). *If $\Vdash_{acy} \{\tau\}$, then, the derivation of $\theta \vdash_s \tau : \gg \tau'$ is decidable.*

Proof. Similar to Theorem 7. \square

Lemma 23 (Structural Isomorphism). *If $\tau_1 \trianglelefteq \tau_2$ and $\tau_2 \trianglelefteq \tau_1$, then for any substitution θ , we have $\theta\langle\tau_1\rangle \trianglelefteq \theta\langle\tau_2\rangle$ and $\theta\langle\tau_2\rangle \trianglelefteq \theta\langle\tau_1\rangle$.*

Proof. Since we have both $\tau_1 \trianglelefteq \tau_2$ and $\tau_2 \trianglelefteq \tau_1$, the two types τ_1 and τ_2 must be structurally equivalent expect for the fact that one of the types can be of the form $\tau \downarrow \tau'$, and the other of the form τ . The conclusion is thus evident. \square

Lemma 24 (Consistency of Heuristic Type Inference). *If $\Gamma; \Sigma \vdash_{\tau} e : \tau \parallel \theta$ and $\Vdash \{\Gamma, \Sigma\}$ then, $\Vdash \theta\{\Gamma, \Sigma, e, \tau\}$*

Proof. By induction on the derivation of $\Gamma; \Sigma \vdash_{\tau} e : \tau \parallel \theta$, noting that (1) all maybe types are introduced through new type variables, (2) the two parts of a maybe type are never separated once constructed (3) all substitutions produced during inference are obtained from the unifier or the solver, which preserve consistency according to Theorem 3 (property 3), and Theorem 4. \square

Lemma 25 (Type Variable Propagation). *If $\Gamma; \Sigma \vdash_{\tau} e_1 : \tau_1 \parallel \theta_1$ and $\theta_1 \parallel \Gamma; \Sigma \vdash_{\tau} e_2 : \tau_2 \parallel \theta_2$, then $\text{tv}(\theta_{1.2}\langle e_1, \tau_1 \rangle) \cap \text{tv}(\theta_{1.2}\langle e_2, \tau_2 \rangle) = \text{tv}(\theta_{1.2}\langle \Gamma, \Sigma \rangle)$.*

Proof. By straightforward induction on the derivation of $\theta_1 \parallel \Gamma; \Sigma \vdash_{\bar{\tau}} e_2 : \tau_2 \parallel \theta_2$. This derivation happens in an environment that already contains the substitutions obtained from the first derivation: θ_1 . All of the cases in the second derivation either use (1) primitive types that do not alter the type variables involved, (2) types (and thus type variables) from $\theta_1 \langle \Gamma \rangle$ or $\theta_1 \langle \Sigma \rangle$, or (3) new type variables. Therefore, it is evident that, if $\{\bar{\alpha}\} = \bigcup \text{tv}(\tau)$, $\forall \tau \in \text{range}(\theta_2)$, we have $(\text{dom}(\theta_2) \cup \{\bar{\alpha}\} \cup \text{tv}(\theta_{1.2} \langle e_2, \tau_2 \rangle)) \cap \text{tv}(\theta_1 \langle e_1, \tau_1 \rangle) = \text{tv}(\theta_1 \langle \Gamma, \Sigma \rangle)$. From this, we can conclude that $\text{tv}(\theta_{1.2} \langle e_1, \tau_1 \rangle) \cap \text{tv}(\theta_{1.2} \langle e_2, \tau_2 \rangle) = \text{tv}(\theta_{1.2} \langle \Gamma, \Sigma \rangle)$.

Note: For the sake of simplicity, type variables present as type qualifications that are unbound in the environment are treated as fresh type variables. For example, the expression `if true then $e_a : \alpha$ else $e_2 : \alpha$` , where $\alpha \notin \text{tv}(\Gamma, \Sigma)$ is equivalent to `if true then $e_a : \beta$ else $e_2 : \gamma$` , where $\frac{}{\text{new}} \beta\gamma$. This formulation obviates the need for tracking type variable scopes in the environment. If necessary, any alternate behaviour can be obtained by introducing dummy bindings (ex: through an lambda expression that encompasses all expressions that must be in scope of a type variable). \square

Theorem 9 (Soundness of Heuristic Type Inference). *If: $\Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \parallel \theta$ and $\Vdash \{\Gamma, \Sigma\}$ then, $\theta \parallel \Gamma; \Sigma \vdash e : \tau$.*

Proof. By induction on the derivation of $\Gamma; \Sigma \vdash_{\bar{\tau}} e : \tau \parallel \theta$. We proceed by case analysis on the last step, assuming α -reduction vacuously.

1. Cases I-Unit, I-True, I-False, I-Id, I-Hloc, I-Sloc are trivial.
2. Case I-Lambda:

(a) In this case, we have:

$$\text{i. } \frac{\begin{array}{l} \text{(A) } \Gamma, x \mapsto \alpha; \Sigma \vdash_{\bar{\tau}} e_i : \tau \parallel \theta \\ \text{(B) } \frac{}{\text{new}} \alpha \end{array}}{\Gamma; \Sigma \vdash_{\bar{\tau}} \lambda x. e_i : [\theta \langle \alpha \rangle] \rightarrow [\tau] \parallel \theta}$$

ii. $\Vdash \{\Gamma, \Sigma\}$

We need to show that $\theta \parallel \Gamma; \Sigma \vdash_{\bar{\tau}} \lambda x. e_i : [\theta \langle \alpha \rangle] \rightarrow [\tau]$.

- (b) From case (2.a.ii), conclusion of case (2.a.i) and Lemma 24, we obtain $\Vdash \theta \{ \Gamma, \Sigma, \lambda x. e_i, [\theta \langle \alpha \rangle] \rightarrow [\tau] \}$. This can be re-written as $\Vdash \theta \{ \Gamma, \Sigma, e_i, \alpha, \tau \}$.
- (c) From cases (2.a.ii and 2.a.i.B), we can write $\Vdash \{ \Gamma, x \mapsto \alpha, \Sigma \}$.
- (d) From cases (2.a.i.A and 2.c), from induction hypothesis, we conclude that $\theta \parallel \Gamma, x \mapsto \alpha; \Sigma \vdash_{\bar{\tau}} e_i : \tau$.
- (e) From cases (2.d and 2.b), and the J-Lambda rule in figure 5, we obtain $\theta \parallel \Gamma; \Sigma \vdash_{\bar{\tau}} \lambda x. e_i : [\alpha] \rightarrow [\tau]$. This can be equivalently stated as: $\theta \parallel \Gamma; \Sigma \vdash_{\bar{\tau}} \lambda x. e_i : [\theta \langle \alpha \rangle] \rightarrow [\tau]$.

3. Case I-App:

(a) In this case, we have:

$$\text{i. } \frac{\begin{array}{l} \text{(A) } \Gamma; \Sigma \vdash_{\bar{\tau}} e_1 : \tau_1 \parallel \theta_1 \\ \text{(B) } \theta_1 \parallel \Gamma; \Sigma \vdash_{\bar{\tau}} e_2 : \tau_2 \parallel \theta_2 \\ \text{(C) } \theta_f \vdash_{\text{unf}} \theta_2 \langle \tau_1 \rangle = \beta \downarrow ([\delta] \rightarrow [\alpha]) \\ \text{(D) } \theta_a \vdash_{\text{unf}} \tau_2 = \gamma \downarrow \nabla (\theta_f \langle \delta \rangle) \\ \text{(E) } \tau = \varepsilon \downarrow \nabla (\theta_{a.f} \langle \alpha \rangle) \\ \text{(F) } \frac{}{\text{new}} \alpha \beta \gamma \delta \varepsilon \end{array}}{\Gamma; \Sigma \vdash_{\bar{\tau}} e_1 e_2 : \tau \parallel \theta_{1.2.f.a}}$$

ii. $\Vdash \{\Gamma, \Sigma\}$

iii. $\theta = \theta_{1.2.f.a}$

We need to show that $\theta \parallel \Gamma; \Sigma \vdash_{\bar{\tau}} e_1 e_2 : \tau$.

- (b) From cases (3.a.i.A and 3.a.ii) and induction hypothesis, we can write, $\theta_1 \parallel \Gamma; \Sigma \vdash_{\bar{\tau}} e_1 : \tau_1$.
- (c) From cases (3.a.ii and 3.b) and Lemma 24, we obtain $\Vdash \theta_1 \{ \Gamma, \Sigma, e_1, \tau_1 \}$.

- (d) From case (3.c) and Lemma 13 (weakening), we obtain $\Vdash \theta_1\{\Gamma, \Sigma\}$. Using this result along with case (3.a.i.B) and induction hypothesis, we obtain $\theta_{1.2} \Vdash \Gamma; \Sigma \vdash e_2 : \tau_2$.
- (e) From case (3.d) and Lemma 16, we obtain $\Vdash \theta_2\langle\theta_1\{\Gamma, \Sigma, e_2, \tau_2\}\rangle$.
- (f) From cases (3.b and 3.e), and Lemma 20, we obtain $\theta_{1.2} \Vdash \Gamma; \Sigma \vdash e_1 : \tau_1$.
- (g) From case (3.f) and Lemma 16, we obtain $\Vdash \theta_{1.2}\{\Gamma, \Sigma, e_1, \tau_1\}$.
- (h) From cases (3.a.i.A and 3.a.i.B), and Lemma 25, we obtain: $\text{tv}(\theta_{1.2}\langle e_1, \tau_1 \rangle) \cap \text{tv}(\theta_{1.2}\langle e_2, \tau_2 \rangle) = \text{tv}(\theta_{1.2}\langle \Gamma, \Sigma \rangle)$. This can be written as: $\text{tv}(\theta_{1.2}\langle \Gamma, \Sigma, e_1, \tau_1 \rangle) \cap \text{tv}(\theta_{1.2}\langle \Gamma, \Sigma, e_2, \tau_2 \rangle) = \text{tv}(\theta_{1.2}\langle \Gamma, \Sigma \rangle)$.
- (i) From cases (3.g, 3.e, and 3.h), we obtain $\Vdash \theta_{1.2}\{\Gamma, \Sigma, e_1, \tau_1, e_2, \tau_2\}$.
- (j) It is evident that $\Vdash \theta_{1.2}\{\Gamma, \Sigma, e_1, \tau_1, e_2, \tau_2, \beta\downarrow([\delta] \rightarrow [\alpha])\}$, since β, γ , and δ are new type variables (from case (3.a.i.F)).
- (k) From cases (3.a.i.C, and 3.j) and Lemma 21-conclusion (1), we obtain $\Vdash \theta_{1.2.f}\{\Gamma, \Sigma, e_1, \tau_1, e_2, \tau_2, \beta\downarrow([\delta] \rightarrow [\alpha])\}$.
- (l) From case (3.a.i.C), and Theorem 3-conclusion(1), we have $\theta_{2.f}\langle\tau_1\rangle \preceq: \theta_f\langle\beta\downarrow([\delta] \rightarrow [\alpha])\rangle$ and $\theta_f\langle\beta\downarrow([\delta] \rightarrow [\alpha])\rangle \preceq: \theta_{2.f}\langle\tau_1\rangle$. This can be equivalently written as $\theta_{1.2.f}\langle\tau_1\rangle \preceq: \theta_{1.2.f}\langle\beta\downarrow([\delta] \rightarrow [\alpha])\rangle$ and $\theta_{1.2.f}\langle\beta\downarrow([\delta] \rightarrow [\alpha])\rangle \preceq: \theta_{1.2.f}\langle\tau_1\rangle$.
- (m) From case (3.a.iii), we know that $\theta = \theta_{1.2.f.a}$. Similar to cases (3.j, 3.k, and 3.l), from case (3.a.i.D) we obtain:
- i. $\Vdash \theta\{\Gamma, \Sigma, e_1, \tau_1, e_2, \tau_2, \beta\downarrow([\delta] \rightarrow [\alpha]), \gamma\downarrow\triangleright(\theta_f\langle\delta\rangle)\}$
 - ii. $\theta\langle\tau_2\rangle \preceq: \theta\langle\gamma\downarrow\triangleright(\theta_f\langle\delta\rangle)\rangle$ and $\theta\langle\gamma\downarrow\triangleright(\theta_f\langle\delta\rangle)\rangle \preceq: \theta\langle\tau_2\rangle$.
- (n) i. From case (3.m.i), and Lemma 13 (weakening-conclusion (3)), we obtain $\Vdash \theta\{\Gamma, \Sigma, e_1, \tau_1\}$ From this, case (3.b), and Lemma 20, we obtain $\theta \Vdash \Gamma; \Sigma \vdash e_1 : \tau_1$.
- ii. Similarly, from case (3.d), and Lemma 20, we obtain $\theta \Vdash \Gamma; \Sigma \vdash e_2 : \tau_2$.
- (o) From case (3.m.i), and Lemma 13 (weakening-conclusion (3)), we obtain $\Vdash \theta\{\Gamma, \Sigma, e_1, \tau_1, e_2, \tau_2, \beta\downarrow([\delta] \rightarrow [\alpha])\}$. From this, it is evident that $\Vdash \theta\{\Gamma, \Sigma, e_1, \tau_1, e_2, \tau_2, [\delta] \rightarrow [\alpha]\}$. Now, from cases (3.m.1 and 3.a.i.E), since ε is a new type variable, and $\theta_{a.f}\langle\alpha\rangle = \theta\langle\alpha\rangle$, and $\theta\langle\alpha\rangle \in \theta\langle[\delta] \rightarrow [\alpha]\rangle$, we conclude that $\Vdash \theta\{\Gamma, \Sigma, e_1, \tau_1, e_2, \tau_2, [\delta] \rightarrow [\alpha], \tau\}$.
- (p) From case (3.l) and Lemma 23, we obtain (as one of the conclusions): $\theta\langle\tau_1\rangle \preceq: \theta\langle\beta\downarrow([\delta] \rightarrow [\alpha])\rangle$, from which we can conclude that $\theta\langle\tau_1\rangle \preceq: \theta\langle[\delta] \rightarrow [\alpha]\rangle$.
- (q) From case (3.m.ii), we have $\theta\langle\tau_2\rangle \preceq: \theta\langle\gamma\downarrow\triangleright(\theta_f\langle\delta\rangle)\rangle$, from which, we can conclude that $\theta\langle\tau_2\rangle \preceq: \theta\langle\triangleright(\theta_f\langle\delta\rangle)\rangle$, and therefore $\theta\langle\tau_2\rangle \preceq: \triangleright(\theta\langle\delta\rangle)$, and thus $\theta\langle\tau_2\rangle \preceq: \Im(\theta\langle\delta\rangle)$.
- (r) For any type τ' , it is evident from the definition of $\Im(\tau')$ in figure 6 that $\triangleright(\tau') \preceq: \Im(\tau')$. Therefore, $\triangleright(\theta\langle\alpha\rangle) \preceq: \Im(\theta\langle\alpha\rangle)$. From this, it is evident that $\gamma\downarrow\triangleright(\theta\langle\alpha\rangle) \preceq: \Im(\theta\langle\alpha\rangle)$. Now, from case (3.a.i.E), (since $\theta_{a.f}\langle\alpha\rangle = \theta\langle\alpha\rangle$, and $\theta\langle\tau\rangle = \tau$), we can write $\theta\langle\alpha\rangle \preceq: \Im(\theta\langle\tau\rangle)$.
- (s) Now, from cases (3.n.i, 3.p, 3.n.ii, 3.q, 3.r, and 3.o) and the J-App rule in figure 5, we obtain $\theta \Vdash \Gamma; \Sigma \vdash e_1 e_2 : \tau$.

4. Cases I-If, I-Dup, I-Deref, I-Set, I-Tqexpr, I-Let[Tq] are similar. □

Theorem 10 ((Direct Statemet of) Soundness of Heuristic Type Inference). *If $\Gamma; \Sigma \vdash e : \tau \Vdash \theta_u$ and $\Vdash \{\Gamma, \Sigma\}$, then, $\forall \theta_s$ such that $\theta_s \Vdash_{\text{sat}} \theta_u\{\Gamma, \Sigma, e, \tau\}$, we have $\theta_{s.u} \Vdash \Gamma; \Sigma \vdash e : \tau$.*

Proof. Follows from Theorem 2 and Theorem 9. □

The inference algorithm is not complete. For example, we cannot type the expression $\text{let } id = \text{dup}(\lambda x.x) \text{ in } id^{\wedge} := id^{\wedge}$ without the help of an annotation for id .

8 Related Work

Grossman [5] provides a theory of using quantified types with imperative C style mutation and $\&$ operator for Cyclone. However, his formalization requires explicit annotation for all polymorphic definitions and instantiations. Since C (and Cyclone) have no notion of immutability, both languages require explicit annotation of polymorphism. In contrast, we believe that the best way to integrate polymorphism into the systems programming paradigm is by automatic — albeit incomplete — inference. A further contribution of our work (in comparison to [5]) is that we give a formal specification and proof of correctness of the inference algorithm, not just the type system.

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 [21, 26] (transitive immutability-by-reference), *etc.* These techniques are orthogonal and complementary to the immutability-by-location property in BitC/ℬ. For example, we could have types like $(\text{const } \Psi\tau)$ that can express both global and local usage properties of a location.

A monadic model [14] of mutability is used in pure functional languages like Haskell [15]. 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 different from the normal programming idioms used by systems programmers. For example, Hughes argues that there is no satisfactory way of creating and using global mutable variables using monads [6]. There have been proposals for adding unboxed representation control to Haskell [13, 7]. However, these systems are pure and therefore do not consider the effects of mutability.

Cqual [4] 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. BitC/ℬ supports a polymorphic language and performs simultaneous inference of base types and mutability.

9 Conclusions

In this paper, we have proposed a type system that integrates all of unboxed representation, well-founded first-class mutability, 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. This type system meets the requirements for systems programming languages.

First class mutability introduces challenges for type inference at copy boundaries. There is a fundamental conflict of goals between the inference of principal types and copy compatibility. We have proposed an inference algorithm that resolves this conflict through a combination of hinting mechanisms and selective heuristics. This algorithm is deterministic, does not require whole program analysis, and minimizes programmer annotations in the common case. We have provided a formal framework for this type system, and implemented it as part of the BitC compiler. The source code can be obtained from <http://bitc-lang.org>.

The bootstrap compiler for BitC has been implemented in C++. Currently, the backend emits portable C code. The core of the compiler involves 28,686 lines of C++ code, of which implementation of the type system accounts for about 6,894 lines. An informal description of the inference algorithm for full BitC can be found in [24].

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] 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.
- [5] D. Grossman, “Quantified Types in an Imperative Language” *ACM Transactions on Programming Languages and Systems*, 2006.
- [6] J. Hughes “Global variables in Haskell” *Journal of Functional Programming archive* Volume 14, Issue 5, Sept. 2004.
- [7] 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.
- [8] International Std. Organization *ISO/IEC 8652:1995 (Information Technology — Prog. Languages — Ada)*, 1995.
- [9] International Std. Organization *ISO/IEC FDIS 14882:1998(E) (Prog. Languages - C++)*, 1998.
- [10] International Std. Organization *ISO/IEC 9899:1999 (Prog. Languages - C)*, 1999.
- [11] 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.
- [12] M. P. Jones “Qualified types: theory and practice.” *Cambridge Distinguished Dissertations In Computer Science* ISBN:0-521-47253-9, 1995
- [13] 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
- [14] S. L. Peyton Jones and P. Wadler “Imperative functional programming.” *Proc. ACM SIGPLAN Principles of Programming Languages.*, 1993
- [15] S. L. Peyton Jones (ed.). *Haskell 98 Language and Libraries: The Revised report*. Cambridge University Press, 2003.
- [16] ECMA International “Standard ECMA-334 C# Language Specification” <http://www.ecma-international.org/publications/standards/Ecma-334.htm>
- [17] R. Milner “A theory of type polymorphism in programming.” *Journal of Computer and System Sciences* pp 348-375, 1978.
- [18] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML - Revised* The MIT Press, May 1997.
- [19] J. Gosling, B. Joy, G. Steele, and G. Bracha “The Java Language Specification,” Third Edition <http://java.sun.com/docs/books/jls>
- [20] G. van Rossum, “Python Reference Manual” F. L. Drake, Jr. (ed.) <http://docs.python.org/ref/ref.html>, 2006.
- [21] J. S. Shapiro, J. M. Smith, and D. J. Farber. “EROS: a fast capability system” *ACM Symposium on Operating Systems Principles*, Dec. 1999.
- [22] J. S. Shapiro, S. Sridhar, M. S. Doerrie, “BitC Language Specification” <http://www.bitc-lang.org/docs/bitc/spec.html>
- [23] 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.

- [24] 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.
- [25] 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.
- [26] M. S. Tschantz and M. D. Ernst, “Javari: Adding reference immutability to Java” *Object-Oriented Programming Systems, Languages, and Applications*, Oct 2005.
- [27] A. Wright, “Simple Imperative Polymorphism” *Lisp and Symbolic Comp.* 8(4):343-355, 1995.