

Sound and Complete Type Inference for a Systems Programming Language

Swaroop Sridhar, Jonathan S. Shapiro, and Scott F. Smith

Department of Computer Science, The Johns Hopkins University
swaroop@cs.jhu.edu, shap@cs.jhu.edu, scott@cs.jhu.edu

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

1 Introduction

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

Representation Control. A systems programming language must be expressive enough to specify details of representation including boxed/unboxed data-structure layout and stack/heap allocation. For systems programs, this is both a correctness as well as a performance requirement. Systems programs interact with the hardware through data structures such as page tables whose representation is dictated by the hardware. Conformance to these representation specifications is necessary for correctness. Languages like ML [16] intentionally omit details of representation from the language definition, since this greatly simplifies the mathematical description of the language. Compilers like TIL [24] implement unboxed representation as a discretionary optimization. However, in systems programs, statements about representation are *prescriptive*, not *descriptive*. Formal treatment of representation is required in systems programming languages.

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

Complete Mutability. One of the key features essential for systems programming is support for mutability. The support for mutability must be ‘complete’ in the sense that any location — whether on the stack, heap, or within other unboxed structures — can be mutated. Allocation of mutable cells on the stack boosts performance because (1) the top of the stack is typically accessible from the data cache (2) stack locations are directly addressable and therefore do not require the extra dereferencing involved in the case of heap locations (3) stack allocation does not involve garbage collection overhead. This is particularly important for high confidence and/or embedded kernels as they cannot tolerate unpredictable variance in overhead caused by heap allocation and collection. ML-like languages require all mutable (`ref`) cells to reside on the heap. In pure languages like Haskell [14], the support for mutability is even more restrictive than ML. These restricted models of mutability are insufficiently expressive from a systems programming perspective.

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

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

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

Type Inference and Polymorphism. Type inference achieves the advantages of static typing with a lower burden on the programmer, facilitating rapid

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

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

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

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

The combination of mutability and unboxed representation presents several challenges for type inference. Mutability is an attribute of the *location* storing a value and not the value itself. Therefore, two expressions across a copy boundary (ex: arguments copied at a function call) can differ in their mutability. We refer to this notion of mutability compatibility of types as *copy compatibility*. Copy compatibility creates ramifications for syntax-directed type and mutability inference. Type inference is further complicated due to well known problems with the interaction of mutability and polymorphism [26]. This has forced a second-class treatment of mutability in ML-like languages and a lack of inferred polymorphism in others.

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

2 Informal Overview

In this section, we give an informal description of our type system and inference algorithm. For purposes of presentation in this paper, we define \mathbb{B} , a core systems programming language calculus. \mathbb{B} is a direct expression of lambda calculus with side effects, extended to be able to reflect the semantics of explicit representation.

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

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

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

$$\mathbf{let } cp : \uparrow\mathbf{bool} = \mathbf{dup}(\text{true}) \text{ in } \mathbf{let } p : \uparrow\Psi\mathbf{bool} = cp \quad (* \text{ Error } *)$$

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

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

$$\mathbf{let } f_{nxn} = \lambda x.(x := \text{false}) \text{ in } \mathbf{let } y : \mathbf{bool} = \text{true} \text{ in } f_{nxn} y$$

the type of f_{nxn} is $(\Psi\mathbf{bool}) \rightarrow \mathbf{unit}$, whereas that of the actual argument y is \mathbf{bool} . Since x is a *copy* of y and occupies a different location, this expression is type safe. Thus, we write $\mathbf{bool} \cong \Psi\mathbf{bool}$, where \cong indicates copy compatibility.

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

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

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

2.1 Type Inference

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

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

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

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

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

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

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

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

$\alpha \downarrow \rho \equiv \alpha \setminus \{\alpha \simeq \rho\}$: any type equal to base type ρ except for top level mutability.
 $\varsigma \downarrow \rho \equiv \varsigma \setminus \{\varsigma \cong \rho\}$: any type copy compatible with ρ , where $\varsigma = \alpha$ or $\Psi\alpha$.

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

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

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

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

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

Soundness implications. Like ML, \mathbb{B} enforces the value restriction [26] to preserve soundness of polymorphic typing. This means that the type of x in `let x = e1 in e2` can only be generalized if e_1 is an *immutable* syntactic value. For example, in the expression `let id = λx.x`, the type of id before generalization is $\beta \downarrow (\alpha \rightarrow \alpha)$. However, giving id the generalized type $\forall \alpha \beta. \beta \downarrow (\alpha \rightarrow \alpha)$ is unsound, since it permits expressions such as `let id = λx.x in (id := λx.true, id ())` to type check. We can give id either the polymorphic type $\forall \alpha. \alpha \rightarrow \alpha$, or the monomorphic type $\beta \downarrow (\alpha \rightarrow \alpha)$. However, neither is a principal type for id .

Overloading Polymorphism. Due to the above interaction of polymorphism and unboxed mutability, a traditional HM-style inference algorithm cannot defer decisions about the mutability of types past their generalization. Therefore, current algorithms fix the mutability of types before generalization based on

certain heuristics — thus sacrificing completeness [22]. In order to alleviate this problem, we use a new form of constrained types that range over both mutability and polymorphism.

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

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

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

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

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

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

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

module boundary. For every top-level definition in a module, an arbitrary choice of $\kappa = \psi$ or $\kappa = \forall$ must be made for every surviving $\star_x^\kappa(\tau)$ constraint.

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

3 Formal Description

In order to formalize the semantics of \mathbb{B} , we extend the calculus with stack and heap locations (Fig. 1). Heap locations are first class values, but stack locations are not. Further, we annotate all **let** expressions with a kind — **let** ^{ψ} : monomorphic, possibly mutable definition, and **let** ^{\forall} : polymorphic definitions. The two kinds of **let** expressions have different execution semantics. We write **let** ^{κ} to range over the two kinds of **let** expressions. This distinction is similar to Smith and Volpano’s Polymorphic-C [21]. However, unlike Polymorphic-C, let-kind is *meta syntax*, and is not a part of the input program. The correct kind of **let** is inferred from the static type information. We do not show the semantics for type-qualified expressions as they are trivial.

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

Fig. 1. Extended \mathbb{B} grammar

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

Since the E-Dup and E- \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-:=* rules). The stack is modeled as a pseudo-heap. This enables us to abstract away details such as closure-construction and garbage collection while illustrating the core semantics, as they can later be reified independently.

| Rule | Pre-conditions | Evaluation Step |
|----------------|---|---|
| E-Rval | $S(l) = v$ | $S; H; l \Rightarrow S; H; v$ |
| E-# | $S; H; e \Rightarrow S'; H'; e'$ | $S; H; \mathcal{R}[e] \Rightarrow S'; H'; \mathcal{R}[e']$ |
| E-App | $l \notin \text{dom}(S)$ | $S; H; \lambda x.e v \Rightarrow S; l \mapsto v; H; e[l/x]$ |
| E-If | $b_1 = \text{true} \quad b_2 = \text{false}$ | $S; H; \text{if } b_i \text{ then } e_1 \text{ else } e_2 \Rightarrow S; H; e_i$ |
| E-i | | $S; H; (v_1, v_2).i \Rightarrow S; H; v_i$ |
| E-Dup | $\ell \notin \text{dom}(H)$ | $S; H; \text{dup}(v) \Rightarrow S; H; \ell \mapsto v; \ell$ |
| EL- \wedge # | $S; H; e \Rightarrow S'; H'; e'$ | $S; H; e^\wedge \Rightarrow S'; H'; e'^\wedge$ |
| E- \wedge | $H(\ell) = v$ | $S; H; \ell^\wedge \Rightarrow S; H; v$ |
| E-:=# | $S; H; l \Rightarrow S'; H'; l'$ | $S; H; l := e \Rightarrow S'; H'; l' := e$ |
| E-:=Stack | | $S; l \mapsto v; H; l := v' \Rightarrow S; l \mapsto v'; H; ()$ |
| E-:=Heap | | $S; H; \ell \mapsto v; \ell^\wedge := v' \Rightarrow S; H; \ell \mapsto v'; ()$ |
| E-:=S.p | $v'_{1i} = v_{1i} \quad S; l \mapsto v_i; H; l.p := v'_i$ $\Rightarrow S; l \mapsto v'_i; H; ()$ | $S; l \mapsto (v_1, v_2); H; l.i.p := v'_i$ $\Rightarrow S; l \mapsto (v'_1, v'_2); H; ()$ |
| E-:=H.p | $v'_{1i} = v_{1i} \quad S; H; \ell \mapsto v_i; \ell^\wedge.p := v'_i$ $\Rightarrow S; H; \ell \mapsto v'_i; ()$ | $S; H; \ell \mapsto (v_1, v_2); \ell^\wedge.i.p := v'_i$ $\Rightarrow S; H; \ell \mapsto (v'_1, v'_2); ()$ |
| 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 | | $S; H; \text{let}^\forall x = v_1 \text{ in } e_2 \Rightarrow S; H; e_2[v_1/x]$ |

Fig. 2. Small Step Operational Semantics

| τ | $\Delta(\tau)$ | $\nabla(\tau)$ | $\blacktriangle(\tau)$ | $\blacktriangledown(\tau)$ | $\mathcal{J}(\tau)$ | $\{\tau\}$ |
|-----------------------------|--|---|---|--|--|---|
| α | $\Psi\alpha$ | α | $\Psi\alpha$ | α | α | $\{\alpha\}$ |
| unit | Ψunit | unit | Ψunit | unit | unit | \emptyset |
| bool | Ψbool | bool | Ψbool | bool | bool | \emptyset |
| $\tau_1 \mapsto \tau_2$ | $\Psi(\tau_1 \mapsto \tau_2)$ | $\tau_1 \mapsto \tau_2$ | $\Psi(\tau_1 \mapsto \tau_2)$ | $\tau_1 \mapsto \tau_2$ | $\tau_1 \mapsto \tau_2$ | $\{\tau_1\} \cup \{\tau_2\}$ |
| $\uparrow\tau$ | $\Psi\uparrow\tau$ | $\uparrow\tau$ | $\Psi\uparrow\tau$ | $\uparrow\tau$ | $\uparrow\mathcal{J}(\tau)$ | $\{\tau\}$ |
| $\Psi\rho$ | $\Delta(\rho)$ | $\nabla(\rho)$ | $\blacktriangle(\rho)$ | $\blacktriangledown(\rho)$ | $\mathcal{J}(\rho)$ | $\{\rho\}$ |
| $\tau_1 \times \tau_2$ | $\Psi(\Delta(\tau_1) \times \Delta(\tau_2))$ | $\nabla(\tau_1) \times \nabla(\tau_2)$ | $\Psi(\tau_1 \times \tau_2)$ | $\tau_1 \times \tau_2$ | $\mathcal{J}(\tau_1) \times \mathcal{J}(\tau_2)$ | $\{\tau_1\} \cup \{\tau_2\}$ |
| $\alpha \downarrow \rho$ | $\Delta(\rho)$ | $\nabla(\rho)$ | $\blacktriangle(\rho)$ | $\blacktriangledown(\rho)$ | $\mathcal{J}(\rho)$ | $\{\alpha \downarrow \rho\} \cup \{\rho\}$ |
| $\varsigma \downarrow \rho$ | $\Delta(\rho)$ | $\nabla(\rho)$ | $\blacktriangle(\varsigma) \downarrow \rho$ | $\blacktriangledown(\varsigma) \downarrow \rho$ | $\mathcal{J}(\rho)$ | $\{\varsigma \downarrow \rho\} \cup \{\rho\}$ |
| τ | Immut(τ) | Mut(τ) | $\square(\tau)$ | $\theta(\tau)$ | | |
| α | false | false | false | τ if $[\alpha \mapsto \tau] \in \theta$, else α . | | |
| unit | true | false | true | unit | | |
| bool | true | false | true | bool | | |
| $\tau_1 \mapsto \tau_2$ | true | false | true | $\theta\langle\tau_1\rangle \mapsto \theta\langle\tau_2\rangle$ | | |
| $\uparrow\tau$ | Immut(τ) | Mut(τ) | $\square(\tau)$ | $\uparrow\theta\langle\tau\rangle$ | | |
| $\Psi\rho$ | false | true | $\square(\rho)$ | $\Psi\theta\langle\rho\rangle$ | | |
| $\tau_1 \times \tau_2$ | $\text{Immut}(\tau_1) \wedge \text{Immut}(\tau_2)$ | $\text{Mut}(\tau_1) \vee \text{Mut}(\tau_2)$ | $\square(\tau_1) \wedge \square(\tau_2)$ | $\theta\langle\tau_1\rangle \times \theta\langle\tau_2\rangle$ | | |
| $\alpha \downarrow \rho$ | false | $\text{Mut}(\blacktriangledown(\rho))$ | $\square(\rho)$ | $\alpha' \downarrow \theta\langle\rho\rangle$ if $\theta\langle\alpha\rangle = \alpha'$ $\rho' \downarrow \theta\langle\rho\rangle$ if $\rho' \neq \alpha'$ $\varsigma' \downarrow \theta\langle\rho\rangle$ if $\theta\langle\varsigma\rangle = \varsigma'$ $\varrho \downarrow \theta\langle\varsigma\rangle = \varrho \neq \varsigma'$ | | |
| $\varsigma \downarrow \rho$ | false | $\text{Mut}(\varsigma) \vee \text{Mut}(\nabla(\rho))$ | $\square(\rho)$ | | | |

Fig. 3. Operations and Predicates on Types

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

Static Semantics. Fig. 3 defines several operators and predicates on types that we use in this section. The operators \blacktriangle and \blacktriangledown respectively increase and decrease the shallow top-level mutability of a type. Δ and ∇ maximize / minimize the mutability of a type up to a reference or function boundary. \mathcal{J} removes all mutability in a type up to a function boundary. We write $\tau_1 \stackrel{\Psi}{=} \tau_2$ as shorthand

for $\nabla(\tau_1) = \nabla(\tau_2)$ and $\tau_1 \stackrel{\nabla}{=} \tau_2$ for $\nabla(\tau_1) = \nabla(\tau_2)$. In our algebra of types, the mutable type constructor is idempotent ($\Psi\Psi\tau \equiv \Psi\tau$). We also define the equivalences: $\alpha \downarrow \rho \equiv \alpha \downarrow \rho'$, where $\rho \stackrel{\nabla}{=} \rho'$ and $\varsigma \downarrow \rho \equiv \varsigma \downarrow \rho'$, where $\rho \stackrel{\nabla}{=} \rho'$. The predicates *Immut* and *Mut* identify types that are observably immutable and mutable respectively. The $\square(\tau)$ predicate tests if the type τ is concretizable by fixing variables that range over mutability.

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

Definition 1 (Canonical Expressions). *An expression e is said to be canonical if all `let` expressions in e are annotated with one of the kinds ψ or \forall .*

Definition 2 (Consistency of Constrained types). *Let $\text{mtv}(\bar{\tau})$, $\text{Mtv}(\bar{\tau})$, and $\text{ntv}(\bar{\tau})$ be the set of all type variables appearing in $\{\bar{\tau}\}$ constrained by $\alpha \downarrow \rho$, by $\varsigma \downarrow \rho$ and unconstrained respectively. We say that the set of types $\{\bar{\tau}\}$ is consistent, written $\Vdash \{\bar{\tau}\}$, if: (1) For all $\{\alpha \downarrow \rho, \alpha \downarrow \rho'\} \subseteq \{\bar{\tau}\}$, we have $\rho \stackrel{\nabla}{=} \rho'$. (2) For all $\{\varsigma \downarrow \rho, \varsigma' \downarrow \rho'\} \subseteq \{\bar{\tau}\}$ such that $\varsigma \stackrel{\nabla}{=} \varsigma'$, we have $\rho \stackrel{\nabla}{=} \rho'$. (3) $\text{mtv}(\bar{\tau})$, $\text{Mtv}(\bar{\tau})$, and $\text{ntv}(\bar{\tau})$ are mutually exclusive.*

Definition 3 (Consistency of substitutions). *A substitution θ is said to be consistent over a set of types $\{\bar{\tau}\}$, written $\theta \Vdash \{\bar{\tau}\}$ if: (1) $\Vdash \theta\{\bar{\tau}\}$. (2) For all $\alpha \downarrow \rho \in \{\bar{\tau}\}$, we have $\theta\langle\alpha\rangle = \beta$, or $\theta\langle\alpha\rangle = \rho'$ such that $\rho' \stackrel{\nabla}{=} \theta\langle\rho\rangle$. (3) For all $\varsigma \downarrow \rho \in \{\bar{\tau}\}$, we have $\theta\langle\varsigma\rangle = \varsigma'$, or $\theta\langle\varsigma\rangle = \varrho$ such that $\varrho \stackrel{\nabla}{=} \theta\langle\rho\rangle$.*

Definition 4 (Consistency of \star constraints). *A set of \star constraints \mathcal{D} is said to be consistent, written $\models \mathcal{D}$ if: (1) For all $\star_x^\forall(\tau) \in \mathcal{D}$, we have *Immut*(τ). (2) For all $\star_x^\psi(\tau_1) \dots \star_x^\psi(\tau_n) \in \mathcal{D}$, we have $\tau_1 = \dots = \tau_n$. (3) For all $\star_x^{\varkappa_1}(\tau_1) \in \mathcal{D}$ and $\star_y^{\varkappa_2}(\tau_2) \in \mathcal{D}$, $\varkappa_1 \neq \varkappa_2$ implies $x \neq y$.*

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

The type judgment $\mathcal{D}; \Gamma; \Sigma \vdash e : \tau$ is understood as: given a binding environment Γ and store typing Σ , the expression e has type τ subject to the set of \star constraints \mathcal{D} . We write $e \preceq : \tau$ as a shorthand for $e : \tau'$ and $\tau' \preceq : \tau$, for some type τ' . The rule T-Lambda permits the interface type of a function to be different from its internal type, as explained in Sec. 2.1. The rule T-App introduces copy-coercions at argument and return positions of an application. T-Let-M rule

$$\begin{array}{c}
\begin{array}{c}
\text{S-Refl} \\
\frac{}{\tau \triangleq: \tau} \\
\text{S-Trans} \\
\frac{\tau_0 \triangleq: \tau_1 \quad \tau_1 \triangleq: \tau_2}{\tau_0 \triangleq: \tau_2} \\
\text{S-Mut} \\
\frac{\rho \triangleq: \rho'}{\Psi\rho \triangleq: \Psi\rho'} \\
\text{S-Pair} \\
\frac{\tau_1 \triangleq: \tau'_1 \quad \tau_2 \triangleq: \tau'_2}{\tau_1 \times \tau_2 \triangleq: \tau'_1 \times \tau'_2} \\
\text{S-Mt1} \\
\frac{\nabla(\rho) = \tau}{\alpha\downarrow\rho \triangleq: \tau} \\
\text{S-Mt2} \\
\frac{\blacktriangle(\rho) = \tau}{\tau \triangleq: \alpha\downarrow\rho} \\
\text{S-Mf1} \\
\frac{\nabla(\rho) = \tau}{\alpha\downarrow\rho \triangleq: \tau} \\
\text{S-Mf2} \\
\frac{\alpha\downarrow\rho \triangleq: \rho'}{\Psi\alpha\downarrow\rho \triangleq: \Psi\rho'} \\
\text{S-MF3} \\
\frac{\Delta(\rho) = \tau}{\tau \triangleq: \varsigma\downarrow\rho}
\end{array} \\
\text{T-Unit} \\
\frac{}{\emptyset; \Gamma; \Sigma \vdash () : \text{unit}} \\
\text{T-Bool} \\
\frac{}{\emptyset; \Gamma; \Sigma \vdash \text{b} : \text{bool}} \\
\text{T-Id} \\
\frac{\Gamma(x) = \forall \bar{\alpha}. \tau \setminus \mathcal{D} \quad \theta \Vdash \{\tau, \mathcal{D}\} \quad \text{dom}(\theta) = \{\bar{\alpha}\}}{\emptyset(\mathcal{D}); \Gamma; \Sigma \vdash x : \theta(\tau)} \\
\text{T-Hloc} \\
\frac{\Sigma(\ell) = \tau}{\emptyset; \Gamma; \Sigma \vdash \ell : \uparrow\tau} \\
\text{T-Sloc} \\
\frac{\Sigma(l) = \tau}{\emptyset; \Gamma; \Sigma \vdash l : \tau} \\
\text{T-Lambda} \\
\frac{\mathcal{D}; \Gamma, x \mapsto \tau_1; \Sigma \vdash e : \tau_2 \quad \tau_1 \stackrel{\forall}{=} \tau'_1 \quad \tau_2 \stackrel{\forall}{=} \tau'_2}{\mathcal{D}; \Gamma; \Sigma \vdash \lambda x. e : \tau'_1 \rightarrow \tau'_2} \\
\text{T-App} \\
\frac{\mathcal{D}_1; \Gamma; \Sigma \vdash e_1 \triangleq: \tau_a \rightarrow \tau_r \quad \mathcal{D}_2; \Gamma; \Sigma \vdash e_2 \triangleq: \nabla(\tau_a) \quad \Delta(\tau_r) \triangleq: \tau}{\mathcal{D}_1 \cup \mathcal{D}_2; \Gamma; \Sigma \vdash e_1 e_2 : \tau} \\
\text{T-If} \\
\frac{\mathcal{D}_1; \Gamma; \Sigma \vdash e_1 \triangleq: \text{bool} \quad \mathcal{D}_2; \Gamma; \Sigma \vdash e_2 \triangleq: \tau \quad \mathcal{D}_3; \Gamma; \Sigma \vdash e_3 \triangleq: \tau \quad \tau' \triangleq: \tau}{\mathcal{D}_1 \cup \mathcal{D}_2 \cup \mathcal{D}_3; \Gamma; \Sigma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau'} \\
\text{T-Pair} \\
\frac{\mathcal{D}_1; \Gamma; \Sigma \vdash e_1 \triangleq: \tau_1 \quad \mathcal{D}_2; \Gamma; \Sigma \vdash e_2 \triangleq: \tau_2 \quad \tau'_1 \triangleq: \tau_1 \quad \tau'_2 \triangleq: \tau_2}{\mathcal{D}_1 \cup \mathcal{D}_2; \Gamma; \Sigma \vdash (e_1, e_2) : \tau'_1 \times \tau'_2} \\
\text{T-Sel} \\
\frac{\mathcal{D}; \Gamma; \Sigma \vdash e : \tau \quad \tau \stackrel{\forall}{=} \tau_1 \times \tau_2}{\mathcal{D}; \Gamma; \Sigma \vdash e.i : \tau_i} \\
\text{T-Set} \\
\frac{\mathcal{D}_1; \Gamma; \Sigma \vdash l \triangleq: \Psi\rho \quad \mathcal{D}_2; \Gamma; \Sigma \vdash e \triangleq: \rho}{\mathcal{D}_1 \cup \mathcal{D}_2; \Gamma; \Sigma \vdash l := e : \text{unit}} \\
\text{T-Dup} \\
\frac{\mathcal{D}; \Gamma; \Sigma \vdash e \triangleq: \tau \quad \tau' \triangleq: \tau}{\mathcal{D}; \Gamma; \Sigma \vdash \text{dup}(e) : \uparrow\tau'} \\
\text{T-Deref} \\
\frac{\mathcal{D}; \Gamma; \Sigma \vdash e \triangleq: \uparrow\tau}{\mathcal{D}; \Gamma; \Sigma \vdash e^\wedge : \tau} \\
\text{T-Let-M} \\
\frac{\mathcal{D}_1; \Gamma; \Sigma \vdash e_1 \triangleq: \tau_1 \quad \tau \triangleq: \tau_1 \quad \mathcal{D}_2; \Gamma, x \mapsto \tau; \Sigma \vdash e_2 : \tau_2}{\mathcal{D}_1 \cup \mathcal{D}_2; \Gamma; \Sigma \vdash (\text{let}^\psi x = e_1 \text{ in } e_2) : \tau_2} \\
\text{T-Let-MP} \\
\frac{\mathcal{D}_1; \Gamma; \Sigma \vdash v \triangleq: \tau_1 \quad \tau \triangleq: \tau_1 \quad \mathcal{D} = \mathcal{D}_1 \cup \{\star_x^\tau(\tau)\} \quad \{\bar{\alpha}\} = \text{ftv}(\tau, \mathcal{D}) \setminus \text{ftv}(\Gamma, \Sigma)}{\mathcal{D}_2; \Gamma, x \mapsto \forall \bar{\alpha}. \tau \setminus \mathcal{D}; \Sigma \vdash e : \tau_2 \quad \frac{}{\overline{\text{fresh}} \bar{\beta}}}{\mathcal{D}[\bar{\beta}/\bar{\alpha}] \cup \mathcal{D}_2; \Gamma; \Sigma \vdash (\text{let}^x x = v \text{ in } e) : \tau_2}
\end{array}$$

Fig. 4. Declarative Type Rules

types `let` expressions monomorphically, and thus requires a `letψ` annotation. In this case, the expression e_1 is permitted to be expansive (i.e. need not be a syntactic value v). The `T-Let-MP` rule types `let` expressions where the expression being bound is a syntactic value. It assigns x a constrained type scheme along with the constraint $\star_x^\tau(\tau)$. The `T-Id` rule instantiates types and constraints. The instantiated constraints are collected over the entire derivation, so that we can enforce instantiation consistency. $\frac{}{\overline{\text{fresh}} \bar{\alpha}}$ identifies fresh type variables.

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

The interesting case is the safety of polymorphic `let` expressions. The `T-Let-MP` rule does not require that the type τ being quantified over be immutable, but adds the $\star_x^\tau(\tau)$ constraint. Now, if we have a derivation $\mathcal{D}; \Gamma; \Sigma \vdash e : \tau$

such that $\models \mathcal{D}$, then one of the two cases must follow. (1) If any instantiation of τ is mutable, then $\varkappa = \psi$. In this case, execution proceeds through the E-Let-M rule, which create a stack location for x . Therefore, x is permitted to be the target of an assignment. $\models \mathcal{D}$ guarantees that all instantiations of τ are identical, which ensures that the type of a location cannot change. (2) If τ is instantiated polymorphically, then $\varkappa = \forall$. Execution proceeds through the E-Let-P rule, which performs a value substitution. Here, $\models \mathcal{D}$ guarantees that all instantiations are deeply immutable. Therefore, x cannot be directly used (in the forms x or $x.p$) as the target of an assignment, which ensures that the value substitution cannot lead to a stuck state.

Definition 5 (Consistent Type Derivation). Let $\{\{\mathcal{D}; \Gamma; \Sigma \vdash e : \tau\}\}$ denote the extension $\{\}\{\}$ function to the set of all types used in the derivation of $\mathcal{D}; \Gamma; \Sigma \vdash e : \tau$. We say that $\mathcal{D}; \Gamma; \Sigma \vdash_* e : \tau$ is a consistent derivation if $\mathcal{D}'; \Gamma; \Sigma \vdash e : \tau$ for some $\mathcal{D}' \subseteq \mathcal{D}$, and $\Vdash \{\{\mathcal{D}\}\} \cup \{\{\mathcal{D}'\}; \Gamma; \Sigma \vdash e : \tau\}$.

Definition 6 (Stack and Heap Typing) A heap H and a stack S are said to be well typed with respect to Γ, Σ and \mathcal{D} , written $\mathcal{D}; \Gamma; \Sigma \vdash_* H + S$, if:

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

Definition 7 (Valid Lvalues). We say that an lvalue \mathcal{L} is valid with respect to a stack S and heap H , written $H + S \vdash_v \mathcal{L}$ if for some p , either (1) $\mathcal{L} = l$ or $\mathcal{L} = l.p$ where $l \in \text{dom}(S)$; or (2) $\mathcal{L} = \ell^\wedge$ or $\mathcal{L} = \ell^\wedge.p$ where $l \in \text{dom}(H)$.

Lemma 1 (Progress). If e is a closed canonical well typed expression, that is, $\mathcal{D}; \emptyset; \Sigma \vdash_* e : \tau$ for some τ and Σ , given any heap and stack such that $\mathcal{D}; \emptyset; \Sigma \vdash_* H + S$,
 (1) If e is a left expression ($e = l$), then e is either a valid lvalue (that is, $e = \mathcal{L}$ and $H + S \vdash_v \mathcal{L}$) or else $\exists e', S', H'$ such that $S; H; e \Rightarrow S'; H'; e'$.
 (2) e is a value v or else $\exists e', S', H'$ such that $S; H; e \Rightarrow S'; H'; e'$.

Lemma 2 (Preservation). For any canonical expression e , if $\mathcal{D}; \Gamma; \Sigma \vdash_* e : \tau$, $\mathcal{D}; \Gamma; \Sigma \vdash_* H + S$ and $\models \mathcal{D}$ then,

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

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

Type Inference Algorithm. Type inference is a program transformation that accepts a program in which `let` expressions are not annotated with their kinds,

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

Fig. 5. Type Inference Algorithm

and returns the same program with **let** expressions annotated with their kinds and all expressions annotated with their types. The type inference algorithm is shown in Fig. 5. The inference judgment $\Gamma; \Sigma \Vdash e : \tau \mid \mathcal{C}$ is understood as: given a binding environment Γ and store typing Σ , the expression e has type τ subject to the constraints \mathcal{C} .

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

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

The U-Ct* rules perform unification of constrained types with other constrained or unconstrained types. First, immutable versions of the two types are unified to establish compatibility (through constraints involving $\overset{\nabla}{=}$ and $\overset{\nabla}{\equiv}$). Then, the constrained type is made to exactly equal the other type by unifying its variable part with the other type. The key observation here is that the copy

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

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

Fig. 6. Unification Algorithm

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

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

Definition 9 (Constraint Satisfaction). *The satisfaction of a constraint set \mathcal{C} by a substitution θ is defined as follows.*

$$\frac{\forall (\tau_1 = \tau_2) \in \mathcal{C}, \theta(\tau_1) = \theta(\tau_2) \quad \forall (\kappa = \varkappa) \in \mathcal{C}, \theta(\kappa) = \theta(\varkappa) \quad \theta \vdash_{\text{sol}} \mathcal{C} \rightsquigarrow \mathcal{D}}{\theta \vdash_{\text{sol}} \mathcal{C} \rightsquigarrow \mathcal{D}} \quad \frac{\mathcal{D} = \{\theta(\star_x^\kappa(\tau)) \mid \star_x^\kappa(\tau) \in \mathcal{C}\}}{\theta \vdash_{\text{sat}} \mathcal{C} \rightsquigarrow \mathcal{D}}$$

Definition 10 (Notational Derivations). *We write:*

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

Lemma 3 (Correctness of Unification). *If $\mathcal{U}(\mathcal{C}) = (\mathcal{D}, \theta)$, then $\theta \vdash_{\text{sol}} \mathcal{C} \rightsquigarrow \mathcal{D}$*

Lemma 4 (Satisfiability of Unified Constraints). *If $\mathcal{U}(\mathcal{C}) = (\mathcal{D}, \theta_u)$, then there exists a substitution θ_s such that $\theta_u \circ \theta_s \vdash_{\text{sat}} \mathcal{C} \rightsquigarrow \mathcal{D}$.*

Lemma 5 (Principality of Unification). *If $\mathcal{U}(\mathcal{C}) = (\mathcal{D}, \theta_u)$, where \mathcal{C} is a set of constraints obtained from the type inference algorithm, then, for all θ_s such that $\theta_s \vdash_{\text{sol}} \mathcal{C} \rightsquigarrow \mathcal{D}'$, we have $\theta_s \supseteq \theta_u$.*

Lemma 6 (Decidability of Unification). *The problem of computing a canonical derivation of $\mathcal{U}(\mathcal{C})$ for an arbitrary \mathcal{C} , where no two applications of $U\text{-Sym}$ rule happen consecutively is decidable.*

Theorem 2 (Soundness of Type Inference). *If $\theta; \Gamma; \Sigma \vdash_{\bar{i}} e : \tau \mid \mathcal{D}$, then $\theta; \mathcal{D}; \Gamma; \Sigma \vdash_{\bar{*}} e : \tau$.*

Lemma 7 (Type Checkability). *If $\Gamma; \Sigma \vdash_{\bar{i}} e : \tau \mid \mathcal{C}$ and $\mathcal{U}(\mathcal{C}) = (\mathcal{D}, \theta)$, then $\exists \theta'$ such that $\models \theta' \langle \mathcal{D} \rangle$ and $\theta \circ \theta' \langle e \rangle$ is canonical, and $\theta \circ \theta'; \mathcal{D}; \Gamma; \Sigma \vdash_{\bar{*}} e : \tau$.*

Theorem 3 (Completeness of Type Inference). *If $\theta; \mathcal{D}; \Gamma; \Sigma \vdash_{\bar{*}} e : \tau$, then there exists a $\theta' \supseteq \theta$ such that $\theta'; \Gamma; \Sigma \vdash_{\bar{i}} e : \tau \mid \mathcal{D}$.*

Proofs for safety of the type system and soundness and completeness of the inference algorithm can be found in [23].

4 Related Work

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

C's `const` notion of immutability-by-alias offers localized checking of immutability properties, and encourages good programming practice by serving as documentation of programmers' intentions. Other systems have proposed immutability-by-name [2], referential immutability [19,25] (transitive immutability-by-reference), *etc.* These techniques are orthogonal and complementary to the immutability-by-location property in \mathbb{B} . For example, we could have types like `(const $\Psi\tau$)` that can express both global and local usage properties of a location.

A monadic model [13] of mutability is used in pure functional languages like Haskell [14]. In this model, the type system distinguishes side-effecting computations from pure ones (and not just mutable locations from immutable ones). Even though this model is beneficial for integration with verification systems, it is considerably removed from the idioms needed by systems programmers.

For example, Hughes argues that there is no satisfactory way of creating and using global mutable variables using monads [7]. There have been proposals for adding unboxed representation control to Haskell [12,8]. However, these systems are pure and therefore do not consider the effects of mutability.

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

5 Conclusions

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

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

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

References

1. Biagioni, E., Harper, R., Lee, P.: A network protocol stack in Standard ML. *Higher Order and Symbolic Computation* 14(4) (2001)
2. Deline, R., Fähndrich, M.: VAULT: a programming language for reliable systems (2001), <http://research.microsoft.com/vault>
3. Derby, H.: The performance of FoxNet 2.0. Technical Report CMU-CS-99-137 School of Computer Science, Carnegie Mellon University (June 1999)
4. ECMA International Standard ECMA-334 C# Language Specification, <http://www.ecma-international.org/publications/standards/Ecma-334.htm>
5. Foster, J.S., Johnson, R., Kodumal, J., Aiken, A.: Flow-Insensitive Type Qualifiers. *Trans. on Programming Languages and Systems*. 28(6), 1035–1087 (2006)
6. Grossman, D.: Quantified Types in an Imperative Language. *ACM Transactions on Programming Languages and Systems* (2006)

7. Hughes, J.: Global variables in Haskell. *Journal of Functional Programming* archive 14(5) (September 2004)
8. Diatchki, I.S., Jones, M.P., Leslie, R.: High-level Views on Low-level Representations. In: *Proc. ACM Int. Conference on Functional Programming*, pp. 168–179 (2005)
9. International Std. Organization ISO/IEC 9899:1999 (Prog. Languages - C) (1999)
10. Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., Wang, Y.: Cyclone: A safe dialect of C. In: *Proc. of USENIX Annual Technical Conference*, pp. 275–288 (2002)
11. Jones, M.P.: Qualified types: theory and practice. *Cambridge Distinguished Dissertation in Computer Science* (1995) ISBN:0-521-47253-9
12. Peyton Jones, S.L., Launchbury, J.: Unboxed values as first class citizens in a non-strict functional language. *Functional Programming Languages and Computer Architecture* (1991)
13. Peyton Jones, S.L., Wadler, P.: Imperative functional programming. In: *Proc. ACM SIGPLAN Principles of Programming Languages* (1993)
14. Peyton Jones, S.L. (ed.): *Haskell 98 Language and Libraries: The Revised report*. Cambridge University Press, Cambridge (2003)
15. Milner, R.: A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 348–375 (1978)
16. Milner, R., Tofte, M., Harper, R., MacQueen, D.: *The Definition of Standard ML - Revised*. The MIT Press, Cambridge (1997)
17. Gosling, J., Joy, B., Steele, G., Bracha, G.: *The Java Language Specification*, 3rd edn., <http://java.sun.com/docs/books/jls>
18. van Rossum, G.: *Python Reference Manual*. In: Drake Jr., F.L., ed. (2006), <http://docs.python.org/ref/ref.html>
19. Shapiro, J.S., Smith, J.M., Farber, D.J.: EROS: a fast capability system. In: *ACM Symposium on Operating Systems Principles* (December 1999)
20. Shapiro, J.S., Sridhar, S., Doerrie, M.S.: *BitC Language Specification*, <http://www.bitc-lang.org/docs/bitc/spec.html>
21. Smith, G., Volpano, D.: A sound polymorphic type system for a dialect of C. *Science of Computer Programming* 32(2–3), 49–72 (1998)
22. Sridhar, S., Shapiro, J.S.: Type Inference for Unboxed Types and First Class Mutability. In: *Proc. 3rd Workshop on Prog. Languages and Operating Systems* (2006)
23. Sridhar, S., Shapiro, J.S., Smith, S.F.: Sound and Complete Type Inference in BitC. Technical Report SRL-2008-02, Systems Research Laboratory, The Johns Hopkins University (2008)
24. Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., Lee, P.: TIL: A type-directed optimizing compiler for ML. In: *Proc. ACM SIGPLAN PLDI* (1996)
25. Tschantz, M.S., Ernst, M.D.: Javari: Adding reference immutability to Java. *Object-Oriented Programming Systems, Languages, and Applications* (October 2005)
26. Wright, A.: Simple Imperative Polymorphism. *Lisp and Symbolic Comp.* 8(4), 343–355 (1995)