

Formalization of the BitC Type System

SRL Technical Report SRL2007-01

Swaroop Sridhar, Jonathan S. Shapiro, Ph.D., Scott F. Smith, Ph.D.

Dept. of Computer Science
Johns Hopkins University

July 30, 2010

Contents

1	Abstract	1
2	Introduction	3
3	Mutability Model and Type Inference	5
3.1	BitC	5
3.2	Mutability Model in BitC	6
3.3	Copy Compatibility	7
3.4	Type Inference	8
3.4.1	Why Should We Infer Mutability?	8
3.4.2	Incompleteness of Inference	8
3.4.3	Inference Considerations	9
3.5	Type Inference in BitC	10
4	Formalization	13
4.1	The Language	13
4.2	Dynamic Semantics	13
4.3	Static Semantics	15
4.3.1	Copy Compatibility	15
4.3.2	Compatibility of Function Types	15
4.3.3	Copy Coercions	16
4.3.4	Location Semantics	16
4.3.5	Declarative Type Rules	16
4.3.6	Generalization	17
4.3.7	Soundness of Declarative system	18
4.3.8	Equational Inference Algorithm	23
4.3.9	Unification	24
4.3.10	Soundness of Equational Inference	25
4.3.11	Inference with Eager Unification	26
4.3.12	Unification	27
4.3.13	Solving Copy Compatibility Constraints	28
4.3.14	Soundness of Eager Inference	28

5	Implementation	45
6	Related Work	47
7	Conclusion	49

Chapter 1

Abstract

There is a persistent gap between modern language designs and the requirements of systems programmers. Systems programs rely on fine-grain control of data representation and use of state to achieve performance, conformance to hardware specification, and temporal predictability. Modern type systems such as those used in ML and Haskell rely on boxed representation of composite types and restricted support for mutability to enable features such as polymorphism, type inference, and sound type systems. No current language fully supports both feature sets, partly because no mutability model has been proposed that adequately combines explicitly unboxed types with consistent typing of mutability. C’s “const” type qualifier is unsound, while from a systems programming perspective ML’s “ref” construct is insufficiently expressive.

This paper introduces a new type system in which deep mutability is a first-class component of type. The type system is provably sound, expresses unboxed composite types, supports polymorphism and type inference, and reduces the amount of state that must be handled by stateful verification methods. The resulting system integrates these features in a way that is subjectively natural to systems programmers — in particular supporting naïve programmer intuitions about locations. A key element of this success is the adoption of certain “hinting” mechanisms that guide the inference process to the programmer-expected result. A practical and efficient implementation of this type system and inference mechanism has been constructed as part of the BitC programming language.

Chapter 2

Introduction

Safe systems programming is a focal topic for researchers in systems as well as programming languages and verification communities in the recent years. Although there seems to be a consensus about the fact that thirty years of programming in high level assembly is long enough [5], the level of correctness guarantee sought in the proposed alternatives vary greatly — ranging from memory safety [17, 27] to static analysis or model checking to validate certain safety properties [2] to full semantic correctness verification [12]. We are currently working on the Coyotos project [39], which investigates the possibility of a fully verified implementation of a secure microkernel. In order to achieve this, there is a need for a language framework that has the the right combination of expressiveness, formally founded semantics, control over low-level representation, and integration with modern verification technology.

Modern programming languages such as ML [26] or Haskell [19] provide newer, stronger, and more expressive type systems than systems programming languages such as C [22, 16] or Ada [15]. These features improve the robustness and safety of programs, and it is desirable to incorporate them into languages that can be used for high-performance systems codes. Polymorphism facilitates better code reuse and organization. Type inference achieves the consistency advantages of static typing with a lower burden on the programmer, facilitating more rapid prototyping and development.

A key property of these ML-like languages is that the mutability model is a part of the type system. A term subset language suitable for modern logical frameworks [29, 28, 21, 6] can be achieved simply by removing the “ref” construct. ML Types are decisive about the mutability of all locations (memory cells): every location has one and only one type across all aliases. Tools that perform static analysis or model checking benefit from the ML mutability model because conclusions drawn about location immutability need never be conservative [1]. This also reduces the state space that must be examined by stateful reasoning techniques. Since mutable values must not be let-polymorphic [50], automatic inference of polymorphism requires a mathematically well-founded notion of mutability.

Unfortunately, most modern programming language implementations do not support certain key features that are essential for systems programming — most notably support for low-level representation management, and support for first class mutability. The support for mutability must be first class in the sense that a value of any type (and not just references) can be mutable, and we should be able to specify mutability at field level granularity.

Efforts have been made to retrofit safety and other high level language features into C-like low level languages. Systems like CCured [27] and Cyclone [17] use a combination of sound static analysis techniques, dynamic checks, user annotations, pointer restrictions and conservative garbage collection to ensure memory safety of C (or C-like) programs. First, a fundamental problem with the safe C-like languages is the lack of a rigorous semantic specification. Second, these systems conform to the C model of mutability, where all data is mutable (that is, types do not authentically distinguish mutable and immutable values see section 3.2). This promiscuity of mutability presents a great challenge for integration into a verification framework — we will either have to perform complex and whole program alias analyses, or draw conservative and weak conclusions about the semantic behavior of programs. The mutability model also limits the ability to perform polymorphic type inference. For example, Cyclone supports first class polymorphism only for function definitions [53] that are explicitly annotated with a polymorphic type.

In this paper, we propose a new language BitC [55] which integrates all of the desirable features mentioned above

into a single, consistent language framework. BitC is a type safe, higher order programming language that exposes machine-level representation of types, supports polymorphic type inference and well-founded first class mutability. BitC is a call-by-value expression language. Support for unboxed mutability means that we can allow some freedom in the compatibility of types with respect to their mutability at copy boundaries. This kind of compatibility has interesting ramifications for type inference, because there is no longer a unique way to type an expression. In this paper, we discuss some of these issues, and present a solution based on a simple extension to the Hindley-Milner inference algorithm [25]. A partial sketch of this approach was given in an earlier workshop paper [38]; the current paper presents the first complete treatment, consisting of a formalization and proof of soundness, as well as an implementation.

Chapter 3

Mutability Model and Type Inference

3.1 BitC

In this section, we give a brief introduction to BitC and the facilities available in BitC to suit systems programming. BitC supports a rich set of primitive datatypes and bit-fields: `int8`, `int64`, `(bitfield uint32 8)`, `float`, `double`, *etc.* It also supports type classes [18] to support operator overloading over these types. For example:¹

```
(define (inc x) (+ x 1))
inc: (forall ((Arith 'a) (IntLit 'a)) (fn ('a) 'a))
```

Like C, BitC provides full control over data structure representation, which is necessary for high-performance systems programming. Composite types (structures and unions) may be explicitly declared as boxed (`:ref`) or unboxed (`:val`). The default representation is boxed. For example:

```
(defstruct (pair 'a 'b):val fst: 'a snd: 'b)
(defunion (list 'a) nil
          (cons car:'a cdr:(list 'a)))
```

BitC also has homogeneous aggregate types in the form of arrays (unboxed) and vectors (boxed).

BitC is a stateful language. Variables or individual fields may be given a mutable type. In the following example, `rec` defines an unboxed structure in which one of the fields is mutable, while `xyz` is a stack variable that is mutable.

```
(defstruct rec:val id:uint32 mVal:(mutable int64))
(let ((xyz:(mutable bool) #t)) ...)
xyz: (mutable bool)
```

The `dup` operator performs a heap copy and returns the corresponding heap location. For example:

```
(define bPtr (dup #t))
bPtr: (ref bool)
```

Unlike ML, heap copy does not entail mutability. The type of `bPtr` in the above example is `(ref bool)`, which just states that `bPtr` is a reference (pointer) to a location containing an immutable value of type `bool`. Expressions that have a reference type can later be dereferenced through the `deref` (or `^`) operator. BitC does not have an address-of (`&`) operator to obtain the address of stack locations. As in the case of most safe languages, heap locations are first class values, but stack locations are not. We use the unqualified term “location” wherever both stack and heap locations are applicable.

¹ We use `texttt` font to show program fragments and an emphasized `texttt` font to show the inferred types.

BitC supports let-polymorphism as in ML. Polymorphism is supported even over unboxed types. However, in some cases, we may want to restrict the polymorphism to reference types only, in order to ensure that the definition is not polyinstantiated (or otherwise adjusted to handle unboxed types). In BitC, there is a built in type classes called `ref-type` to which all boxed types belong. Now, we can write a polymorphic identity function that only works on reference types as:

```
(define id:(forall ((ref-type 'a)) (fn ('a) 'a))
  (lambda (x) x))
```

There are several syntactic constructs in BitC to support a metalanguage, which allows application-specific safety properties to be embedded within a program. This meta-language will eventually be interfaced with a theorem prover in order to discharge the proof obligations expressed by the programmer.

3.2 Mutability Model in BitC

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.”

The other well known model of mutability is the C model, wherein the support for mutability is “first-class” in the sense that mutation of stack variables and unboxed values are supported. 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 (and safe-C languages) it is legal to write:

```
const int *cpi = ...;
int *pi = cpi;    // Warning only.
*pi = 5;         // OK!
```

The alleged “constness” of the location pointed to by `cpi` is a local property with (only) respect to this alias (`cpi`) and not a statement of true immutability of the location referenced by it. The compiler or other analytical engines are not entitled to believe that certain locations or fields are constant even if so declared.

BitC supports well-founded first class mutability. Similar to ML, we impose the the “one location, one type” rule.

```
(let ((cpi:(ref int32) (dup 10)))
  (let ((pi:(ref (mutable int32)) cpi)) ;ERROR
```

In order to support unboxed mutability, we still need to have a notion of *lvalues*. It is necessary for both preserving the programmer’s mental model of the relationship between locations and storage, as well as ensuring that compiler transformations are semantics preserving. In an assignment context, the following syntactic forms in BitC accept only *lvalues* at positions indicated as *lval*, and return *lvalues* (except `set!`, which returns `unit`):

```
id
(array-nth lval ndx)
(vector-nth e ndx)
(member lval ident)
(deref e)
(set! lval e)
```

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 [7] (a simplified form of `const`), referential immutability [35] (immutability-by-reference that can be enforced shallowly or transitively) *etc.*, which have versatile applications. These techniques are orthogonal and complementary to the immutability-by-location property that we have in BitC. For example, we could have types like `(const (mutable τ))` that can express both global and local usage properties of a location.

3.3 Copy Compatibility

Since BitC 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 argument passing, assignment, and binding boundaries. We will refer to this as **copy compatibility**, denoted by \cong . For example:

```
(define (plus1 x:(mutable int32))
  (set! x (+ x 1)) x)
plus1: (fn (mutable int32) int32)

(define v1 (plus1 10:int32))
v1: int32
```

In the application `(plus1 10:int32)` above, the type of the actual parameter `10` is `int32` and that of the formal parameter `x` is `(mutable int32)`. Here, we allow $\text{int32} \cong (\text{mutable int32})$.

Copy compatibility need not be restricted to the outermost mutability compatibility, but must not extend past a reference boundary. This is necessary to enforce the invariant that every location must have a unique type, since the target of the reference is not copied. We define copy compatibility as follows:

- $\tau \cong (\text{mutable } \tau)$ for any type τ (direct compatibility).
- $(\text{array } \tau) \cong (\text{array } (\text{mutable } \tau))$. Arrays are unboxed types, the entire array is copied by value.
- $(\text{vector } \tau) \not\cong (\text{vector } (\text{mutable } \tau))$
- $(\text{ref } \tau) \not\cong (\text{ref } (\text{mutable } \tau))$.
- Compatibility of composites: Composite types are copy compatible if and only if all of their fields have equal types in the case of boxed types and copy compatible types in the case of unboxed types. In order to enforce this rule, the following restriction must be imposed for unboxed parametric types: instantiations of any type variable used within another reference type must match exactly. For example in the following structure:

```
(defstruct (St 'a 'b):val f1:'a f2:(ref 'b))
```

instantiations of `'a` are only required to be copy compatible, but instantiations of `'b` must match exactly.

```
(St char char)  $\cong$  (St (mutable char) char)
(St char char)  $\not\cong$  (St char (mutable char))
```

In addition to argument passing, and new variable bindings, we can also permit copy compatibility at argument and return position of all expressions that do not return an lvalue. This is because we can think of them in terms of equivalent SSA forms, which introduce temporary bindings for all intermediate results. For example, we can permit the various branches of conditional expressions to have to have different but copy compatible types:

```
(if #t m:(mutable int32) 10:int32)
```

3.4 Type Inference

When an exact type compatibility requirement is replaced in the language design by copy compatibility, it is no longer possible to infer a unique type for the expression. For example, in the following expression:

```
(let ((p 10:int32)) ... )
```

we know that the type of the literal 10 is `int32`, but the type of `p` could either be `int32`, or `(mutable int32)`. When we cannot ascertain the mutability status of a bound identifier, we give it the so-called “maybe” type `(?mutable? int32)`, which is actually a shorthand for the constrained type $\alpha \setminus \{\alpha \cong \text{int32}\}$. That is, it is undecided as to whether the actual type is `int32`, or `(mutable int32)`, and can be resolved by later unification. If it is not, a choice must eventually be fixed by the language definition. For example, the `let` form:

```
(let ((p (pair 1:int32 #t))) ... )
```

introduces copy compatibility at both the `pair` constructor application, and the formation of a new binding for `p`. The types assigned by the compiler are:

```
p:(?mutable? (pair (?mutable? int32)
                  (?mutable? bool)))
```

3.4.1 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 an expression language with copy compatibility, inferring immutable types by default 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 that returns the first element of a pair and `m` is a variable with type `(mutable bool)`, we will have to write:

```
(define xyz
  (vector (fst (pair m 10)
            :(pair (mutable bool) int32)))
         :(vector (mutable int32)))
```

Pierce and Turner have conducted a study on the impact of requiring explicit type annotations in higher order typed programming languages [31]. Their measurements on about 1,60,000 lines of Objective Caml [23] code revealed that polymorphic type instantiations happen every third line of code, anonymous function definitions happen once in 10-100 lines of code and local bindings occur about once every twelve lines. Therefore, in BitC, not inferring mutability would make type inference a liability rather than an asset in the case of stateful programs.

3.4.2 Incompleteness of Inference

The key idea of maybe types is to defer commitments about the mutability status of types, and thus infer most-general types wherever possible. BitC is a let-polymorphic language and enforces the value restriction [50]. This means that the decision about the mutability of types cannot be deferred past their `let` bindings, since mutable types must not be generalized. For example, in the case of the expression:

```
(let ((p nil)) ... )
```

we cannot give `p` the type

```
(forall ('a) (?mutable? (list 'a)))
```

We must instead choose one of:

```
(forall ('a) (list 'a)) ; polymorphic
(?mutable? (list 'a))  ; monomorphic
```

That is, there is no principal type that we can infer for `p`. Given this, we must fix these maybe types to either mutable or immutable at a let-boundary. In the next section, we will identify various choices for how to fix these maybe types, and discuss their merits and limitations.

In contrast, ML is able to infer principal types since its inference rules are purely syntax directed. In BitC, we trade completeness of inference to obtain a more expressive language without making any major changes to the core type system.

3.4.3 Inference Considerations

In this section we outline certain design considerations for a type inference scheme in the presence of copy compatibility. An ideal scheme must not require excessive programmer annotations in the common case, and must be capable of inferring all sound types at least when guided by explicit annotation.

The problem with programmer annotations is pragmatic rather than ideological: we do not view programmer specification of types as bad *per se* (indeed, in certain places BitC requires annotations), but ease of prototyping requires that these annotations be minimized. As a matter of good programming ideology and interfacing with other static analysis or verification tools, the inferred types must not be promiscuous with respect to mutability.

First, we consider how to solve the copy compatibility constraints introduced by the maybe types. One possibility is to fix all unresolved maybe types to immutable versions. For example:

```
(let ((p (pair n:(mutable int32)
              (lambda (x) x)))) ...)
p: (pair int32 (fn ('a) 'a))
```

This scheme will preserve all polymorphism possible, but will mandate a programmer annotation for every mutable location. The alternative would be to choose the mutable variants, in which case we will effectively have no polymorphism (by default). In the case of local definitions, we can collect more usage information and fix maybe-ness accordingly.

The previous section argued that we “lose” precision of inferred types (with respect to mutability) by the introduction of copy compatibility. We can think of this as a trade-off between freedom in type compatibility and precision of inference. Therefore, we can choose whether to (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 we permit copy compatibility to the maximum permissible limit (as defined in section 3.3), 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 we can make function types 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:

```
(import ls bitc.list)
(define (list2vec lst)
  (make-vector (length lst)
    (lambda (n) (ls.list-nth lst n))))
```

For a naïve reader, the type of `list2vec` appears to be `(fn ((list 'a)) (vector 'a))`, but is actually the more general type:

```
(forall ((copy-compat 'a 'b))
        (fn ((list 'a)) (vector 'b)))
```

`copy-compat` is a special type class that relates two copy compatible types. Now, if we default maybe types that are ultimately unresolved to immutable, in the following definition we obtain:

```
(define mVec (list2vec mLst:(list (mutable bool)))
mVec: (vector bool) ;; !!!
```

which is a correct typing, but is most likely not what the programmer expects. In this case, even though the both the argument and return types of `list2vec` are reference types, they are only required to be copy compatible because `list2vec` copies the constituent elements, thereby using new locations.

3.5 Type Inference in BitC

Having identified the various issues and trade-offs involved in type inference, we now describe the particular design choices made in BitC for handling copy compatibility. This is by no means “the” solution to the problem, but reflects our judgment of the best way to capture the programmer’s intuition about the flow of types in the language. It has been driven in part by our experience writing BitC programs. In BitC, 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.

Every time we form a “maybe” type due to a copy operation, we remember the original type as a hint to resolve the copy compatibility constraints in the resultant type. At a let boundary, we resolve any unresolved compatibility constraints by unifying with this hint. Intuitively, this means that we will default maybe types to the types of their original copies, unless overridden by an explicit annotation. Here, we are approximating the user’s intent to the lexical “flow” of type information. For example:

```
(define mb:(mutable bool) #t))
mb: (mutable bool)

(define p (vector mb))
p: (vector (mutable bool))

(define q:(vector bool) (vector mb))
q: (vector bool)
```

The type of `p` shows how maybe types are defaulted based on hint information, and the type of `q` shows how this can be overridden by programmer annotation. Since we default unresolved maybe-types to original ones the `list2vec` example described in section 3.4.3 now gets the more intuitive type:

```
list2vec: (fn ((list 'a)) (vector 'a))
```

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 a `set!`, it is given a shallowly mutable type. This is an *ad hoc* rule that tries to reduce the need for explicit type qualifications by the programmer in the common case (ex: iterators). However, this rule must not be invoked for top-level (global) definitions. Otherwise, inferred types will no longer be deterministic, as the top level definitions have unlimited scope.

```
(define (fact x) (do ((ans 1 ans)
                    (i x (- i 1)))
                    ((= i 0) ans)
                    (set! ans (* ans i))))
```

In the case of conflicting hints in the different branches of conditional expressions, we pick the most immutable of all hints. This ensures that inferred types are always deterministic. For example:

```
(define boolPair
  (if #t
      (pair #t #f):((mutable bool), bool)
      (pair #f #t):(bool, (mutable bool))))
p: (bool, bool)
```

If there are any residual compatibility constraints even after unifying with hints, we resolve them to immutable variants. 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 (the external type), and must hide the “internal” mutability information.

```
(define (f x) (set! x x))
Internal Type f: (fn ((mutable 'a)) ())
External Type f: (fn ('a) ())
```

Even though function types must be written in external form, any type-qualifications on the arguments of a function within its body correspond to the internal types, and may contain mutable qualifications.

```
(define abc:(fn ((mutable bool)) 'a) ...) ;; ERROR
(define (abc x:(mutable bool)) ... )      ;; OK
```

This internal/external type notion is also important in the process of resolving copy compatibility constraints using hints. We should be sure 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. For example:

```
(define p:(mutable bool) #t)
(define (f x) p) ;; f: (fn ('a) bool)
(define (g x) x) ;; g: (fn ('a) 'a)

(define ff (f p)) ;; ff: bool
(define gg (g p)) ;; gg: (mutable bool)
```

Further, two instances of a type class can co-exist only if all methods have different external signatures. For example:

```
(deftypeclass (TCL 'a 'b)
  mtd: (fn ('a (vector 'b)) 'a))

(definstance (TCL bool bool) ...)
(definstance (TCL (mutable bool) bool) ..) ;CONFLICT!
(definstance (TCL bool (mutable bool)) ..) ;OK.
```


Chapter 4

Formalization

4.1 The Language

In the interest of brevity, we will limit ourselves to the following core calculus:

Syntax

Identifiers	$x ::= y \mid z \mid \dots$
Stack Locations	$l ::= l_1 \mid l_2 \mid \dots$
Heap Locations	$\ell ::= \ell_1 \mid \ell_2 \mid \dots$
Values	$v ::= () \mid true \mid false \mid \ell \mid \lambda x. e$
lvalues	$\mathcal{L} ::= l \mid \ell^\wedge$
Expressions	$e ::= v \mid e e \mid e : \tau \mid e := e \mid \text{let}^\kappa x[:\tau] = e \text{ in } e$ $\quad \mid \text{dup}(e) \mid e^\wedge \mid \text{if } e \text{ then } e \text{ else } e$
Let-kinds	$\kappa ::= - \mid \alpha \mid \psi \mid \forall$

All syntactic forms introduced in this document can be parenthesized without change in meaning. An optional qualification is provided on the identifier defined in a `let` expression since the same effect cannot be obtained by qualifying the defining expression (due to copy compatibility). The let-kind “-” is a placeholder for the unkinded `let` form.

4.2 Dynamic Semantics

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, the heap, and the expression 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.

Table 4.1 shows the evaluation rules for our core language. Following the theoretical development in [53, 54], 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_l and \Rightarrow_r respectively.

In the above operational semantics rules, a distinction is made between the stack and the heap in order to ensure that we can only capture references to heap cells (E-DUP, E-LEFT-DEREF, and E-DEREF work only on the heap). The heap locations are first class values but stack locations are not. Therefore stack locations cannot escape beyond their scope (although the bindings themselves are not removed from the stack, in the interest of of simplicity). E-RVAL represents implicit value extraction for stack locations. State updates can be performed either on the stack or on the heap (E-SET-STACK and E-SET-HEAP). We assume that the program is alpha-converted so that there are no name collisions due to inner bindings. We do not model garbage collection, and assume an infinite supply of stack and heap

	E-RVAL	E-LEFT-TQ
	$\frac{S(l) = v}{S; H; l \Rightarrow S; H; v}$	$\frac{}{S; H; e : \tau \Rightarrow S; H; e}$
E-TQ	E-LET-TQ	
$\frac{}{S; H; e : \tau \Rightarrow S; H; e}$	$\frac{}{S; H; \text{let } x : \tau = e_1 \text{ in } e_2 \Rightarrow S; H; \text{let } x = e_1 \text{ in } e_2}$	
E-APP-STEP-FN	E-APP-STEP-ARG	E-APP
$\frac{S; H; e_1 \Rightarrow S'; H'; e'_1}{S; H; e_1 e_2 \Rightarrow S'; H'; e'_1 e_2}$	$\frac{S; H; e_2 \Rightarrow S'; H'; e'_2}{S; H; v_1 e_2 \Rightarrow S'; H'; v_1 e'_2}$	$\frac{l \notin \text{dom}(S)}{S; H; \lambda x. e v \Rightarrow S, l \mapsto v; H; e[l/x]}$
E-LET-STEP		
	$\frac{S; H; e_1 \Rightarrow S'; H'; e'_1}{S; H; \text{let } x = e_1 \text{ in } e_2 \Rightarrow S'; H'; \text{let } x = e'_1 \text{ in } e_2}$	
E-LET-M	E-LET-P	
$\frac{l \notin \text{dom}(S)}{S; H; \text{let}^\psi x = v_1 \text{ in } e_2 \Rightarrow S, l \mapsto v_1; H; e_2[l/x]}$	$\frac{}{S; H; \text{let}^\forall x = v_1 \text{ in } e_2 \Rightarrow S; H; e_2[v_1/x]}$	
E-IF-STEP		
	$\frac{S; H; e \Rightarrow S'; H'; e'}{S; H; \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow S'; H'; \text{if } e'_1 \text{ then } e_2 \text{ else } e_3}$	
E-IF-TRUE	E-IF-FALSE	
$\frac{}{S; H; \text{if } \text{true} \text{ then } e_1 \text{ else } e_2 \Rightarrow S; H; e_1}$	$\frac{}{S; H; \text{if } \text{false} \text{ then } e_1 \text{ else } e_2 \Rightarrow S; H; e_2}$	
E-DUP-STEP	E-DUP	
$\frac{S; H; e \Rightarrow S'; H'; e'}{S; H; \text{dup}(e) \Rightarrow S'; H'; \text{dup}(e')}$	$\frac{l \notin \text{dom}(H)}{S; H; \text{dup}(v) \Rightarrow S; H, l \mapsto v; \ell}$	
E-LEFT-DEREF-STEP	E-DEREF-STEP	E-DEREF
$\frac{S; H; e \Rightarrow S'; H'; e'}{S; H; e^\wedge \Rightarrow S'; H'; e'^\wedge}$	$\frac{S; H; e \Rightarrow S'; H'; e'}{S; H; e^\wedge \Rightarrow S'; H'; e'^\wedge}$	$\frac{H(\ell) = v}{S; H; \ell^\wedge \Rightarrow S; H; v}$
E-SET-STEP-LHS	E-SET-STEP-RHS	
$\frac{S; H; e_1 \Rightarrow S'; H'; e'_1}{S; H; e_1 := e_2 \Rightarrow S'; H'; e'_1 := e_2}$	$\frac{S; H; e_2 \Rightarrow S'; H'; e'_2}{S; H; \mathcal{L} := e_2 \Rightarrow S'; H'; \mathcal{L} := e'_2}$	
E-SET-STACK	E-SET-HEAP	
$\frac{}{S, l \mapsto v_1; H; l := v_2 \Rightarrow S, l \mapsto v_2; H; ()}$		$\frac{}{S; H, \ell \mapsto v_1; \ell^\wedge := v_2 \Rightarrow S; H, \ell \mapsto v_2; ()}$

Table 4.1: Dynamic Semantics

cells.

We cannot give *one* correct execution semantics for the (unkinded) expression: $\text{let } x = v \text{ in } e$, since there is not enough syntactic support to determine whether x is a (mutable) location or a polymorphic immutable term. The correct step to take is always clear from static type information. We must take the LET-M path for all mutable definitions, and the LET-P path for all polymorphic definitions. For immutable non-polymorphic definitions, either step will work, but we always choose LET-M.

Therefore, we make a distinction among two “kinds” of let definitions as: let^ψ (monomorphic, possibly mutable definition) and let^\forall (polymorphic definition). The non-polymorphic version of let is shown as let^ψ and not let because an immutable monomorphic/concrete definition is also defined using this construct. Now, we give separate execution semantics for each of these let forms. Since the type rules only derive a type for the correct kinds of let in each case, execution of typed expressions is always expected to take the correct path.

We use let to range over let^ψ and let^\forall . and will still use the unkinded term let , when either version is (equally) applicable. Note that let is different from let because two occurrences of let in the same context correspond to the same kind, whereas two occurrences of let need not match the same kind.

4.3 Static Semantics

Syntax

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	$\sigma ::= \tau \mid \forall\alpha.\sigma$
Binding Environment	$\Gamma ::= \emptyset \mid \Gamma, x \mapsto \sigma$
Store Typing	$\Sigma ::= \emptyset \mid \Sigma, \ell \mapsto \tau \mid \Sigma, l \mapsto \tau$
Logical Relations	$\Omega ::= \text{true} \mid \text{false} \mid \Omega \wedge \Omega \mid \Omega \vee \Omega \mid \neg\Omega \mid \text{Predicate}(\overline{\Omega})$

A substitution is of Z for Y in X is written using the standard notation: $X[Z/Y]$. Substitutions within a type up to a ref boundary are written as: $\tau[Z/Y]$.

4.3.1 Copy Compatibility

Compatibility of types that differ in shallow mutability at copy boundaries is called copy compatibility [56], denoted by \cong , and is defined as:

$$\begin{aligned} \tau &\cong \tau \\ \Psi\tau &\cong \tau \end{aligned}$$

In our algebra of types, we have the equation $\Psi\Psi\tau \equiv \Psi\tau$.

We also define the operators Δ and ∇ that increase or decrease the mutability of a type, 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 $\Delta(\tau) = \Delta(\tau')$.

4.3.2 Compatibility of Function Types

Two function types are *equal* regardless of the shallow mutability of the argument and return types [56]. Due to copy compatibility, two function types are equivalent in all respects regardless of the (shallow) mutability of the argument and return positions. Therefore we always write all function types in the following normalized form:¹

1. The (contravariant) argument type is written in the maximally immutable form (devoid of shallow mutability).
2. The (covariant) return type is written in the maximally mutable form.

This ensures that the “external” type of a function is maximally permissive with respect to mutability. However, during type inference, if we infer the type $\nabla(\tau_{arg}) \rightarrow \Delta(\tau_{ret})$ as the external type of a function, this normalization can later get violated due to substitution of type-variables. Therefore, we define the $\lfloor\tau\rfloor$ and $\lceil\tau\rceil$ “meta-constructors” which (respectively) minimize and maximize the mutability of a type, but are interpreted lazily.

We also define (and implicitly use) the following equivalences in the algebra of types:

$$\begin{aligned} \lfloor\tau\rfloor &\equiv \nabla(\tau) \text{ (lazily)} \\ \lceil\tau\rceil &\equiv \Delta(\tau) \text{ (lazily)} \end{aligned}$$

TS-MUT1	TS-MUT2	TS-REF	TS-FN	TS-CEIL	TS-FLOOR
$\Psi\tau \preceq: \tau$	$\tau \preceq: \tau'$ $\Psi\tau \preceq: \Psi\tau'$	$\tau = \tau'$ $\uparrow\tau \preceq: \uparrow\tau'$	$\tau_1 = \tau'_1 \quad \tau_2 = \tau'_2$ $\tau_1 \rightarrow \tau_2 \preceq: \tau'_1 \rightarrow \tau'_2$	$\lceil \tau \rceil \preceq: \tau$	$\tau \preceq: \lfloor \tau \rfloor$
		TS-REFL $\tau \preceq: \tau$	TS-TRANS $\tau_2 \preceq: \tau_1 \quad \tau_1 \preceq: \tau$ $\tau_2 \preceq: \tau$		

Table 4.2: Copy Coercion Rules

4.3.3 Copy Coercions

Table 4.2 shows how we can obtain copy compatibility by copy coercions (similar to subtyping). We can now define copy compatibility as:

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

The TS-REF rule ensures that copy compatibility does not extend beyond a ref-boundary. Since two function types are equivalent in all respects regardless of the (shallow) mutability of the argument and return positions, we will 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 TS-FN rule therefore is invariant in terms of its arguments and return types.

We will also write $\Gamma; \Sigma \vdash e \preceq: \tau$ as a shorthand for: $\Gamma; \Sigma \vdash e : \tau', \tau' \preceq: \tau$.

Problem with Subsumption-like Coercion Rule

In BitC, explicit qualifications are a statement of absolute typing and not type compatibility. Preserving this rule, requires that we must not have generic subsumption like rules for copy coercion. That is, if we have the following subsumption like rules:

TW-TQEXPR (WRONG)	TW-SUB
$\frac{\Gamma; \Sigma \vdash e : \tau}{\Gamma; \Sigma \vdash (e : \tau) : \tau}$	$\frac{\Gamma; \Sigma \vdash e : \tau' \quad \tau' \preceq: \tau}{\Gamma; \Sigma \vdash e : \tau}$

We can then write:

$$\text{let } x = () \text{ in if } \text{true} \text{ then } x : \text{unit} \text{ else } x := ()$$

Here, in the `else` branch of `if`, we can derive the typing $x : \Psi\text{unit}$ and in the `then`, through subsumption also obtain $x : \text{unit}$, which violates our absolute-compatibility at qualification rule.

Therefore, we will not introduce this rule, but instead introduce copy coercion operations at all copy compatible positions in the type rules.

4.3.4 Location Semantics

The lvalue rules shown in Table 4.3 ensure that only those expressions permitted by location semantics [55, 56] appear on the left hand side of an assignment expression.

4.3.5 Declarative Type Rules

Declarative Type rules are given in Table 4.4. 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 write $e \preceq: \tau$ as a shorthand for: $e : \tau'$,

¹ Of course, types are not displayed in this form to the user. Function types are always printed in “interface form,” which does not expose the “internal” mutability of argument or return types [56].

L-ID	L-HLOC	L-SLOC	L-DEREF	L-TQ
$\frac{}{\vdash_{\text{val}} x}$	$\frac{}{\vdash_{\text{val}} \ell}$	$\frac{}{\vdash_{\text{val}} 1}$	$\frac{}{\vdash_{\text{val}} e^\wedge}$	$\frac{\vdash_{\text{val}} e}{\vdash_{\text{val}} e : \tau}$

Table 4.3: Location Semantics Rules

T-UNIT	T-TRUE	T-FALSE	T-ID
$\frac{}{\Gamma; \Sigma \vdash () : \text{unit}}$	$\frac{}{\Gamma; \Sigma \vdash \text{true} : \text{bool}}$	$\frac{}{\Gamma; \Sigma \vdash \text{false} : \text{bool}}$	$\frac{\Gamma(x) = \forall \bar{\alpha}. \tau}{\Gamma; \Sigma \vdash x : \tau[\bar{\tau}'/\bar{\alpha}]}$
T-HLOC	T-SLOC	T-LAMBDA	
$\frac{\Sigma(\ell) = \tau}{\Gamma; \Sigma \vdash \ell : \uparrow\tau}$	$\frac{\Sigma(l) = \tau}{\Gamma; \Sigma \vdash l : \tau}$	$\frac{\Gamma, x \mapsto \tau_1; \Sigma \vdash e : \tau_2}{\Gamma; \Sigma \vdash \lambda x. e : \nabla(\tau_1) \rightarrow \Delta(\tau_2)}$	
T-APP	T-SET		
$\frac{\Gamma; \Sigma \vdash e_1 \preccurlyeq : \tau_2 \rightarrow \tau_0 \quad \Gamma; \Sigma \vdash e_2 \preccurlyeq : \tau_2 \quad \tau_0 \preccurlyeq : \tau'_0}{\Gamma; \Sigma \vdash e_1 e_2 : \tau'_0}$	$\frac{\Gamma; \Sigma \vdash e_1 \preccurlyeq : \Psi\tau \quad \Gamma; \Sigma \vdash e_2 \preccurlyeq : \tau \quad \vdash_{\text{val}} e_1}{\Gamma; \Sigma \vdash e_1 := e_2 : \text{unit}}$		
T-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-TQEXPR	T-DUP	T-DEREF	
$\frac{\Gamma; \Sigma \vdash e : \tau}{\Gamma; \Sigma \vdash (e : \tau) : \tau}$	$\frac{\Gamma; \Sigma \vdash e \preccurlyeq : \tau \quad \tau' \preccurlyeq : \tau}{\Gamma; \Sigma \vdash \text{dup}(e) : \uparrow\tau'}$	$\frac{\Gamma; \Sigma \vdash e \preccurlyeq : \uparrow\tau}{\Gamma; \Sigma \vdash e^\wedge : \tau}$	
T-LET-M [TQ]	T-LET-P [TQ]		Q-LOC
$\frac{\Gamma; \Sigma \vdash e_1 \preccurlyeq : \tau_1 \quad \tau \preccurlyeq : \tau_1 \quad \Gamma; \Sigma; e_1 \vdash_{\text{gen}} \tau \triangleleft \sigma \quad \vdash_{\text{loc}} x : \sigma \quad \Gamma, x \mapsto \sigma; \Sigma \vdash e_2 : \tau_2}{\Gamma; \Sigma \vdash (\text{let}^\psi x[\tau] = e_1 \text{ in } e_2) : \tau_2}$	$\frac{\Gamma; \Sigma \vdash e_1 \preccurlyeq : \tau_1 \quad \tau \preccurlyeq : \tau_1 \quad \Gamma; \Sigma; e_1 \vdash_{\text{gen}} \tau \triangleleft \sigma \quad \vdash_{\text{term}} x : \sigma \quad \Gamma, x \mapsto \sigma; \Sigma \vdash e_2 : \tau_2}{\Gamma; \Sigma \vdash (\text{let}^\forall x[\tau] = e_1 \text{ in } e_2) : \tau_2}$		$\frac{\sigma = \tau}{\vdash_{\text{loc}} x : \sigma}$
			Q-TERM
			$\frac{\sigma = \forall \bar{\alpha}. \tau}{\vdash_{\text{term}} x : \sigma}$

Table 4.4: Declarative Type Rules

$\tau' \preccurlyeq : \tau$, for some type τ' . In the type rules, we introduce copy coercions at all positions where copy compatibility is applicable. Type generalization at a `let` is decided by the judgment \vdash_{gen} .

4.3.6 Generalization

This section uses the “full” value restriction as proposed by Wright [50]. Relaxations to this rule based on Garriague’s scheme [52] were proposed in [56]. Those rules must be incorporated at a later stage.

The type generalization rules are shown in Table 4.5. The rule G-VALUE-PF is not actually used for typing user programs, but is only necessary to prove preservation of types.

DEFINITION: Free Type Variables

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

$$\begin{aligned} \text{ftv}(\alpha) &= \alpha \\ \text{ftv}(\text{unit}) &= \{\} \end{aligned}$$

G-VALUE	G-EXPANSIVE
$\frac{\text{Value}(e) \quad \text{Immut}(\tau) \quad \bar{\alpha} = \text{ftv}(\tau) \setminus \text{ftv}(\Gamma) \setminus \text{ftv}(\Sigma)}{\Gamma; \Sigma; e \vdash_{\text{gen}} \tau \triangleleft \forall \bar{\alpha}. \tau}$	$\frac{}{\Gamma; \Sigma; e \vdash_{\text{gen}} \tau \triangleleft \tau}$

Table 4.5: Generalization Rules

$$\begin{aligned} \text{ftv}(\text{bool}) &= \{\} \\ \text{ftv}(\uparrow\tau) &= \text{ftv}(\tau) \\ \text{ftv}(\Psi\tau) &= \text{ftv}(\tau) \\ \text{ftv}(\tau_1 \rightarrow \tau_2) &= \text{ftv}(\tau_1) \cup \text{ftv}(\tau_2) \\ \text{We also write:} \\ \text{ftv}(\overline{\tau}) &= \bigcup \text{ftv}(\tau) \\ \text{ftv}(\sigma) &= \text{ftv}(\overline{\alpha}) \cup \text{ftv}(\tau) \text{ where } \sigma = \forall \overline{\alpha}. \tau \\ \text{ftv}(\Gamma) &= \bigcup \text{ftv}(\sigma) \text{ for all } x : \sigma \in \Gamma \\ \text{ftv}(\Sigma) &= \bigcup \text{ftv}(\tau) \text{ for all } \ell : \tau \in \Sigma \text{ or } l : \tau \in \Sigma \\ \text{ftv}(\mathcal{C}) &= \bigcup \text{ftv}(\tau) \text{ where } \mathcal{C} \text{ is a set of constraints, and } \tau \text{ is a type involved in any constituent constraint.} \\ \text{FTVS}(X, Y, \dots) &= \text{FTVS}(X) \cup \text{FTVS}(Y) \cup \dots, \text{ where } X, Y, \dots \text{ are any of the above permissible arguments.} \end{aligned}$$
DEFINITION: Value Restriction

$$\begin{aligned} \text{Value}(v) &= \text{true} \\ \text{Value}(x) &= \text{true} \\ \text{Value}(\ell) &= \text{true} \\ \text{Value}(l) &= \text{true} \\ \text{Value}(e : \tau) &= \text{Value}(e) \\ \text{Value}(\text{dup}(e)) &= \text{Value}(e) \\ \text{Value}(e \wedge) &= \text{Value}(e) \\ \text{Value}(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) &= \text{Value}(e_1) \wedge \text{Value}(e_2) \wedge \text{Value}(e_3) \\ \text{Value}(\text{let } x = e_1 \text{ in } e_2) &= \text{Value}(e_1) \wedge \text{Value}(e_2) \\ \text{Value}(e) &= \text{false} \\ \text{Immut}(\text{unit}) &= \text{true} \\ \text{Immut}(\text{bool}) &= \text{true} \\ \text{Immut}(\alpha) &= \text{true} \\ \text{Immut}(\tau_1 \rightarrow \tau_2) &= \text{true} \\ \text{Immut}(\uparrow\tau) &= \text{Immut}(\tau) \\ \text{Immut}(\forall \overline{\alpha}. \tau) &= \text{Immut}(\tau) \\ \text{Immut}(\tau) &= \text{false} \end{aligned}$$
4.3.7 Soundness of Declarative system**DEFINITION: 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: 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: 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: 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: 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: $e = e_1 e_2$, $e_1 \preccurlyeq: \tau_1 \rightarrow \tau_2$, and $e_2 \preccurlyeq: \tau_1$. If e_1 or e_2 is not a value, we can take E-APP-STEP-FN or E-APP-STEP-ARG. 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: $e = e_1 := e_2$, $e_1 \preccurlyeq: \tau_1 \rightarrow \tau_2$, $\Gamma; \Sigma \vdash e_1 \preccurlyeq: \Psi\tau$, $\Gamma; \Sigma \vdash e_2 \preccurlyeq: \tau$, and $\vdash_{lval} e_1$. If e_1 not an lvalue, since we have $\vdash_{lval} e_1$ we can take E-SET-STEP-LHS by induction hypothesis. Similarly, if e_2 is not a value, we can take E-SET-STEP-RHS. Finally, if $e_1 = \mathcal{L}$ and $e_2 = v$, we can take the step E-SET-STACK or E-SET-HEAP as applicable.
7. Case T-DUP: Only right execution is permitted, and can take E-DUP-STEP or E-SUP-STEP as applicable.
8. Case T-DEREF: We have: $e = e_1 \hat{\ }^$, and $\Gamma; \Sigma \vdash e_1 \preccurlyeq: \uparrow\tau$. Execution can take E-LEFT-DEREF-STEP or E-DEREF-STEP as applicable if e_1 is not a value. If $e_1 \preccurlyeq: \uparrow\tau$ is a value, then, from the canonical forms lemma, $e_1 = \ell$, $\ell \in dom(\Sigma)$. 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-DEREF since the heap is well typed ($\Gamma; \Sigma \vdash H + S$).
9. Case T-LET-M: Only right execution is applicable. We have: $e = (\text{let}^\psi x = e_1 \text{ in } e_2)$, $\tau \preccurlyeq: \tau_1$, $\Gamma; \Sigma; e_1 \vdash_{gen} \tau \prec \sigma$, and $\vdash_{loc} x : \sigma$, and $\Gamma, x \mapsto \sigma; \Sigma \vdash e_2 : \tau_2$. If e_1 is not a value, we can take E-LET-STEP. Otherwise, we can take E-STEP-M.
10. Case T-LET-P: Only right execution is applicable. We have: $e = (\text{let}^\psi x = e_1 \text{ in } e_2)$, $\tau \preccurlyeq: \tau_1$, $\Gamma; \Sigma; e_1 \vdash_{gen} \tau \prec \sigma$, and $\vdash_{term} x : \sigma$, and $\Gamma, x \mapsto \sigma; \Sigma \vdash e_2 : \tau_2$. If e_1 is not a value, we can take E-LET-STEP. Otherwise, can take E-LET-P.
11. Case T-TQEXPR and Case T-LET-M-TQ, T-LET-P-TQ are similar.

LEMMA: Weakening

We will write $\Gamma; \Sigma \vdash e : \tau \prec \sigma$ as a shorthand for $\Gamma; \Sigma \vdash e : \tau$, and $\Gamma; \Sigma; e \vdash_{gen} \tau \prec \sigma$.

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

Proof: Straightforward induction on the typing derivation.

LEMMA: Value Substitution

If $\Gamma, x : \sigma; \Sigma \vdash e : \tau$, $\text{Immut}(\sigma)$ and $\Gamma; \Sigma \vdash v : \tau_v$, and $\Gamma; \Sigma; e \vdash_{gen} \tau_v \triangleleft \sigma$ 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 $\Gamma; \Sigma \vdash v : \tau_v \triangleleft \sigma$. 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: $\Gamma, y : \tau_1; \Sigma \vdash v : \tau_v \triangleleft \sigma$, 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: 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 4.3.7.7.

LEMMA: 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: 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 $\Gamma; \Sigma \vdash e_1 \preccurlyeq: \tau_2 \rightarrow \tau_0$, and $\Gamma; \Sigma \vdash e_2 \preccurlyeq: \tau_2$, and $\tau_0 \preccurlyeq: \tau$, 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-APP-STEP-FN: We have: $S; H; e_1 \Rightarrow S'; H'; e'_1$ and $e' = e'_1 e_2$. By induction hypothesis, we have: $\Gamma; \Sigma' \vdash e'_1 \preccurlyeq: \tau_2 \rightarrow \tau_0$ for some $\Sigma' \supseteq \Sigma$. One of the assumptions of the T-APP rule states that $\Gamma; \Sigma \vdash e_2 \preccurlyeq: \tau_2$, and by weakening lemma, we have, $\Gamma; \Sigma' \vdash e_2 \preccurlyeq: \tau_2$. Finally, by the T-APP rule, we conclude that $(e'_1 e_2) : \tau$.
- (b) Case E-APP-STEP-ARG: 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 $\tau_0 \preccurlyeq: \tau'_0$ and $\tau_0 \preccurlyeq: \tau$. Therefore, it is clear that $\nabla(\tau'_0) = \nabla(\tau)$.
4. Case T-SET: $e = e_1 := e_2$, and $\Gamma; \Sigma \vdash e_1 \preccurlyeq: \Psi\tau$, and $\Gamma; \Sigma \vdash e_2 \preccurlyeq: \tau \upharpoonright_{\text{loc}} e_1$
- If the step taken is E-SET-STEP-LHS or E-SET-STEP-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-SET-STACK or E-SET-HEAP, the result follows from the stack and heap assignment lemma.
5. Case T-DEREF: We have: $e = e' \hat{\ }^{\ }$ and $\Gamma; \Sigma \vdash e' \preccurlyeq: \uparrow\tau$. If the step taken is E-LEFT-DEREF-STEP or E-LEFT-DEREF, the result follows from induction hypothesis and T-DEREF rule. If the step taken is E-DEREF (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 $\Gamma; \Sigma \vdash e_1 \preccurlyeq: \tau_1$ and $\tau \preccurlyeq: \tau_1$ and $\Gamma; \Sigma; e_1 \upharpoonright_{\text{gen}} \tau \preccurlyeq \sigma$ 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-STEP, $S; H; e_1 \Rightarrow S'; H'; e'_1$ 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: 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: 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 4.3.7.5 (Progress), we know that we can take at least one step forward. Further, from Lemma 4.3.7.10 (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.

TE-UNIT	TE-TRUE	TE-FALSE
$\frac{}{\Gamma; \Sigma \vdash_{eq} () : \text{unit} \mid \emptyset}$	$\frac{}{\Gamma; \Sigma \vdash_{eq} \text{true} : \text{bool} \mid \emptyset}$	$\frac{}{\Gamma; \Sigma \vdash_{eq} \text{false} : \text{bool} \mid \emptyset}$
TE-ID	TE-HLOC	TE-SLOC
$\frac{\Gamma(x) = \forall \bar{\alpha}. \tau \quad \vdash_{new} \bar{\beta}}{\Gamma; \Sigma \vdash_{eq} x : \tau[\bar{\beta}/\bar{\alpha}] \mid \emptyset}$	$\frac{\Sigma(\ell) = \tau}{\Gamma; \Sigma \vdash_{eq} \ell : \uparrow \tau \mid \emptyset}$	$\frac{\Sigma(l) = \tau}{\Gamma; \Sigma \vdash_{eq} l : \tau \mid \emptyset}$
TE-LAMBDA	TE-APP	
$\frac{\Gamma, x \mapsto \alpha; \Sigma \vdash_{eq} e : \tau \mid \mathcal{C}}{\Gamma; \Sigma \vdash_{eq} \lambda x. e : [\alpha] \rightarrow [\tau] \mid \mathcal{C}}$	$\frac{\Gamma; \Sigma \vdash_{eq} e_1 : \tau_1 \mid \mathcal{C}_1 \quad \Gamma; \Sigma \vdash_{eq} e_2 : \tau_2 \mid \mathcal{C}_2 \quad \vdash_{new} \alpha}{\Gamma; \Sigma \vdash_{eq} e_1 e_2 : \alpha \mid \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \preceq: [\beta] \rightarrow [\gamma], \tau_2 \preceq: [\beta], [\gamma] \preceq: \alpha\}}$	
TE-IF	$\frac{\Gamma; \Sigma \vdash_{eq} e_1 : \tau_1 \mid \mathcal{C}_1 \quad \Gamma; \Sigma \vdash_{eq} e_2 : \tau_2 \mid \mathcal{C}_2 \quad \Gamma; \Sigma \vdash_{eq} e_3 : \tau_3 \mid \mathcal{C}_3 \quad \vdash_{new} \alpha}{\Gamma; \Sigma \vdash_{eq} \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \alpha \mid \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\tau_1 \preceq: \text{bool}, \tau_2 \preceq: \beta, \tau_3 \preceq: \beta, \alpha \preceq: \beta\}}$	
TE-TQEXPR	TE-SET	
$\frac{\Gamma; \Sigma \vdash_{eq} e : \tau' \mid \mathcal{C}}{\Gamma; \Sigma \vdash_{eq} (e : \tau) : \tau \mid \mathcal{C} \cup \{\tau' = \tau\}}$	$\frac{\Gamma; \Sigma \vdash_{eq} e_1 : \tau_1 \mid \mathcal{C}_1 \quad \Gamma; \Sigma \vdash_{eq} e_2 : \tau_2 \mid \mathcal{C}_2 \quad \vdash_{\text{val}} e_1 \quad \vdash_{new} \alpha}{\Gamma; \Sigma \vdash_{eq} e_1 := e_2 : \text{unit} \mid \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 = \Psi \alpha, \tau_1 \preceq: [\tau_2]\}}$	
TE-DUP	TE-DEREF	
$\frac{\Gamma; \Sigma \vdash_{eq} e : \tau \mid \mathcal{C} \quad \vdash_{new} \alpha}{\Gamma; \Sigma \vdash_{eq} \text{dup}(e) : \uparrow \alpha \mid \mathcal{C} \cup \{\alpha \preceq: [\tau]\}}$	$\frac{\Gamma; \Sigma \vdash_{eq} e : \tau \mid \mathcal{C} \quad \vdash_{new} \alpha}{\Gamma; \Sigma \vdash_{eq} e^\wedge : \alpha \mid \mathcal{C} \cup \{\tau \preceq: \uparrow \alpha\}}$	
TE-LET-M [TQ]		
$\frac{\Gamma; \Sigma \vdash_{eq} e_1 : \tau_1 \mid \mathcal{C}_1 \quad \mathcal{C}'_1 = \mathcal{C}_1 \cup \{\tau \preceq: [\tau_1]\} \quad \theta \Vdash_{\text{unif}} \mathcal{C}'_1 \quad \Gamma; \Sigma; e_1 \vdash_{\text{gen}} \theta \langle \tau \rangle \prec \sigma \quad \vdash_{\text{loc}} x : \sigma \quad \theta \langle \Gamma \rangle, x \mapsto \sigma; \theta \langle \Sigma \rangle \vdash_{eq} e_2 : \tau_2 \mid \mathcal{C}_2}{\Gamma; \Sigma \vdash_{eq} \text{let}^\psi x = e_1 \text{ in } e_2 : \tau_2 \mid \mathcal{C}_2 \cup \{\forall [\alpha \mapsto \tau] \in \theta, \alpha = \tau\}}$		
TE-LET-P [TQ]		
$\frac{\Gamma; \Sigma \vdash_{eq} e_1 : \tau_1 \mid \mathcal{C}_1 \quad \mathcal{C}'_1 = \mathcal{C}_1 \cup \{\tau \preceq: [\tau_1]\} \quad \theta \Vdash_{\text{unif}} \mathcal{C}'_1 \quad \Gamma; \Sigma; e_1 \vdash_{\text{gen}} \theta \langle \tau \rangle \prec \sigma \quad \vdash_{\text{term}} x : \sigma \quad \theta \langle \Gamma \rangle, x \mapsto \sigma; \theta \langle \Sigma \rangle \vdash_{eq} e_2 : \tau_2 \mid \mathcal{C}_2}{\Gamma; \Sigma \vdash_{eq} \text{let}^\forall x : \tau = e_1 \text{ in } e_2 : \tau_2 \mid \mathcal{C}_2 \cup \{\forall [\alpha \mapsto \tau] \in \theta, \alpha = \tau\}}$		

Table 4.6: Equational Inference Rules

4.3.8 Equational Inference Algorithm

Syntax

Types	$\tau ::= \dots$
Constrained Type	$\tau \setminus \mathcal{C}$
Constraint Sets	$\mathcal{C} ::= \emptyset \mid \{(\tau = \tau \mid \tau \cong \tau \mid \tau \preceq: \tau)^*\}$
Substitutions	$\theta ::= \emptyset \mid [\alpha \mapsto \tau] \mid \theta \circ \theta$

We will denote applications of a list of substitutions of these substitutions as $\theta \langle X \rangle$, while an individual substitution is written using the standard notation $X[Z/Y]$.

The inference judgment $\Gamma; \Sigma \vdash_{eq} e : \tau \mid \mathcal{C}$ should be understood as: given the stack (binding) context Γ and heap (store) context Σ , we can infer the type τ for the expression e under the set of constraints \mathcal{C} . In the interest brevity, we will not track the freshness of type variables. We just write $\vdash_{new} \bar{\alpha}$ to mean that the sequence of type variables $\bar{\alpha}$ are new type variables.

Equational Inference rules are shown in Table 4.6. We will hereinafter write LET- α to range over LET-M and LET-P. We propagate solved constraints at a let-boundary (TE-LET- α and TE-LET- α -TQ rules) as equality constraints rather than substitutions. This is done just to present the rules in a simple form. A real implementation must propagate the substitutions so that closure computation does not increase exponentially. In this equational presentation of inference, types can be fixed by equality constraints, or compatibility requirements expressed as copy coercion constraints. The coercion constraints are used as hints and we obtain the substituted type by subsumption. The solution to these constraints closely corresponds to the rule defined in BitC. In this section, we do not present the fact that mutability of local variables are determined by examining the expression in which they are used. The actual implementation in BitC uses an eager unification algorithm, and is explained in the next section.

$$\begin{array}{c}
\mathbf{E-UNIFY} \\
\frac{\text{Close}(\mathcal{C}) = \underline{\mathcal{C}} \quad \frac{}{\vdash_{\text{consistent}} \underline{\mathcal{C}}} \quad \frac{}{\vdash_{\text{acyclic}} \underline{\mathcal{C}}}}{\vdash_{\text{UNF}} \underline{\mathcal{C}}} \quad \frac{\mathbf{E-UNIFY-SOLVE}}{\frac{}{\vdash_{\text{UNF}} \underline{\mathcal{C}}} \quad \theta \frac{}{\vdash_{\text{sol}} \underline{\mathcal{C}}}}{\theta \frac{}{\vdash_{\text{unf}} \underline{\mathcal{C}}}}}
\end{array}$$

Table 4.7: Equational Unification Rules

$$\begin{array}{c}
\mathbf{SOL-EQ} \\
\frac{\alpha = \tau \in \mathcal{C} \quad \theta = [\alpha \mapsto \tau] \quad \theta' \frac{}{\vdash_{\text{sol}} \theta \langle \mathcal{C} \setminus \{\alpha = \tau\} \rangle}}{\theta \circ \theta' \frac{}{\vdash_{\text{sol}} \mathcal{C}}} \\
\mathbf{SOL-SUB} \\
\frac{\alpha = \tau' \notin \mathcal{C} \quad \mathcal{C}_a = \{\forall \alpha \preccurlyeq: \tau_1 \in \mathcal{C}, \forall \tau_2 \preccurlyeq: \alpha \in \mathcal{C}\} \quad \tau = \text{some type such that} \quad \theta = [\alpha \mapsto \tau] \quad \theta \vdash \mathcal{C}_a \quad \theta' \frac{}{\vdash_{\text{sol}} \theta \langle \mathcal{C} \setminus \{\alpha \preccurlyeq: \tau\} \rangle}}{\theta \circ \theta' \frac{}{\vdash_{\text{sol}} \mathcal{C}}}
\end{array}$$

Table 4.8: Solving Equational Constraints

4.3.9 Unification

Unification Rules are shown in Table 4.7. We write $\theta \vdash \mathcal{C}$ to mean that the substitution θ *satisfies* the set of constraints \mathcal{C} . A constraint set $\underline{\mathcal{C}}$ is said to be the normalized form of \mathcal{C} if it is written as a set of atomic constraints by using the copy coercion rules defined in Table 4.2 (note that this conversion is total). Further this set is closed, such that all transitive relationships are explicitly added. (see Definition 4.3.9.1). The unification judgment $\frac{}{\vdash_{\text{UNF}} \underline{\mathcal{C}}}$ says that the constraint set \mathcal{C} when written equivalent canonical form as $\underline{\mathcal{C}}$ is consistent and acyclic. The statement $\theta \frac{}{\vdash_{\text{unf}} \underline{\mathcal{C}}}$ is understood as: The substitution θ obtained as a result of unification and constraint solving satisfies the set of constraints \mathcal{C} .

DEFINITION: Constraint Set Closure

1. For each $\tau_1 \rightarrow \tau_2 = \tau'_1 \rightarrow \tau'_2 \in \mathcal{C}$ add $\tau_1 = \tau'_1$ and $\tau_2 = \tau'_2$.
2. For each $\tau_1 \rightarrow \tau_2 \preccurlyeq: \tau'_1 \rightarrow \tau'_2 \in \mathcal{C}$ add $\tau_1 = \tau'_1$ and $\tau_2 = \tau'_2$.
3. For each $\uparrow\tau = \uparrow\tau' \in \mathcal{C}$ add $\tau = \tau'$.
4. For each $\uparrow\tau \preccurlyeq: \uparrow\tau' \in \mathcal{C}$ add $\tau = \tau'$.
5. For each $\Psi\tau = \Psi\tau' \in \mathcal{C}$ add $\tau = \tau'$.
6. For each $\Psi\tau \preccurlyeq: \Psi\tau' \in \mathcal{C}$ add $\tau \preccurlyeq: \tau'$.
7. For each $\tau_1 = \tau_2 \in \mathcal{C}$ and $\tau_2 = \tau_3 \in \mathcal{C}$ add $\tau_1 = \tau_3$.
8. For each $\tau_1 \preccurlyeq: \tau_2 \in \mathcal{C}$ and $\tau_2 \preccurlyeq: \tau_3 \in \mathcal{C}$ add $\tau_1 \preccurlyeq: \tau_3$.
9. For each $\tau_1 = \tau_2 \in \mathcal{C}$ and $\tau_2 \preccurlyeq: \tau_3 \in \mathcal{C}$ add $\tau_1 \preccurlyeq: \tau_3$.
10. For each $\tau_1 \preccurlyeq: \tau_2 \in \mathcal{C}$ and $\tau_2 = \tau_3 \in \mathcal{C}$ add $\tau_1 \preccurlyeq: \tau_3$.

Continue till no more constraints can be added.

Solving the Constraints

Judgments to solve equational constraints are shown in Table 4.8.

4.3.10 Soundness of Equational Inference

THEOREM: Soundness of Constraint Closure

If $\theta \models C$ then $\theta \models \text{Close}(C)$.

Proof: Immediate from the definition of closure computation and copy coercion rules.

THEOREM: Correctness of Constraint Solver

$\theta \models_{\text{sol}} C$ iff $\theta \models C$.

Proof: Evident from the definition of the constraint solver.

THEOREM: Correctness of Unification

Unification produces a valid substitution that satisfies all the constraints in a constraint set. That is, if $\theta \models_{\text{unf}} C$ then $\theta \models C$.

Proof: Evident from the definition of unification, and Theorem 4.3.10.2.

THEOREM: Decidability of Unification

$\theta \models_{\text{unf}} C$ is decidable.

Proof: Follows from following facts:

1. The constraint set C is finite. Closure computation follows set-union semantics and will eventually terminate after all possible constraints are added (as evident from the cope coercion relationships).
2. Consistency check is bounded by the cardinality of the constraint set and is thus decidable.
3. There are no forms that introduce cyclic types in this system. Even otherwise, cycle check is also bounded by the cardinality of the constraint set.
4. The constraint solver builds a solution tree by always invoking itself on *smaller* sets. Therefore, there can be no infinite derivations.

LEMMA: Substitution on Declarative Derivation

If $\Gamma; \Sigma \vdash e : \tau$ then $\theta(\Gamma); \theta(\Sigma) \vdash e : \theta(\tau)$.

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.

LEMMA: Weakening of Substitutions

1. If $\theta \models C_1 \cup C_2$ then $\theta \models C_1$ and $\theta \models C_2$.
2. If $\theta \models_{\text{sol}} C_1 \cup C_2$ then $\theta \models C_1$ and $\theta \models C_2$.

Proof: Part (1) is immediate. Part(2) follows from Theorem 4.3.10.2.

LEMMA: Composition of Solutions and Substitutions

If $\theta_1 \models_{\text{unf}} C_1$ and $C_2 \subseteq C_1$ then $\exists \theta_2$ such that $\theta_1 = \theta_2 \circ \theta_0$ and $\theta_2 \models_{\text{unf}} C_2$.

THEOREM: Soundness of Equational Inference

If $\Gamma; \Sigma \vdash_{eq} e : \tau \mid \mathcal{C}$ and $\theta \stackrel{\text{unif}}{=} \mathcal{C}$ then $\theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e : \theta\langle\tau\rangle$.

Proof: By induction on the equational inference derivation, proceeding by case analysis on the final step of derivation (we vacuously assume α -reduction):

1. Cases TE-UNIT, TE-TRUE, TE-FALSE, TE-ID, TE-HLOC, TE-SLOC are trivial.
2. Case TE-LAMBDA: From induction hypothesis, we have: $\theta\langle\Gamma, x \mapsto \alpha\rangle; \theta\langle\Sigma\rangle \vdash e' : \theta\langle\tau\rangle$. This can be re-written as: $\theta\langle\Gamma\rangle, x \mapsto \theta\langle\alpha\rangle; \theta\langle\Sigma\rangle \vdash e' : \theta\langle\tau\rangle$. Now, from the T-LAMBDA rule, we can conclude that $\theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash \lambda x. e' : \theta\langle\alpha\rangle \rightarrow \theta\langle\tau\rangle$.
3. Case TE-APP: By induction hypothesis, and Lemma 4.3.10.5 and Lemma 4.3.10.7 we have: $\theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e_1 \preccurlyeq: \theta\langle\tau_1\rangle\theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e_2 \preccurlyeq: \theta\langle\tau_2\rangle$. By inspecting the constraints that get added at the T-APP step we can conclude that the result of substitution must imply $\theta\langle\tau_1\rangle \preccurlyeq: \nabla(\theta\langle\tau_2\rangle) \rightarrow \Delta(\theta\langle\alpha\rangle)$. We also have the axiom $\Delta(\theta\langle\alpha\rangle) \preccurlyeq: \theta\langle\alpha\rangle$. Now, from T-APP rule, we can obtain $\theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e_1 e_2 : \theta\langle\alpha\rangle$.
4. Cases TE-IF, TE-SET, TE-DUP, TE-DEREF and TE-SET are similar.
5. Case TE-LET- α : By induction hypothesis, (similar to TE-APP case) we have: $\theta_u\langle\Gamma\rangle; \theta_u\langle\Sigma\rangle \vdash e_1 : \theta_u\langle\tau_1\rangle$ and $\theta_u \stackrel{\text{unif}}{=} \mathcal{C}'_1$. Let τ be a type such that $\tau \preccurlyeq: [\theta\langle\tau_1\rangle]$. This can be re-written as: $\theta\langle\tau_1\rangle \preccurlyeq: \tau'$ and $\tau \preccurlyeq: \tau'$ to match the requirements of T-LET- α rule. Also, since both T-LET- α and TE-LET- α use the same generalization rules, we can obtain: $\theta_u\langle\Gamma\rangle; \theta_u\langle\Sigma\rangle; e_1 \vdash_{gen} \tau \leq \theta\langle\sigma\rangle$. By Lemma 4.3.10.7 $\theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e_1 : \theta\langle\tau_1\rangle$ and by suitable variable renaming we can obtain $\theta_u\langle\Gamma\rangle; \theta_u\langle\Sigma\rangle; e_1 \vdash_{gen} \tau \leq \theta\langle\sigma\rangle$. By induction hypothesis, we also have: $\theta\langle\Gamma\rangle, x \mapsto \theta\langle\sigma\rangle; \theta\langle\Sigma\rangle \vdash e_2 : \theta\langle\tau_2\rangle$. Therefore, from T-LET- α rule, we can obtain $\theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash \text{let } x = e_1 \text{ in } e_2 : \theta\langle\tau_2\rangle$.
6. Cases TE-TQEXPR and TE-LET- α -TQ are similar.

4.3.11 Inference with Eager Unification**Syntax**

Types $\tau ::= \dots$
 Maybe Type $\mid \tau \downarrow \tau'$

While performing eager unification, we cannot always infer a mutable type and later apply subsumption to obtain a less mutable type in favor of polymorphism. This is because, we must differentiate between types whose mutability is fixed (equality constraint due to the TE-SET rule) or, we have inferred a mutable type that is subject to change (copy coercion constraint). In order to achieve this, we maintain compatibility constraints on types themselves. We write $\tau \downarrow \tau'$ as a shorthand for the constrained type $\tau \setminus \{\tau \cong \tau'\}$. We can think of the type $\tau \downarrow \tau'$ as a “maybe type” τ which must be copy compatible but not necessarily equal to the type τ' . The type τ' is used as a hint to default the mutability of τ , unless it gets automatically fixed as a result of unification. In our language we *only* have maybe-mutability constraints, but we will still continue to use the constrained type notation $\tau \setminus \mathcal{C}$ in some cases while expressing generic mathematical properties over constrained types.

The inference judgment $\Gamma; \Sigma; \Pi \vdash_i e : \tau \parallel \theta; \Pi'$ should be understood as: given the stack (binding) context Γ and heap (store) context Σ , we can infer the type τ for the expression e . θ is list of substitutions obtained as a result of unifications performed in the process of inference, which must be propagate to further derivations. We thread a tape of type variables Π through the derivations (as in [57]) so that fresh type variables can be introduced reliably. Π is a tape that supplies fresh type variables. Π' is the tape rolled past the current derivation. Also, we write $\Pi_{\bar{\alpha}}$ to denote the tape that is rolled past the type variables $\bar{\alpha}$. Hindley Milner style inference rules are shown in Table 4.9.

The TI-LAMBDA rule uses the meta-constructors defined above to infer a normalized type for functions. The TI-APP rule infers copy compatible types by introducing maybe types at three positions — the function type itself (by unifying $\theta_2\langle\tau_1\rangle$ and $\beta \downarrow ([\delta] \rightarrow [\bar{\alpha}])$), the argument type (by unifying τ_2 with $\gamma \downarrow [\theta\langle\delta\rangle]$) and the return type (through $\varepsilon \downarrow \theta'_2\langle\tau\rangle$). Computing the type τ shows how the the hint for the result of an application is calculated. We start with an immutable version of the return type $\nabla(\theta\langle\alpha\rangle)$. $\bar{\theta}$ is the set of type variables that appear in the return type of the function but not in

<p>TI-UNIT</p> $\frac{}{\Gamma; \Sigma; \Pi \vdash () : \text{unit} \parallel \emptyset; \Pi}$	<p>TI-HLOC</p> $\frac{\Sigma(\ell) = \tau}{\Gamma; \Sigma; \Pi \vdash \ell : \uparrow\tau \parallel \emptyset; \Pi}$	<p>TI-SLOC</p> $\frac{\Sigma(\mathbf{l}) = \tau}{\Gamma; \Sigma; \Pi \vdash \mathbf{l} : \tau \parallel \emptyset; \Pi}$
<p>TI-ID</p> $\frac{\Gamma(x) = \forall \bar{\alpha}. \tau}{\Gamma; \Sigma; \Pi \vdash x : \tau[\bar{\beta}/\bar{\alpha}] \parallel \emptyset; \Pi_{\bar{\beta}}}$	<p>TI-TRUE</p> $\frac{}{\Gamma; \Sigma; \Pi \vdash \text{true} : \text{bool} \parallel \emptyset; \Pi}$	<p>TI-FALSE</p> $\frac{}{\Gamma; \Sigma; \Pi \vdash \text{false} : \text{bool} \parallel \emptyset; \Pi}$
<p>TI-LAMBDA</p> $\frac{\Gamma, x \mapsto \alpha; \Sigma; \Pi_{\alpha} \vdash e : \tau \parallel \theta; \Pi'}{\Gamma; \Sigma; \Pi \vdash \lambda x. e : [\theta(\alpha)] \rightarrow [\tau] \parallel \theta; \Pi'}$	<p>TI-TQEXPR</p> $\frac{\Gamma; \Sigma; \Pi \vdash e : \tau' \parallel \theta'; \Pi' \quad \theta \vdash_{\text{unf}} \tau' = \theta' \langle \tau \rangle}{\Gamma; \Sigma; \Pi \vdash (e : \tau) : \theta \langle \tau' \rangle \parallel \theta \circ \theta'; \Pi'}$	
<p>TI-APP</p> $\frac{\Gamma; \Sigma; \Pi_{\alpha\beta\gamma\delta\epsilon} \vdash e_1 : \tau_1 \parallel \theta_1; \Pi' \quad \theta_1 \langle \Gamma \rangle; \theta_1 \langle \Sigma \rangle; \Pi' \vdash e_2 : \tau_2 \parallel \theta_2; \Pi'' \quad \theta'_1 \vdash_{\text{unf}} \theta_2 \langle \tau_1 \rangle = \beta \downarrow ([\delta] \rightarrow [\alpha]) \quad \theta = \theta_1 \circ \theta_2 \circ \theta'_1 \quad \bar{\theta} = \text{ftv}(\theta \langle \alpha \rangle) \setminus \text{ftv}(\theta \langle \delta \rangle) \quad \tau = \nabla(\theta \langle \alpha \rangle)[\bar{\theta}/\bar{\theta}]}{\Gamma; \Sigma; \Pi \vdash e_1 e_2 : \tau' \parallel \theta_1 \circ \theta_2 \circ \theta'_1 \circ \theta'_2; \Pi''}$		
<p>TI-IF</p> $\frac{\Gamma; \Sigma; \Pi_{\alpha\beta\gamma\delta\epsilon} \vdash e_1 : \tau_1 \parallel \theta_1; \Pi_1 \quad \theta_1 \langle \Gamma \rangle; \theta_1 \langle \Sigma \rangle; \Pi_1 \vdash e_2 : \tau_2 \parallel \theta_2; \Pi_2 \quad \theta_1 \circ \theta_2 \langle \Gamma \rangle; \theta_1 \circ \theta_2 \langle \Sigma \rangle; \Pi_2 \vdash e_3 : \tau_3 \parallel \theta_3; \Pi_3 \quad \theta = \theta_1 \circ \theta_2 \circ \theta_3 \quad \theta'_2 \vdash_{\text{unf}} \theta \langle \tau_2 \rangle = \alpha \downarrow \beta \quad \theta'_3 \vdash_{\text{unf}} \theta \circ \theta'_2 \langle \tau_3 \rangle = \gamma \downarrow \theta'_2 \langle \beta \rangle \quad \theta'_1 \vdash_{\text{unf}} \theta \circ \theta'_2 \circ \theta'_3 \langle \tau_1 \rangle = \delta \downarrow \text{bool} \quad \theta' = \theta \circ \theta'_1 \circ \theta'_2 \circ \theta'_3 \quad \tau_0 = \epsilon \downarrow \text{join}(\theta' \langle \tau_2 \rangle, \theta' \langle \tau_3 \rangle)}{\Gamma; \Sigma; \Pi \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau_0 \parallel \theta'; \Pi_3}$		
<p>TI-SET</p> $\frac{\Gamma; \Sigma; \Pi_{\alpha\beta} \vdash e_1 : \tau_1 \parallel \theta_1; \Pi' \quad \theta_1 \langle \Gamma \rangle; \theta_1 \langle \Sigma \rangle; \Pi' \vdash e_2 : \tau_2 \parallel \theta_2; \Pi'' \quad \theta'_1 \vdash_{\text{unf}} \theta_2 \langle \tau_1 \rangle = \Psi \alpha \quad \theta'_2 \vdash_{\text{unf}} \theta'_1 \langle \tau_2 \rangle = \beta \downarrow \theta'_1 \circ \theta_2 \langle \tau_1 \rangle}{\Gamma; \Sigma; \Pi \vdash e_1 := e_2 : \text{unit} \parallel \theta_1 \circ \theta_2 \circ \theta'_1 \circ \theta'_2; \Pi''}$		
<p>TI-DUP</p> $\frac{\Gamma; \Sigma; \Pi_{\alpha} \vdash e : \tau \parallel \theta; \Pi' \quad \tau' = \alpha \downarrow \tau}{\Gamma; \Sigma; \Pi \vdash \text{dup}(e) : \uparrow\tau' \parallel \theta; \Pi'}$	<p>TI-DEREF</p> $\frac{\Gamma; \Sigma; \Pi_{\alpha\beta} \vdash e : \tau \parallel \theta; \Pi' \quad \theta' \vdash_{\text{unf}} \tau = \beta \downarrow \uparrow\alpha}{\Gamma; \Sigma \vdash e^{\wedge} : \theta \circ \theta' \langle \alpha \rangle \parallel \theta \circ \theta'; \Pi'}$	
<p>TI-LET-M [TQ]</p> $\frac{\Gamma; \Sigma; \Pi_{\alpha} \vdash e_1 : \tau_1 \parallel \theta_1; \Pi_1 \quad \theta \vdash_{\text{unf}} \theta_1 \langle \tau \rangle = \alpha \downarrow \tau_1 \quad \theta'; x; e_2 \vdash_{\text{solve}} \theta \circ \theta_1 \langle \tau \rangle : \gg \tau' \quad \vdash_{\text{loc}} x : \sigma \quad \theta_1 \circ \theta \circ \theta' \langle \Gamma \rangle; \theta_1 \circ \theta \circ \theta' \langle \Sigma \rangle; e_1 \vdash_{\text{gen}} \tau \leq \sigma \quad \theta_1 \circ \theta \circ \theta' \langle \Gamma \rangle, x \mapsto \sigma; \theta_1 \circ \theta \circ \theta' \langle \Sigma \rangle; \Pi_1 \vdash e_2 : \tau_2 \parallel \theta_2; \Pi_2}{\Gamma; \Sigma; \Pi \vdash \text{let}^{\psi} x[\tau] = e_1 \text{ in } e_2 : \tau_2 \parallel \theta \circ \theta' \circ \theta_1 \circ \theta_2; \Pi_2}$		
<p>TI-LET-P [TQ]</p> $\frac{\Gamma; \Sigma; \Pi_{\alpha} \vdash e_1 : \tau_1 \parallel \theta_1; \Pi_1 \quad \theta \vdash_{\text{unf}} \theta_1 \langle \tau \rangle = \alpha \downarrow \tau_1 \quad \theta'; x; e_2 \vdash_{\text{solve}} \theta \circ \theta_1 \langle \tau \rangle : \gg \tau' \quad \vdash_{\text{term}} x : \sigma \quad \theta_1 \circ \theta \circ \theta' \langle \Gamma \rangle; \theta_1 \circ \theta \circ \theta' \langle \Sigma \rangle; e_1 \vdash_{\text{gen}} \tau \leq \sigma \quad \theta_1 \circ \theta \circ \theta' \langle \Gamma \rangle, x \mapsto \sigma; \theta_1 \circ \theta \circ \theta' \langle \Sigma \rangle; \Pi_1 \vdash e_2 : \tau_2 \parallel \theta_2; \Pi_2}{\Gamma; \Sigma; \Pi \vdash \text{let}^{\vee} x[\tau] = e_1 \text{ in } e_2 : \tau_2 \parallel \theta \circ \theta' \circ \theta_1 \circ \theta_2; \Pi_2}$		

Table 4.9: Type Inference Rules

its argument type. We constrain $\bar{\theta}$ to be immutable by shallow substitution with $[\bar{\theta}]$. This ensures that mutability not induced by the argument does not affect the return type. Later unification with τ_2 ensures that the mutability induced by the actual argument is preserved in the hint. Similarly, the TI-IF rule infers copy compatible types for the two branches and the result. It calculates the most immutable type as the hint for the result type by computing the *join* [51] of $\theta' \langle \tau_2 \rangle$ and $\theta' \langle \tau_3 \rangle$ (note that two copy compatible types always have a join). Other rules are similar.

4.3.12 Unification

The unification judgment $\theta \vdash_{\text{unf}} \tau_1 = \tau_2$ is understood as τ_1 unifies with (can be transformed to structurally equal) τ_2 under the substitution θ . Eager unification rules can be found in Table 4.10. Definitions for some auxiliary functions used by the unifier are given in Table 4.11.

The unification judgment $\theta \vdash_{\text{unf}} \tau_1 = \tau_2$ is understood as τ_1 unifies with τ_2 under the substitution θ . The interesting cases are U-CT1 and U-CT2. U-CT1 shown the unification of a maybe type $\tau_1 \downarrow \tau'_1$ with an unconstrained type τ_2 .

UNIFY $\frac{\theta \Vdash_u \mathfrak{R}(\tau_1) = \mathfrak{R}(\tau_2)}{\theta \Vdash_{unf} \tau_1 = \tau_2}$	U-REFL $\frac{}{\emptyset \Vdash_u \tau = \tau}$	U-COMMUT $\frac{\theta \Vdash_u \tau = \tau'}{\theta \Vdash_u \tau' = \tau}$	U-TVAR $\frac{\alpha \neq \tau}{[\alpha \rightsquigarrow \tau] \Vdash_u \alpha = \tau}$	U-REF $\frac{\theta \Vdash_u \tau_1 = \tau_2}{\theta \Vdash_u \uparrow \tau_1 = \uparrow \tau_2}$
U-MUT $\frac{\theta \Vdash_u \tau_1 = \tau_2}{\theta \Vdash_u \Psi \tau_1 = \Psi \tau_2}$		U-FN $\frac{\theta \Vdash_u \tau_1 = \tau'_1 \quad \theta' \Vdash_u \theta \langle \tau_2 \rangle = \theta \langle \tau'_2 \rangle \quad \Vdash_{acyclic} \theta \circ \theta'}{\theta \circ \theta' \Vdash_u \tau_1 \rightarrow \tau_2 = \tau'_1 \rightarrow \tau'_2}$		
U-CT1 $\frac{\tau_2 \neq \tau_3 \downarrow \tau'_3 \quad \tau_2 \neq \alpha \quad \theta \Vdash_u \mathfrak{S}(\tau'_1) = \nabla(\tau_2) \quad \theta' \Vdash_u \theta \langle \tau_1 \rangle = \theta \langle \tau_2 \rangle \quad \Vdash_{acyclic} \theta \circ \theta'}{\theta \circ \theta' \Vdash_u \tau_1 \downarrow \tau'_1 = \tau_2}$				
U-CT2 $\frac{\theta \Vdash_u \mathfrak{S}(\tau'_1) = \mathfrak{S}(\tau'_2) \quad \theta' \Vdash_u \theta \langle \tau_1 \rangle = \theta \langle \tau_2 \rangle \quad \Vdash_{acyclic} \theta \circ \theta'}{\theta \circ \theta' \Vdash_u \tau_1 \downarrow \tau'_1 = \tau_2 \downarrow \tau'_2}$		U-CEIL $\frac{\theta \Vdash_u \tau_1 = \tau_2}{\theta \Vdash_u \lceil \tau_1 \rceil = \lceil \tau_2 \rceil}$	U-FLOOR $\frac{\theta \Vdash_u \tau_1 = \tau_2}{\theta \Vdash_u \lfloor \tau_1 \rfloor = \lfloor \tau_2 \rfloor}$	

Table 4.10: Unification Rules

$\frac{\tau \neq \tau_1 \downarrow \tau_2}{\mathfrak{S}(\tau) = \nabla(\tau)}$	$\frac{\tau = \tau_1 \downarrow \tau_2 \quad \mathfrak{S}(\tau_2) = \tau'_2}{\mathfrak{S}(\tau) = \tau'_2}$	$\mathfrak{R}(\text{unit}) = \lfloor \text{unit} \rfloor$	$\mathfrak{R}(\text{bool}) = \lfloor \text{bool} \rfloor$
$\mathfrak{R}(\alpha) = \alpha$	$\frac{\mathfrak{R}(\tau) = \tau'}{\mathfrak{R}(\Psi \tau) = \lceil \Psi \tau' \rceil}$	$\frac{\mathfrak{R}(\tau) = \tau'}{\mathfrak{R}(\uparrow \tau) = \lfloor \uparrow \tau' \rfloor}$	$\frac{\mathfrak{R}(\tau_1) = \tau'_1 \quad \mathfrak{R}(\tau_2) = \tau'_2}{\mathfrak{R}(\tau_1 \rightarrow \tau_2) = \lfloor \tau'_1 \rightarrow \tau'_2 \rfloor}$
$\frac{\mathfrak{R}(\tau_1) = \tau'_1 \quad \mathfrak{R}(\tau_2) = \tau'_2}{\mathfrak{R}(\tau_1 \downarrow \tau_2) = \tau'_1 \downarrow \tau'_2}$	$\frac{\mathfrak{R}(\nabla(\tau)) = \tau' \quad \tau' = \lfloor \tau'' \rfloor}{\mathfrak{R}(\lfloor \tau \rfloor) = \tau'}$	$\frac{\mathfrak{R}(\nabla(\tau)) = \tau' \quad \tau' \neq \lfloor \tau'' \rfloor}{\mathfrak{R}(\lfloor \tau \rfloor) = \lfloor \tau' \rfloor}$	$\frac{\mathfrak{R}(\Delta(\tau)) = \tau'}{\mathfrak{R}(\lceil \tau \rceil) = \tau'}$

Table 4.11: Auxiliary Functions for Unification

In this case, an immutable version of the constraint is extracted by the \mathfrak{S} operator, and is unified with an immutable version of τ_2 . Once compatibility is established, we unify the type τ_1 to equal τ_2 . U-CT2 shows the unification of two maybe types. After establishing compatibility, we unify the actual types τ_1 and τ_2 so that they ultimately resolve to the same type.

The function $\mathfrak{R}(\tau)$ transforms a type τ into an equivalent canonical form where all meta constructors are made explicit at all structural levels of τ . Strictly speaking, unified types are equal only under the \mathfrak{R} relationship. However, since $\mathfrak{R}(\tau) \equiv \tau$, we will just say that unified types are equal.

4.3.13 Solving Copy Compatibility Constraints

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 transformed² to the unconstrained type τ_2 by solving all the copy compatibility constraints in τ_1 . The \Vdash_{solve} rule fixes the top-level mutability of an “open” type by examining whether x is the target of an assignment in the expression e . The judgment \Vdash_{sol} will actually solve the constraints, either trivially (there are no constraints, or the constraints have already been solved by unification), or by unifying the type with the constraint. Rules for solving copy compatibility constraints can be found in Table 4.12. The relation $\Vdash_{consistent} \tau_1 \downarrow \tau_2$ is defined in Section 4.3.14.5.

4.3.14 Soundness of Eager Inference

DEFINITION: Normalization of Constrained Types

The eager unification algorithm maintains constraints on types. However, these can be written in a normal form as in the case of equational inference so that all constraints only appear on the outermost type. For example:

² This transformation can be thought of as a (safe) coercion. Hence the use of the operator \gg .

$\frac{\text{SOLVE-KNOWN}}{\theta \vdash_{sol} \tau_1 \downarrow \tau_2 \quad \theta \vdash_{sol} \tau : \gg \tau'}{\theta; x; e \vdash_{solve} \tau : \gg \tau'}$	$\frac{\text{SOLVE-MUT}}{\theta \vdash_{sol} \tau_1 \downarrow \tau_2 : \gg \tau \quad \text{mutated}(x, e)}{\theta; x; e \vdash_{solve} \tau_1 \downarrow \tau_2 : \gg \Delta(\tau)}$		
$\frac{\text{SOLVE-IMMUT}}{\theta \vdash_{sol} \tau_1 \downarrow \tau_2 : \gg \tau \quad \neg \text{mutated}(x, e)}{\theta; x; e \vdash_{solve} \tau_1 \downarrow \tau_2 : \gg \nabla(\tau)}$			
$\frac{\text{SOL-UNIT}}{\emptyset \vdash_{sol} \text{unit} : \gg \text{unit}}$	$\frac{\text{SOL-BOOL}}{\emptyset \vdash_{sol} \text{bool} : \gg \text{bool}}$	$\frac{\text{SOL-FN}}{\theta \vdash_{sol} \tau_1 : \gg \tau'_1 \quad \theta' \vdash_{sol} \theta(\tau_2) : \gg \tau'_2}{\theta \circ \theta' \vdash_{sol} \tau_1 \rightarrow \tau_2 : \gg \theta'(\tau'_1) \rightarrow \tau'_2}$	
$\frac{\text{SOL-MUT}}{\theta \vdash_{sol} \tau : \gg \tau'}{\theta \vdash_{sol} \Psi \tau : \gg \Psi \tau'}$	$\frac{\text{SOL-REF}}{\theta \vdash_{sol} \tau : \gg \tau'}{\theta \vdash_{sol} \uparrow \tau : \gg \uparrow \tau'}$	$\frac{\text{SOL-CEIL}}{\theta \vdash_{sol} \tau : \gg \tau'}{\theta \vdash_{sol} \lceil \tau \rceil : \gg \lceil \tau' \rceil}$	$\frac{\text{SOL-FLOOR}}{\theta \vdash_{sol} \tau : \gg \tau'}{\theta \vdash_{sol} \lfloor \tau \rfloor : \gg \lfloor \tau' \rfloor}$
$\frac{\text{SOL-CT-VAR}}{\tau_1 = \alpha \quad \theta \vdash_{sol} \tau_2 : \gg \tau'_2 \quad \theta' = [\alpha \mapsto \tau'_2]}{\theta \circ \theta' \vdash_{sol} \tau_1 \downarrow \tau_2 : \gg \tau'_2}$	$\frac{\text{SOL-CT-CONST}}{\tau_1 \neq \alpha \quad \models_{consistent} \tau_1 \downarrow \tau_2 \quad \theta \vdash_{sol} \tau_1 : \gg \tau'_1}{\theta \vdash_{sol} \tau_1 \downarrow \tau_2 : \gg \tau'_1}$		

Table 4.12: Solving copy compatibility constraints.

$$\begin{aligned} \alpha \downarrow \uparrow \beta \downarrow \tau &= \alpha \setminus \{ \alpha \cong \uparrow(\beta \setminus \{ \beta \cong \tau \}) \} \\ &= \alpha \setminus \{ \alpha \cong \uparrow \beta, \beta \cong \tau \} \\ &\equiv \alpha \setminus \{ \alpha \preceq \gamma, \uparrow \beta \preceq \gamma, \beta \preceq \delta, \tau \preceq \delta \} \end{aligned}$$

Therefore we define a normalization of inferred types as: $\mathbf{N}(\tau) = \underline{\tau} \setminus \mathcal{C}$, wherein:

1. $\underline{\tau}$ contains no constraints within it.
2. \mathcal{C} only contains copy coercion or equality constraints.
3. All meta-constructors in $\underline{\tau}$ are fully interpreted using the equivalences defined in section 4.3.2. That is, the type $\underline{\tau}$ contains no meta-constructors.

We write $\mathbf{N}(\tau) = \underline{\tau}$ if $\mathbf{N}(\tau) = \underline{\tau} \setminus \mathcal{C}$, and \mathcal{C} consists only of tautological constraints.

DEFINITION: Normalization of Constraint Sets

A constraint set $\underline{\mathcal{C}}$ is said to be the normalized form of \mathcal{C} (that is, $\mathbf{N}(\mathcal{C}) = \underline{\mathcal{C}}$) if:

- Let \mathcal{C}' be a set of constraints equivalent to \mathcal{C} , but expressed only using subtype and equality constraints.
- $\underline{\mathcal{C}} = \text{close}(\mathcal{C}')$. That is, $\underline{\mathcal{C}}$ is written as a set of atomic constraints by using the copy coercion rules defined in Table 4.2 (note that this conversion is total). Further, all transitive relationships are explicitly added

DEFINITION: Normalization of Contexts

The binding context $\underline{\Gamma}$ is said to be the normalized form of Γ (that is, $\mathbf{N}(\Gamma) = \underline{\Gamma}$) if $\forall x : \tau \in \Gamma, x : \underline{\tau} \in \underline{\Gamma}$ and $\mathbf{N}(\tau) = \underline{\tau}$. Similarly, the store context $\underline{\Sigma}$ is said to be the normalized form of Σ (that is, $\mathbf{N}(\Sigma) = \underline{\Sigma}$) if $\forall \ell : \tau \in \Sigma$, it is true that $\ell : \underline{\tau} \in \underline{\Sigma}$ and $\mathbf{N}(\tau) = \underline{\tau}$; and $\forall l : \tau \in \Sigma$, it is true that $l : \underline{\tau} \in \underline{\Sigma}$, and $\mathbf{N}(\tau) = \underline{\tau}$.

DEFINITION: Solvable Entities

Syntax

Solvable Entities $\omega ::= \tau \mid \Gamma \mid \Sigma \mid \omega + \omega$

Note that:

1. Normalization is defined on all atomic solvable entities. We add the rule: If $\mathbf{N}(\omega_1) = \underline{\omega_1} \setminus \mathcal{C}_1$ and $\mathbf{N}(\omega_2) = \underline{\omega_2} \setminus \mathcal{C}_2$ then $\mathbf{N}(\omega_1 + \omega_2) = \underline{\omega_1 + \omega_2} \setminus \mathcal{C}_1 \cup \mathcal{C}_2$.
2. As usual, we write $\mathbf{N}(\omega) = \underline{\omega}$ if $\mathbf{N}(\omega) = \underline{\omega} \setminus \mathcal{C}$ where \mathcal{C} consists only of tautological constraints.
3. Substitution is defined on all atomic solvable entities. We define: $\theta\langle\omega_1 + \omega_2\rangle = \theta\langle\omega_1\rangle + \theta\langle\omega_2\rangle$.
4. For the sake of brevity, we write $\emptyset + \omega = \omega + \emptyset = \omega$.
5. Definition of \subseteq over solvable entities:
 - (a) $\tau \subseteq \tau'$ if the type τ is structurally a part of the type τ' .
 - (b) $\Gamma \subseteq \Gamma'$ if Γ' contains all the mappings in Γ , and possibly more.
 - (c) $\Sigma \subseteq \Sigma'$ if Σ' contains all the mappings in Σ , and possibly more.
 - (d) $\omega_1 + \omega_2 \subseteq \omega'_1 + \omega'_2$ if $\omega_1 \subseteq \omega'_1$ and $\omega_2 \subseteq \omega'_2$.
6. Definition of \supseteq over solvable entities:
 - (a) $\tau \supseteq \tau'$ if the type τ' is structurally a part of the type τ .
 - (b) $\Gamma \supseteq \Gamma'$ if Γ contains all the mappings in Γ' , and possibly more.
 - (c) $\Sigma \supseteq \Sigma'$ if Σ contains all the mappings in Σ' , and possibly more.
 - (d) $\omega_1 + \omega_2 \supseteq \omega'_1 + \omega'_2$ if $\omega_1 \supseteq \omega'_1$ and $\omega_2 \supseteq \omega'_2$.

DEFINITION: Consistency of Solvable Entities

A (possibly) constrained type or context ω is said to be consistent, that is, $\models_{\text{consistent}} \omega$ iff $\mathbf{N}(\omega) = \underline{\omega} \setminus \mathcal{C}$ and $\models_{\text{consistent}} \mathcal{C}$.

DEFINITION: Reachable Store

We define $|\Sigma|^{e_1, e_2, \dots}$ as a store such that $|\Sigma|^{e_1, e_2, \dots} \langle \Sigma \rangle$ and $|\Sigma|^{e_1, e_2, \dots}$ contains mappings for *only* those locations that are reachable from e_1, e_2, \dots (that is, the location is syntactically a part of e_1, e_2, \dots).

DEFINITION: MTVs

We define the function $\text{MTVS}(\omega) = \bar{\alpha}$ 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 $\alpha \downarrow \tau_h$.

$$\begin{aligned}
 \text{MTVS}(\alpha) &= \{\} \\
 \text{MTVS}(\text{unit}) &= \{\} \\
 \text{MTVS}(\text{bool}) &= \{\} \\
 \text{MTVS}(\alpha \downarrow \tau_h) &= \alpha \cup \text{MTVS}(\tau_h) \\
 \text{MTVS}(\tau \downarrow \tau_h) &= \text{MTVS}(\tau) \text{ where } \tau \neq \alpha \\
 \text{MTVS}(\uparrow \tau) &= \text{MTVS}(\tau) \\
 \text{MTVS}(\Psi \tau) &= \text{MTVS}(\tau) \\
 \text{MTVS}(\tau_1 \rightarrow \tau_2) &= \text{MTVS}(\tau_1) \cup \text{MTVS}(\tau_2) \\
 \text{MTVS}(\emptyset) &= \{\} \text{ (empty context).} \\
 \text{MTVS}(\Gamma, x : \tau) &= \text{MTVS}(\Gamma) \cup \text{MTVS}(\tau) \\
 \text{MTVS}(\Sigma, \ell : \tau) &= \text{MTVS}(\Sigma) \cup \text{MTVS}(\tau) \\
 \text{MTVS}(\Sigma, \mathbf{1} : \tau) &= \text{MTVS}(\Sigma) \cup \text{MTVS}(\tau) \\
 \text{MTVS}(\omega_1 + \omega_2) &= \text{MTVS}(\omega_1) \cup \text{MTVS}(\omega_2)
 \end{aligned}$$

DEFINITION: FTVs (Extension)

We need to enhance the definition of FTVS as:

$$\begin{aligned} & \dots \\ \text{ftv}(\tau_1 \downarrow \tau_2) &= \text{ftv}(\tau_1) \cup \text{ftv}(\mathfrak{S}(\tau_2)) \\ \text{ftv}(\omega_1 + \omega_2) &= \text{ftv}(\omega_1) \cup \text{ftv}(\omega_2). \end{aligned}$$

DEFINITION: NTVs

We define the set of unconstrained variables as:

$$\begin{aligned} \text{NTVS}(\alpha) &= \{\alpha\} \\ \text{NTVS}(\text{unit}) &= \{\} \\ \text{NTVS}(\text{bool}) &= \{\} \\ \text{NTVS}(\alpha \downarrow \tau_h) &= \text{NTVS}(\tau_h) \\ \text{NTVS}(\tau \downarrow \tau_h) &= \text{NTVS}(\tau) \text{ where } \tau \neq \alpha \\ \text{NTVS}(\uparrow \tau) &= \text{NTVS}(\tau) \\ \text{NTVS}(\Psi \tau) &= \text{NTVS}(\tau) \\ \text{NTVS}(\tau_1 \rightarrow \tau_2) &= \text{NTVS}(\tau_1) \cup \text{NTVS}(\tau_2) \\ \text{NTVS}(\emptyset) &= \{\} \text{ (empty context).} \\ \text{NTVS}(\Gamma, x : \tau) &= \text{NTVS}(\Gamma) \cup \text{NTVS}(\tau) \\ \text{NTVS}(\Sigma, \ell : \tau) &= \text{NTVS}(\Sigma) \cup \text{NTVS}(\tau) \\ \text{NTVS}(\Sigma, l : \tau) &= \text{NTVS}(\Sigma) \cup \text{NTVS}(\tau) \\ \text{NTVS}(\omega_1 + \omega_2) &= \text{NTVS}(\omega_1) \cup \text{NTVS}(\omega_2) \end{aligned}$$

DEFINITION: TVs

The set of all type variables in a solvable entity is given by TVS() function, defined as follows: Note that this function is different from FTVS(). (see Definition 4.3.14.8.

$$\text{TVS}(\omega) = \text{MTVS}(\omega) \cup \text{NTVS}(\omega)$$

DEFINITION: \vdash_{sol}

We write $\theta \vdash_{sol} \tau$ to mean that the substitution is θ is obtained by solving the copy compatibility constraints in τ . We define solving all solvable entities as follows:

S-EMPTY-CTX	S-EMPTY-TYPE	S-GAMMA
$\frac{}{\emptyset \vdash_{sol} \emptyset}$	$\frac{\theta \vdash_{sol} \tau : \gg \tau''}{\theta \vdash_{sol} \tau}$	$\frac{\theta_1 \vdash_{sol} \tau \quad \theta_2 \vdash_{sol} \theta_1 \langle \Gamma \rangle}{\theta_1 \circ \theta_2 \vdash_{sol} \Gamma, x : \tau}$
S-STORE-HL	S-STORE-SL	S-MULTIPLE
$\frac{\theta_1 \vdash_{sol} \tau \quad \theta_2 \vdash_{sol} \theta_1 \langle \Sigma \rangle}{\theta_1 \circ \theta_2 \vdash_{sol} \Sigma, \ell : \tau}$	$\frac{\theta_1 \vdash_{sol} \tau \quad \theta_2 \vdash_{sol} \theta_1 \langle \Sigma \rangle}{\theta_1 \circ \theta_2 \vdash_{sol} \Sigma, l : \tau}$	$\frac{\theta_1 \vdash_{sol} \omega_1 \quad \theta_2 \vdash_{sol} \theta_1 \langle \omega_2 \rangle}{\theta_1 \circ \theta_2 \vdash_{sol} \omega_1 + \omega_2}$

DEFINITION: \models_{sol}

$$\frac{\text{N}(\theta \langle \omega \rangle) = \theta \langle \omega \rangle \quad \theta = \theta_s \circ \theta_o \quad \text{dom}(\theta_s) = \text{MTVS}(\omega) \quad \theta_o \langle \theta \langle \omega \rangle \rangle = \theta \langle \omega \rangle}{\theta \models_{sol} \omega}$$

That is, θ solves all copy compatibility constraints in ω , but does not otherwise affect the resultant $\theta \langle \omega \rangle$. It is, however, free to contain other substitutions that do not matter wrt $\theta \langle \omega \rangle$. Note that \models_{sol} is a special case of \models and \models_{sol} .

DEFINITION: Strong Consistency

$$\frac{\models_{\text{consistent}} \omega \quad \text{MTVS}(\omega) \cap \text{NTVS}(\omega) = \emptyset}{\models_{\text{cst}} \omega}$$

Since we have $\models_{\text{consistent}} \omega$, if $\alpha \downarrow \tau_{h_1}$ and $\alpha \downarrow \tau_{h_2}$ are structurally a part of ω , we must have $\mathfrak{S}(\alpha \downarrow \tau_{h_1}) \cong \mathfrak{S}(\alpha \downarrow \tau_{h_2})$. Further, $\text{MTVS}(\omega) \cap \text{NTVS}(\omega) = \emptyset$ guarantees that constrained type variables do not appear unconstrained elsewhere in ω . We call this property strong consistency, denoted by \models_{cst} .

DEFINITION: Compatible Solutions (Substitutions)

We write $\theta_1 \cong \theta_2$ if $\forall [\alpha \mapsto \tau_1] \in \theta_1$, it is true that $[\alpha \mapsto \tau_2] \in \theta_2$ (that is, $\text{dom}(\theta_1) = \text{dom}(\theta_2)$), and $\tau_1 \cong \tau_2$.

DEFINITION: Equivalent Substitutions

We write $\omega \vdash_{\text{eqi}} \theta_1 \approx \theta_2$ if $\theta_1 \langle \omega \rangle = \theta_2 \langle \omega \rangle$

For example: $\alpha \vdash_{\text{eqi}} [\alpha \mapsto \tau] \approx [\alpha \mapsto \beta] \circ [\beta \mapsto \tau]$.

DEFINITION: Sub-Substitutions

1. We write $\theta' \sqsubseteq \theta$ if $\exists \theta_1, \theta_2$ such that $\theta = \theta_1 \circ \theta_2$, and $\theta' = \theta_1$.
2. We write $\theta' \sqsubseteq \theta$ if $\exists \theta_1, \theta_2$ such that $\theta = \theta_1 \circ \theta_2$, and $\theta' \cong \theta_1$.

THEOREM: Correctness of Eager Unification

If $\mathbf{N}(\tau_1) = \underline{\tau_1} \setminus \mathcal{C}_1$, and $\mathbf{N}(\tau_2) = \underline{\tau_2} \setminus \mathcal{C}_2$, and $\theta \vdash_{\text{unf}} \tau_1 = \tau_2$, then:

1. If $\models_{\text{UNF}} \mathcal{C}_1$ and $\models_{\text{UNF}} \mathcal{C}_2$, then $\models_{\text{UNF}} \{\underline{\tau_1} = \underline{\tau_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2$
2. If $\models_{\text{consistent}} \tau_1$ and $\models_{\text{consistent}} \tau_2$ then $\models_{\text{consistent}} \theta \langle \tau_1 \rangle + \theta \langle \tau_2 \rangle$ and $\models_{\text{acyclic}} \theta$

THEOREM: Correctness of the Constraint Solver

If $\theta \vdash_{\text{sol}} \omega$, then $\theta \models_{\text{sol}} \omega$.

We will also use the following equivalent forms (or special cases of the above statement):

1. If $\theta \vdash_{\text{sol}} \omega$, then $\models_{\text{consistent}} \theta \langle \omega \rangle$.
2. If $\theta \vdash_{\text{sol}} \omega$, then $\mathbf{N}(\theta \langle \omega \rangle) = \underline{\theta \langle \omega \rangle}$.
3. If $\theta \vdash_{\text{sol}} \tau \gg \tau'$ then $\mathbf{N}(\tau') = \underline{\tau'}$.

THEOREM: Totality of the Constraint Solver

1. If $\models_{\text{consistent}} \omega$ then $\exists \theta$ such that $\theta \vdash_{\text{sol}} \omega$.
2. If $\theta \models_{\text{sol}} \omega$ then $\exists \theta' \sqsubseteq \theta$ such that $\theta' \vdash_{\text{sol}} \omega$.

THEOREM: Decidability of Unification and Solver

$\theta \vdash_{unf} \tau_1 = \tau_2$ and $\theta \vdash_{sol} \tau : \gg \tau'$ are decidable.

Proof: The unifier and constraint solver builds a solution tree by always invoking itself types having *smaller* shapes of types (except for the U-COMMUT rule, but we can consider a canonical derivation in which no two U-COMMUT rules are used consecutively in the unification derivation). Since types are of bounded size, and there are no infinite or circular substitutions, these derivations must be bounded.

LEMMA: Properties of \models_{sol}

1. If $\theta \models_{sol} \omega$ then $\models_{consistent} \theta\langle\omega\rangle$.
2. If $\models_{consistent} \theta\langle\omega\rangle$ then $\exists \theta$ such that $\theta \models_{sol} \omega$.
3. $\theta \models_{sol} \omega$ iff $N(\theta\langle\omega\rangle) = \underline{\theta\langle\omega\rangle}$.

LEMMA: Properties of Equivalent Substitutions

If $\omega \vdash_{eqi} \theta_1 \approx \theta_2$, then:

1. $\theta_1 \models_{sol} \omega$ implies $\theta_2 \models_{sol} \omega$.
2. $\models_{consistent} \theta_1\langle\omega\rangle$ implies $\models_{consistent} \theta_2\langle\omega\rangle$.
3. $\models_{cst} \theta_1\langle\omega\rangle$ implies $\models_{cst} \theta_2\langle\omega\rangle$.
4. $N(\theta_1\langle\omega\rangle) = \underline{\theta_1\langle\omega\rangle}$ implies $N(\theta_2\langle\omega\rangle) = \underline{\theta_2\langle\omega\rangle}$.

Proof: Evident from the definition of equivalent substitutions.

LEMMA: Strong Consistency Implies Consistency

If $\models_{cst} \omega$, then $\models_{consistent} \omega$.

Proof: Evident from Definition 4.3.14.13.

LEMMA: Properties of Compatible Constrained Types

1. If $N(\tau) = \underline{\tau}$, and $N(\tau') = \underline{\tau'}$, and $\tau \cong \tau'$, then $\lfloor \tau \rfloor \preceq \underline{\tau'} \preceq \lfloor \tau \rfloor$
2. If $N(\theta\langle\tau\rangle) = \underline{\theta\langle\tau\rangle}$, then $\lfloor \theta\langle\tau\rangle \rfloor \preceq \underline{\Delta(\theta\langle\tau\rangle)} \preceq \underline{\theta\langle\tau\rangle} \preceq \underline{\nabla(\theta\langle\tau\rangle)} \preceq \lfloor \theta\langle\tau\rangle \rfloor$
3. If $N(\theta\langle\tau\rangle) = \underline{\theta\langle\tau\rangle}$ and $\{\bar{\theta}\} \subseteq \text{ftv}(\tau)$ and $\tau' = \tau[\bar{\theta}/\bar{\theta}]$, then $\lfloor \theta\langle\tau\rangle \rfloor \preceq \underline{\theta\langle\tau\rangle} \preceq \underline{\theta\langle\tau'\rangle} \preceq \underline{\nabla(\theta\langle\tau'\rangle)}$
4. If $N(\theta\langle\tau\rangle) = \underline{\theta\langle\tau\rangle}$ and $\{\bar{\theta}\} \subseteq \text{ftv}(\tau)$ and $\tau' = \tau[\bar{\theta}/\bar{\theta}]$, then $\underline{\Delta(\theta\langle\tau'\rangle)} \preceq \underline{\theta\langle\tau'\rangle} \preceq \underline{\theta\langle\tau\rangle} \preceq \lfloor \theta\langle\tau\rangle \rfloor$.

Proof: Evident from the definition of copy coercion relationships in Section 4.3.3.

LEMMA: Relationship of Solutions

If $\theta_1 \models_{sol} \omega$ and $\theta_2 \models_{sol} \omega$ then $\exists \theta_{11}, \theta_{12}, \theta_{21}, \theta_{22}$ such that

1. $\theta_1 = \theta_{11} \circ \theta_{12}$ and $\theta_2 = \theta_{21} \circ \theta_{22}$
2. $dom(\theta_{11}) = dom(\theta_{21}) = \text{MTVS}(\omega)$
3. $\theta_{11} \cong \theta_{21}$
4. $\theta_{12}(\underline{\omega}) = \theta_{22}(\underline{\omega}) = \underline{\omega}$

LEMMA: Uniqueness of Solutions Produced by the Solver

If $\theta_1 \vdash_{sol} \omega$ and $\theta_2 \vdash_{sol} \omega$, then $\theta_1 = \theta_2$.

LEMMA: Relationship between \vdash_{solve} and \vdash_{sol}

If $\theta_1 \vdash_{sol} \tau \triangleright\triangleright \tau'$ and $\theta_2; x; e \vdash_{solve} \tau \triangleright\triangleright \tau''$ then $\tau' \cong \tau''$ and $\theta_1 \cong \theta_2$.

LEMMA: Commutativity of $+$ over Solvable Entities

1. If $\theta \models_{sol} \omega_1 + \omega_2$ then $\theta \models_{sol} \omega_2 + \omega_1$.
2. If $\theta \vdash_{sol} \omega_1 + \omega_2$ then $\exists \theta' \cong \theta$ such that $\theta' \vdash_{sol} \omega_2 + \omega_1$.
3. If $\models_{consistent} \omega_1 + \omega_2$ then $\models_{consistent} \omega_2 + \omega_1$.
4. If $\models_{cst} \omega_1 + \omega_2$ then $\models_{cst} \omega_2 + \omega_1$.

LEMMA: Weakening over Solvable Entities

1. If $\models_{consistent} \omega_1 + \omega_2$ then $\models_{consistent} \omega_1$ and $\models_{consistent} \omega_2$.
2. If $\models_{consistent} \omega$ and $\omega' \subseteq \omega$ then $\models_{consistent} \omega'$.
3. If $\models_{cst} \omega_1 + \omega_2$ then $\models_{cst} \omega_1$ and $\models_{cst} \omega_2$.
4. If $\models_{cst} \omega$ and $\omega' \subseteq \omega$ then $\models_{cst} \omega'$.
5. If $\mathbf{N}(\theta(\omega)) = \underline{\theta(\omega)}$ and $\omega' \subseteq \omega$, then $\mathbf{N}(\theta(\omega')) = \underline{\theta(\omega')}$.
6. If $\theta \models_{sol} \omega_1 + \omega_2$ then $\exists \theta' \subseteq \theta$ and $\exists \theta'' \subseteq \theta$ such that $\theta' \models_{sol} \omega_1$ and $\theta'' \models_{sol} \omega_2$.
7. If $\theta \models_{sol} \omega$ and $\omega' \subseteq \omega$, then $\exists \theta' \subseteq \theta$ such that $\theta' \models_{sol} \omega'$.
8. If $\models_{cst} \omega_1 + \omega_2$ and $\theta \models_{sol} \omega_1 + \omega_2$ then $\theta \models_{sol} \omega_1$ and $\theta \models_{sol} \omega_2$.
9. If $\models_{cst} \omega$ and $\theta \models_{sol} \omega$ and $\omega' \subseteq \omega$, then $\theta \models_{sol} \omega'$.

LEMMA: Extension to Weakening Lemma 4.3.7.6.

If $\underline{\Gamma}; \underline{\Sigma} \vdash e : \underline{\tau}$ and $\Gamma' \supseteq \Gamma$ and $\Sigma' \supseteq \Sigma$ and $\mathbf{N}(\Gamma') = \underline{\Gamma'}$ and $\mathbf{N}(\Sigma') = \underline{\Sigma'}$, then, $\underline{\Gamma'}; \underline{\Sigma'} \vdash e : \underline{\tau}$.

LEMMA: Substitution on Strongly Compatible Entities

If

1. $\models_{cst} \omega_f + \omega + \omega_s$
2. $\models_{cst} \theta\langle\omega\rangle$ for some substitution θ .
3. $dom(\theta) \cap TVS(\omega_f + \omega + \omega_s) \subseteq TVS(\omega)$.

Then, $\models_{cst} \theta\langle\omega_f + \omega + \omega_s\rangle$.

Proof: From assumption (3), we know that only substitutions in θ that can affect $\omega_f + \omega + \omega_s$ are substitutions to type variables that are present in ω . From the definition of strong solubility, we know that all constrained types are compatibly constrained throughout the solvable entity. Therefore a substitution for some type variable within ω cannot violate the strong consistency of $\omega_f + \omega + \omega_s$.

Note that ω_f or ω_s need not always present since we can imagine the presence of \emptyset in that position.

LEMMA: Substitution Preserves Compatibility of Solutions

If

1. $\theta_1 \models_{sol} \omega$
2. $\theta_2 \models_{sol} \theta\langle\omega\rangle$ for some θ
3. $dom(\theta_1) \cap dom(\theta_2 \circ \theta) = MTVS(\omega)$

Then, $\exists \theta'_1, \theta_0$ such that

1. $\theta_1 \cong \theta'_1$
2. $\omega \vdash_{eqi} \theta_2 \circ \theta \approx \theta'_1 \circ \theta_0$.

Proof: By construction of θ'_1 and θ_0 .

1. For every substitution $[\alpha \mapsto \tau] \in \theta_1$, where $\alpha \notin MTVS(\omega)$, add it to θ'_1 .
2. For every substitution $[\alpha \mapsto \tau] \in \theta_2 \circ \theta$, where $\alpha \notin MTVS(\omega)$, add it to θ_0 .
3. $\forall \alpha \in MTVS(\omega)$, $\exists [\alpha \mapsto \tau] \in \theta_1$ and $\exists [\alpha \mapsto \tau'] \in \theta_2 \circ \theta$. If α appears within ω in the form $\alpha \downarrow \tau_h$, we must have $\tau \cong \theta_1\langle\mathfrak{S}(\tau_h)\rangle$. We must also have $\tau' \cong \theta_2 \circ \theta\langle\mathfrak{S}(\tau_h)\rangle$
 - (a) If $\tau \cong \tau'$, then add $[\alpha \mapsto \tau']$ to θ'_1 .
 - (b) If $\tau \not\cong \tau'$ Since we have $\models_{consistent} \theta_2 \circ \theta\langle\omega\rangle$ (from assumption (2) and Theorem 4.3.14.18) and the fact that substitution must preserve the shape of a solvable entity, τ' can only differ from τ in top-level mutability, and/or by being a more specialized type (has some substitutions for type variables within τ). Therefore, $\exists \theta''$ (which possibly uses fresh type variables) such that the substitution $[\alpha \mapsto \tau']$ in $\theta_2 \circ \theta$ can be equivalently re-written as $[\alpha \mapsto \tau''] \circ \theta''$ where $\tau \cong \tau''$. Now add $[\alpha \mapsto \tau'']$ to θ'_1 and θ'' to θ_0 .

LEMMA: Unification Preserves Strong Consistency

If $\models_{cst} \tau_1 + \tau_2$ and $\theta \vdash_{unf} \tau_1 = \tau_2$ then $\models_{cst} \theta\langle\tau_1\rangle + \theta\langle\tau_2\rangle$

Proof: Follows from Theorem 4.3.14.17, and by observing the rules U-TVAR, U-CT1 and U-CT2.

LEMMA: Corollary to Lemma 4.3.14.33.

If $\frac{}{\text{cst}} \vdash_{\text{unf}} \omega + \tau + \tau'$ and $\theta_v \vdash_{\text{unf}} \tau' = \tau$ then $\frac{}{\text{cst}} \theta_v \langle \omega \rangle + \theta_v \langle \tau \rangle + \theta_v \langle \tau' \rangle$

Proof: Follows from Lemma 4.3.14.33 and Lemma 4.3.14.31.

LEMMA: Unification of Maybe Types

If $\theta_u \vdash_{\text{unf}} \tau_1 = \tau_0 \downarrow \tau_2$ and $\theta_s \frac{}{\text{sol}} \theta_u \langle \tau_1 \rangle + \theta_u \langle \tau_0 \downarrow \tau_2 \rangle$, then:

1. $\frac{}{\text{cst}} \theta_u \circ \theta_s \langle \tau_1 \rangle \preceq: \frac{}{\text{cst}} \theta_u \circ \theta_s \langle \tau_2 \rangle$ and $\frac{}{\text{cst}} \theta_u \circ \theta_s \langle \tau_2 \rangle \preceq: \frac{}{\text{cst}} \theta_u \circ \theta_s \langle \tau_1 \rangle$
2. $\frac{}{\text{cst}} \theta_u \circ \theta_s \langle \tau_1 \rangle \preceq: \frac{}{\text{cst}} \nabla(\theta_u \circ \theta_s \langle \tau_2 \rangle)$ and $\frac{}{\text{cst}} \theta_u \circ \theta_s \langle \tau_2 \rangle \preceq: \frac{}{\text{cst}} \Delta(\theta_u \circ \theta_s \langle \tau_1 \rangle)$

Proof: From the definition of maybe types, we have $\tau_0 \cong \tau_2$. Due to the unification $\theta_u \vdash_{\text{unf}} \tau_1 = \tau_0 \downarrow \tau_2$, we have $\theta_u \langle \tau_1 \rangle \cong \theta_u \langle \tau_2 \rangle$. From $\theta_s \frac{}{\text{sol}} \theta_u \langle \tau_1 \rangle + \theta_u \langle \tau_0 \downarrow \tau_2 \rangle$ and Lemma 4.3.14.21, we have: $\frac{}{\text{consistent}} \theta_s \circ \theta_u \langle \tau_1 \rangle + \theta_s \circ \theta_u \langle \tau_0 \downarrow \tau_2 \rangle$. Therefore, we must have: $\theta_s \circ \theta_u \langle \tau_1 \rangle \cong \theta_s \circ \theta_u \langle \tau_2 \rangle$. Again, from Lemma 4.3.14.21, we have: $\mathbf{N}(\theta_s \circ \theta_u \langle \tau_1 \rangle) = \frac{}{\text{cst}} \theta_s \circ \theta_u \langle \tau_1 \rangle$ and $\mathbf{N}(\theta_s \circ \theta_u \langle \tau_0 \downarrow \tau_2 \rangle) = \frac{}{\text{cst}} \theta_s \circ \theta_u \langle \tau_0 \downarrow \tau_2 \rangle$. Since normalization does not violate any properties, we have: $\frac{}{\text{cst}} \theta_s \circ \theta_u \langle \tau_1 \rangle \cong \frac{}{\text{cst}} \theta_s \circ \theta_u \langle \tau_2 \rangle$. The conclusions are now evident from the definition of copy coercions in Section 4.3.3.

LEMMA: Corollary to Lemma 4.3.14.35

If $\theta_u \vdash_{\text{unf}} \tau_1 = \tau_0 \downarrow \tau_2$ and $\theta_s \frac{}{\text{sol}} \theta \circ \theta_u \langle \tau_1 \rangle + \theta \circ \theta_u \langle \tau_0 \downarrow \tau_2 \rangle$, then:

1. $\frac{}{\text{cst}} \theta_s \circ \theta \circ \theta_u \langle \tau_1 \rangle \preceq: \frac{}{\text{cst}} \theta_s \circ \theta \circ \theta_u \langle \tau_2 \rangle$ and $\frac{}{\text{cst}} \theta_s \circ \theta \circ \theta_u \langle \tau_2 \rangle \preceq: \frac{}{\text{cst}} \theta_s \circ \theta \circ \theta_u \langle \tau_1 \rangle$
2. $\frac{}{\text{cst}} \theta_s \circ \theta \circ \theta_u \langle \tau_1 \rangle \preceq: \frac{}{\text{cst}} \nabla(\theta_s \circ \theta \circ \theta_u \langle \tau_2 \rangle)$ and $\frac{}{\text{cst}} \theta_s \circ \theta \circ \theta_u \langle \tau_2 \rangle \preceq: \frac{}{\text{cst}} \Delta(\theta_s \circ \theta \circ \theta_u \langle \tau_1 \rangle)$

Proof: Straightforward extension to Lemma 4.3.14.35.

LEMMA: Inferred Substitutions are Consistent.

If $\Gamma; \Sigma; \Pi \vdash_i e : \tau \parallel \theta_u; \Pi'$, then $\frac{}{\text{cst}} \theta_u \langle \Gamma \rangle + \theta_u \langle |\Sigma|^e \rangle + \tau$.

Proof: By induction on the derivation of $\Gamma; \Sigma; \Pi \vdash_i e : \tau \parallel \theta_u; \Pi'$ using Lemma 4.3.14.33, and noting the fact that (1) all substitutions produced during inference are in turn produced by the unifier (or the solver) (2) all maybe types in the inference rules are introduced with fresh type variables (3) the two parts of a maybe type are never separated from each other.

Since the store Σ can contain arbitrary typing assumptions in addition to the ones required for this derivation, we define the property on a canonicalized subset of the store.

LEMMA: Inferred Substitutions are Cumulatively Consistent.

If

1. $\Gamma; \Sigma; \Pi \vdash_i e_1 : \tau_1 \parallel \theta_{u_1}; \Pi'$
2. $\theta_{u_1} \langle \Gamma \rangle; \theta_{u_1} \langle \Sigma \rangle; \Pi' \vdash_i e_2 : \tau_2 \parallel \theta_{u_2}; \Pi''$

Then $\frac{}{\text{cst}} \theta_{u_2} \circ \theta_{u_1} \langle \Gamma \rangle + \theta_{u_2} \circ \theta_{u_1} \langle |\Sigma|^{e_1, e_2} \rangle + \theta_{u_2} \langle \tau_1 \rangle + \tau_2$

Proof: Since θ_{u_2} is obtained in an environment that already has applied the substitutions in θ_{u_1} , and since this derivation occurs on a different tape, θ_{u_1} and θ_{u_2} contain mutually exclusive substitutions.

The only effect θ_{u_2} can have on τ_1 is by providing a substitution for some type variable α which is a non-universally quantified free type variables in $\theta_{u_1} \langle \Gamma \rangle$ and is present in τ_1 . The result now follows from Lemma 4.3.14.37 and Lemma 4.3.14.31.

LEMMA: Substitution on Declarative Derivation [Extension to Lemma 4.3.10.5].

If

1. $\underline{\Gamma}; \underline{\Sigma} \vdash e : \underline{\tau}$
2. θ is some substitution such that $\mathbf{N}(\theta\langle\tau + \Gamma + \Sigma\rangle) = \underline{\theta\langle\tau + \Gamma + \Sigma\rangle}$

Then, $\underline{\theta\langle\Gamma\rangle}; \underline{\theta\langle\Sigma\rangle} \vdash e : \underline{\theta\langle\tau\rangle}$ **Proof:** Straightforward extension to Lemma 4.3.10.5.**LEMMA: Corollary to Lemma 4.3.14.39.**

If

1. $\underline{\Gamma}; \underline{\Sigma} \vdash e : \underline{\tau}$
2. θ is some substitution such that $\models_{\text{consistent}} \theta\langle\tau + \Gamma + \Sigma\rangle$

Then, $\exists \theta_e$ such that: $\underline{\theta_e \circ \theta\langle\Gamma\rangle}; \underline{\theta_e \circ \theta\langle\Sigma\rangle} \vdash e : \underline{\theta_e \circ \theta\langle\tau\rangle}$ **Proof:** Straightforward extension to Lemma 4.3.14.39. The extra substitution θ_e is necessary to solve any constraints introduced by the substitution θ .**LEMMA: Elimination of Irrelevant Substitutions.**

If:

1. $\theta \models_{\text{sol}} \tau + \Gamma + \Sigma$
2. $\underline{\theta\langle\Gamma\rangle}; \underline{\theta\langle\Sigma\rangle} \vdash e : \underline{\theta\langle\tau\rangle}$

Then, $\exists \theta' \subseteq \theta$ such that $\theta' \models_{\text{sol}} \tau + \Gamma + |\Sigma|^e$ and $\underline{\theta'\langle\Gamma\rangle}; \underline{\theta'\langle|\Sigma|^e\rangle} \vdash e : \underline{\theta'\langle\tau\rangle}$ **Proof:** Follows from Lemma 4.3.14.29 and by observing the declarative type rules in Table 4.4.**LEMMA: Equivalent Substitutions Preserve Declarative Derivation.**

If

1. $\tau + \Gamma + \Sigma \models_{\text{eqi}} \theta_1 \approx \theta_2$
2. $\underline{\theta_1\langle\Gamma\rangle}; \underline{\theta_1\langle\Sigma\rangle} \vdash e : \underline{\theta_1\langle\tau\rangle}$

Then $\underline{\theta_2\langle\Gamma\rangle}; \underline{\theta_2\langle\Sigma\rangle} \vdash e : \underline{\theta_2\langle\tau\rangle}$ **Proof:** From Definition 4.3.14.15, we have $\theta_1\langle\tau + \Gamma + \Sigma\rangle = \theta_2\langle\tau + \Gamma + \Sigma\rangle$. That is, $\theta_1\langle\tau\rangle = \theta_2\langle\tau\rangle$ and $\theta_1\langle\Gamma\rangle = \theta_2\langle\Gamma\rangle$ and $\theta_1\langle\Sigma\rangle = \theta_2\langle\Sigma\rangle$. The result is now evident from assumption (2).**LEMMA: Compatible Substitutions Preserve Declarative Derivation.**

If

1. $\theta_s \models_{\text{sol}} \tau + \Gamma + \Sigma$

2. $\underline{\theta_s \langle \Gamma \rangle}; \underline{\theta_s \langle \Sigma \rangle} \vdash e : \underline{\theta_s \langle \tau \rangle}$
3. $\theta_s \cong \theta'_s$

Then, $\underline{\theta'_s \langle \Gamma \rangle}; \underline{\theta'_s \langle \Sigma \rangle} \vdash e : \underline{\theta'_s \langle \tau \rangle}$.

Proof: From Definition 4.3.14.12, we know that $\theta_s = \theta_m \circ \theta_o$ where θ_m contains substitutions to $\text{MTVS}(\tau + \Gamma + \Sigma)$ and θ_o is irrelevant to $\tau + \Gamma + \Sigma$. From the definition of compatibility of substitutions, θ'_s must have compatible substitutions to *only* these type variables. The required result can be obtained by straightforward induction on the derivation of $\underline{\theta'_s \langle \Gamma \rangle}; \underline{\theta'_s \langle \Sigma \rangle} \vdash e : \underline{\theta'_s \langle \tau \rangle}$.

LEMMA: Corollary to Lemma 4.3.14.43.

If

1. $\theta_s \models_{\text{sol}} \tau + \Gamma + \Sigma$
2. $\underline{\theta_s \langle \Gamma \rangle}; \underline{\theta_s \langle \Sigma \rangle} \vdash e : \underline{\theta_s \langle \tau \rangle}$
3. $\theta'_s \models_{\text{sol}} \tau + \Gamma + \Sigma$

Then, $\underline{\theta'_s \langle \Gamma \rangle}; \underline{\theta'_s \langle \Sigma \rangle} \vdash e : \underline{\theta'_s \langle \tau \rangle}$.

Proof: Follows from Lemma 4.3.14.25 and Lemma 4.3.14.43.

LEMMA: Corollary to Lemma 4.3.14.44.

If

1. $\theta_e \models_{\text{sol}} \tau + \Gamma + |\Sigma|^e$
2. $\underline{\theta_e \langle \Gamma \rangle}; \underline{\theta_e \langle |\Sigma|^e \rangle} \vdash e : \underline{\theta_e \langle \tau \rangle}$
3. $\theta_s \models_{\text{sol}} \tau + \Gamma + \Sigma$

Then, $\underline{\theta_s \langle \Gamma \rangle}; \underline{\theta_s \langle \Sigma \rangle} \vdash e : \underline{\theta_s \langle \tau \rangle}$.

Proof:

1. From assumption (3), the fact that $|\Sigma|^e \subseteq \Sigma$, and Lemma 4.3.14.29, we conclude that $\exists \theta'_s \subseteq \theta_s$ such that $\theta'_s \models_{\text{sol}} \tau + \Gamma + |\Sigma|^e$.
2. From assumption (2) and (3) and case (1) above, and Lemma 4.3.14.44, we have $\underline{\theta'_s \langle \Gamma \rangle}; \underline{\theta'_s \langle |\Sigma|^e \rangle} \vdash e : \underline{\theta'_s \langle \tau \rangle}$.
3. From case (1), we can write: $\theta_s = \theta_o \circ \theta'_s$.
4. From assumption (3) and Lemma 4.3.14.21, we have $\mathbf{N}(\theta_s \langle \Gamma + \Sigma + \tau \rangle) = \underline{\theta_s \langle \Gamma + \Sigma + \tau \rangle}$.
5. Since we have: $|\Sigma|^e \subseteq \Sigma$, (and thus $\theta_s \langle |\Sigma|^e \rangle \subseteq \theta_s \langle \Sigma \rangle$), from case (4) and Lemma 4.3.14.29, we obtain $\mathbf{N}(\theta_s \langle \Gamma + |\Sigma|^e + \tau \rangle) = \underline{\theta_s \langle \Gamma + |\Sigma|^e + \tau \rangle}$.
6. Case (5) can be re-written as: $\mathbf{N}(\theta_o \langle \theta'_s \langle \Gamma + |\Sigma|^e + \tau \rangle \rangle) = \underline{\theta_o \langle \theta'_s \langle \Gamma + |\Sigma|^e + \tau \rangle \rangle}$.
7. From cases (2), (6) and Lemma 4.3.14.39, we have: $\underline{\theta_o \langle \theta'_s \langle \Gamma \rangle \rangle}; \underline{\theta_o \langle \theta'_s \langle |\Sigma|^e \rangle \rangle} \vdash e : \underline{\theta_o \langle \theta'_s \langle \tau \rangle \rangle}$. That is, $\underline{\theta_s \langle \Gamma \rangle}; \underline{\theta_s \langle |\Sigma|^e \rangle} \vdash e : \underline{\theta_s \langle \tau \rangle}$.
8. Since we have: $\Sigma \supseteq |\Sigma|^e$, (and thus $\theta_s \langle \Sigma \rangle \supseteq \theta_s \langle |\Sigma|^e \rangle$), from cases (4), (7), and Lemma 4.3.14.30, we finally obtain: $\underline{\theta_s \langle \Gamma \rangle}; \underline{\theta_s \langle \Sigma \rangle} \vdash e : \underline{\theta_s \langle \tau \rangle}$.

LEMMA: Altering Shallow Mutability.

If $\{\bar{\theta}\} \subseteq \text{ftv}(\tau)$ and $\theta \models_{\text{sol}} \tau$, then $\theta \models_{\text{sol}} \tau[\bar{\theta}/\bar{\theta}]$ and $\theta \models_{\text{sol}} \tau[\bar{\theta}/\bar{\theta}]$.

Proof: Since a substitution is performed shallowly (up to the reference boundary), it cannot violate copy compatibility (copy coercion) constraints. Since all type variables are still as general as before, and no new variables are used, θ is still a solution for all constraints in the modified type. Note that we can end up with types of the form $\llbracket \tau \rrbracket$ (or $\lceil \tau \rceil$), which is equivalent to $\llbracket \tau \rrbracket$ (or $\lceil \tau \rceil$).

LEMMA: Corollary to Lemma 4.3.14.46.

If $\{\bar{\theta}\} \subseteq \text{ftv}(\tau)$ and $\theta \models_{\text{sol}} \omega_f + \tau + \omega_s$, then $\theta \models_{\text{sol}} \omega_f + \tau[\bar{\theta}/\bar{\theta}] + \omega_s$ and $\theta \models_{\text{sol}} \omega_f + \tau[\bar{\theta}/\bar{\theta}] + \omega_s$.

Proof: Straightforward extension to Lemma 4.3.14.47

LEMMA: Corollary to Lemma 4.3.14.47.

If $\{\bar{\theta}\} \subseteq \text{ftv}(\tau)$ and $\theta_s \models_{\text{sol}} \theta \langle \omega_f + \tau + \omega_s \rangle$, then $\theta_s \models_{\text{sol}} \theta \langle \omega_f + \tau[\bar{\theta}/\bar{\theta}] + \omega_s \rangle$ and $\theta_s \models_{\text{sol}} \theta \langle \omega_f + \tau[\bar{\theta}/\bar{\theta}] + \omega_s \rangle$.

Proof: Straightforward extension to Lemma 4.3.14.47

LEMMA: Valid Substitutions Preserve Declarative Derivation.

If

1. $\theta_1 \models_{\text{sol}} \tau + \Gamma + \Sigma$
2. $\theta_1 \langle \Gamma \rangle; \theta_1 \langle \Sigma \rangle \vdash e : \theta_1 \langle \tau \rangle$
3. $\theta_2 \models_{\text{sol}} \theta \langle \tau + \Gamma + \Sigma \rangle$ for some substitution θ
4. $\text{dom}(\theta_1) \cap \text{dom}(\theta_2 \circ \theta) = \text{MTVS}(\tau + \Gamma + \Sigma)$

Then, $\theta_2 \circ \theta \langle \Gamma \rangle; \theta_2 \circ \theta \langle \Sigma \rangle \vdash e : \theta_2 \circ \theta \langle \tau \rangle$

Proof:

1. From Lemma 4.3.14.32, assumption (1), (3), and (4), we conclude that $\exists \theta'_1, \theta_0$ such that

- (a) $\theta_1 \cong \theta'_1$
- (b) $\tau + \Gamma + \Sigma \vdash_{\text{eqt}} \theta_2 \circ \theta \approx \theta'_1 \circ \theta_0$

2. From assumption (1), (2), case (1.a) above, and Lemma 4.3.14.43, we have: $\theta'_1 \langle \Gamma \rangle; \theta'_1 \langle \Sigma \rangle \vdash e : \theta'_1 \langle \tau \rangle$

3. From assumption (3) and Lemma 4.3.14.21, we have $\mathbf{N}(\theta_2 \circ \theta \langle \tau + \Gamma + \Sigma \rangle) = \theta_2 \circ \theta \langle \tau + \Gamma + \Sigma \rangle$

4. From cases (3), (1.b), and Lemma 4.3.14.22, we have $\mathbf{N}(\theta'_1 \circ \theta_0 \langle \tau + \Gamma + \Sigma \rangle) = \theta'_1 \circ \theta_0 \langle \tau + \Gamma + \Sigma \rangle$. That is, $\mathbf{N}(\theta_0 \langle \theta'_1 \langle \tau + \Gamma + \Sigma \rangle \rangle) = \theta_0 \langle \theta'_1 \langle \tau + \Gamma + \Sigma \rangle \rangle$.

5. From cases (2), (4), and Lemma 4.3.14.39, we have: $\theta_0 \langle \theta'_1 \langle \Gamma \rangle \rangle; \theta_0 \langle \theta'_1 \langle \Sigma \rangle \rangle \vdash e : \theta_0 \langle \theta'_1 \langle \tau \rangle \rangle$. That is, $\theta'_1 \circ \theta_0 \langle \Gamma \rangle; \theta'_1 \circ \theta_0 \langle \Sigma \rangle \vdash e : \theta'_1 \circ \theta_0 \langle \tau \rangle$.

6. Finally, from cases (1.b), (4), and Lemma 4.3.14.42, we have: $\theta_2 \circ \theta \langle \Gamma \rangle; \theta_2 \circ \theta \langle \Sigma \rangle \vdash e : \theta_2 \circ \theta \langle \tau \rangle$.

THEOREM: Soundness of Eager Inference

If:

1. $\Gamma; \Sigma; \Pi \vdash_i e : \tau \parallel \theta_u; \Pi'$
2. $\theta_s \vdash_{sol} \tau + \theta_u \langle \Gamma \rangle + \theta_u \langle \Sigma \rangle$.
3. $\theta_{su} = \theta_s \circ \theta_u$

Then, $\theta_{su} \langle \Gamma \rangle; \theta_{su} \langle \Sigma \rangle \vdash e : \theta_s \langle \tau \rangle$.**Proof:** By induction on the derivation of $\Gamma; \Sigma; \Pi \vdash_i e : \tau \parallel \theta_1; \Pi'$. We proceed by case analysis on the last step (again assuming α -reduction vacuously):

1. Cases TI-UNIT, TI-TRUE, TI-FALSE, TI-ID, TI-HLOC, TI-SLOC are trivial.
2. Case TI-LAMBDA:

(a) We know that:

- i.
$$\frac{\Gamma, x \mapsto \alpha; \Sigma; \Pi_\alpha \vdash_i e_i : \tau_r \parallel \theta_u; \Pi'}{\Gamma; \Sigma; \Pi \vdash_i \lambda x. e_i : [\theta_u \langle \alpha \rangle] \rightarrow [\tau_r]} \parallel \theta_u; \Pi'$$
- ii. $\theta_s \vdash_{sol} [\theta_u \langle \alpha \rangle] \rightarrow [\tau_r] + \theta_u \langle \Gamma \rangle + \theta_u \langle \Sigma \rangle$

(b) From induction hypothesis, we have:

- i. If $\theta_{si} \vdash_{sol} \tau_r + \theta_u \langle \Gamma, x : \alpha \rangle + \theta_u \langle \Sigma \rangle$
- ii. Then, $\theta_{si} \circ \theta_u \langle \Gamma, x : \alpha \rangle; \theta_{si} \circ \theta_u \langle \Sigma \rangle \vdash e_i : \theta_{si} \langle \tau_r \rangle$
- iii. That is, $\theta_{si} \langle \theta_u \langle \Gamma, x : \alpha \rangle \rangle; \theta_{si} \langle \theta_u \langle \Sigma \rangle \rangle \vdash e_i : \theta_{si} \langle \tau_r \rangle$

(c) From case (2.b.i) above, and Definition 4.3.14.11 rule S-GAMMA, we have:
 $\theta_{si} \vdash_{sol} \tau_r + \theta_u \langle \Gamma \rangle + \theta_u \langle \alpha \rangle + \theta_u \langle \Sigma \rangle$ (d) From case (2.c) above and Lemma 4.3.14.28 (commutativity), we conclude that $\exists \theta'_{si} \cong \theta_{si}$ such that
 $\theta'_{si} \vdash_{sol} \tau_r + \theta_u \langle \alpha \rangle + \theta_u \langle \Gamma \rangle + \theta_u \langle \Sigma \rangle$.(e) From Definition 4.3.14.11 rule S-MULTIPLE, we can write: $\theta'_{si} = \theta_f \circ \theta_o$ where $\theta_f \vdash_{sol} \tau_r + \theta_u \langle \alpha \rangle$ and
 $\theta_o \vdash_{sol} \theta_f \langle \theta_u \langle \Gamma \rangle + \theta_u \langle \Sigma \rangle \rangle$.(f) Given $\theta_f \vdash_{sol} \tau_r + \theta_u \langle \alpha \rangle$, using the SOL-FN rule, we can conclude that $\theta_f \vdash_{sol} [\theta_u \langle \alpha \rangle] \rightarrow [\tau_r]$.(g) Therefore, $\theta'_{si} \vdash_{sol} [\theta_u \langle \alpha \rangle] \rightarrow [\tau_r] + \theta_u \langle \Gamma \rangle + \theta_u \langle \Sigma \rangle$. That is, $\theta'_{si} = \theta_s$ (h) From cases (d) and (g), we conclude that $\theta_{si} \cong \theta_s$.(i) From case (2.b.i) and Theorem 4.3.14.18, we have: $\theta_{si} \vdash_{sol} \tau_r + \theta_u \langle \Gamma, x : \alpha \rangle + \theta_u \langle \Sigma \rangle$.(j) From cases (2.i), (2.b.iii), (2.h), and Lemma 4.3.14.43, we have:
 $\theta_s \langle \theta_u \langle \Gamma, x : \alpha \rangle \rangle; \theta_s \langle \theta_u \langle \Sigma \rangle \rangle \vdash e_i : \theta_s \langle \tau_r \rangle$ (k) Since $\theta_{su} = \theta_s \circ \theta_u$, we can write $\theta_{su} \langle \Gamma, x : \alpha \rangle; \theta_{su} \langle \Sigma \rangle \vdash e_i : \theta_s \langle \tau_r \rangle$. That is,
 $\theta_{su} \langle \Gamma, x : \theta_{su} \langle \alpha \rangle \rangle; \theta_{su} \langle \Sigma \rangle \vdash e_i : \theta_s \langle \tau_r \rangle$. Or, $\theta_{su} \langle \Gamma \rangle, x : \theta_{su} \langle \alpha \rangle; \theta_{su} \langle \Sigma \rangle \vdash e_i : \theta_s \langle \tau_r \rangle$.(l) From the T-LAMBDA rule, we have: $\theta_{su} \langle \Gamma \rangle; \theta_{su} \langle \Sigma \rangle \vdash \lambda x. e_i : \nabla (\theta_{su} \langle \alpha \rangle) \rightarrow \Delta (\theta_s \langle \tau_r \rangle)$ (m) By definition, $\nabla (\theta_{su} \langle \alpha \rangle) \rightarrow \Delta (\theta_{su} \langle \tau_r \rangle) = \theta_{su} \langle \alpha \rangle \rightarrow [\theta_s \langle \tau_r \rangle] = \theta_{su} \langle \alpha \rangle \rightarrow [\theta_s \langle \tau_r \rangle] = \theta_s \langle \theta_u \langle \alpha \rangle \rangle \rightarrow [\tau_r]$ (n) From cases (2.j) and (2.k), we finally have: $\theta_{su} \langle \Gamma \rangle; \theta_{su} \langle \Sigma \rangle \vdash \lambda x. e_i : \theta_s \langle \theta_u \langle \alpha \rangle \rangle \rightarrow [\tau_r]$

3. Case TI-APP:

(a) We know that:

$$\begin{array}{l}
\text{[I]} \quad \Gamma; \Sigma; \Pi_{\alpha\beta\gamma\delta\varepsilon} \vdash e_f : \tau_f \parallel \theta_{uf}; \Pi' \\
\text{[II]} \quad \theta_{uf} \langle \Gamma \rangle; \theta_{uf} \langle \Sigma \rangle; \Pi' \vdash e_a : \tau_a \parallel \theta_{ua}; \Pi'' \\
\text{[III]} \quad \theta_{vf} \vdash_{w_f} \theta_{ua} \langle \tau_f \rangle = \beta \downarrow \langle [\delta] \rightarrow [\alpha] \rangle \\
\text{[IV]} \quad \bar{\theta} = \text{ftv}(\theta_{vf} \langle \alpha \rangle) \setminus \text{ftv}(\theta_{vf} \langle \delta \rangle) \\
\text{[V]} \quad \tau_{rh} = \nabla(\theta_{vf} \langle \alpha \rangle) \uparrow [\bar{\theta}] / \bar{\theta} \\
\text{[VI]} \quad \theta_{va} \vdash_{w_f} \tau_a = \gamma \downarrow \langle \theta_{vf} \langle \delta \rangle \rangle \\
\text{[VII]} \quad \tau_{res} = \varepsilon \downarrow \theta_{va} \langle \tau_{rh} \rangle \\
\hline
\Gamma; \Sigma; \Pi \vdash e_f e_a : \tau_{res} \parallel \theta_{va} \circ \theta_{vf} \circ \theta_{ua} \circ \theta_{uf}; \Pi''
\end{array}$$

Note that this inference rule is the same as the one in Table 4.9, in which all substitutions are written in every step (without abbreviation), and some redundant substitutions removed.

- i.
 - ii. $\theta_u = \theta_{va} \circ \theta_{vf} \circ \theta_{ua} \circ \theta_{uf}$
 - iii. $\theta_s \vdash_{sol} \tau_{res} + \theta_u \langle \Gamma \rangle + \theta_u \langle \Sigma \rangle$
 - iv. $\theta_{su} = \theta_s \circ \theta_u$
 - v. $e = e_f e_a$
- (b) From induction hypothesis (wrt [I]), we have:
 - i. If $\theta_{sf} \vdash_{sol} \tau_f + \theta_{uf} \langle \Gamma \rangle + \theta_{uf} \langle \Sigma \rangle$
 - ii. Then, $\theta_{sf} \circ \theta_{uf} \langle \Gamma \rangle; \theta_{sf} \circ \theta_{uf} \langle \Sigma \rangle \vdash e_f : \theta_{sf} \langle \tau_f \rangle$
- (c) Similarly, from induction hypothesis (wrt [II]), we have:
 - i. If $\theta_{sa} \vdash_{sol} \tau_a + \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle + \theta_{ua} \circ \theta_{uf} \langle \Sigma \rangle$
 - ii. Then, $\theta_{sa} \circ \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle; \theta_{sa} \circ \theta_{ua} \circ \theta_{uf} \langle \Sigma \rangle \vdash e_a : \theta_{sa} \langle \tau_a \rangle$
- (d) From cases (3.b.i), (3.c.i) and Theorem 4.3.14.18, we have:
 - i. $\theta_{sf} \vDash_{sol} \tau_f + \theta_{uf} \langle \Gamma \rangle + \theta_{uf} \langle \Sigma \rangle$
 - ii. $\theta_{sa} \vDash_{sol} \tau_a + \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle + \theta_{ua} \circ \theta_{uf} \langle \Sigma \rangle$
- (e)
 - i. From [I], [II], and Lemma 4.3.14.38, we have: $\vDash_{cst} \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle + \theta_{ua} \circ \theta_{uf} \langle |\Sigma|^{e_f, e_a} \rangle + \theta_{ua} \langle \tau_f \rangle + \tau_a$.
 - ii. We know that $|\Sigma|^{e_f e_a} = |\Sigma|^{e_f, e_a}$, and thus from case (3.a.v), $|\Sigma|^e = |\Sigma|^{e_f, e_a}$.
 - iii. From cases (3.e.i) and (3.e.ii), we obtain: $\vDash_{cst} \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle + \theta_{ua} \circ \theta_{uf} \langle |\Sigma|^e \rangle + \theta_{ua} \langle \tau_f \rangle + \tau_a$.
 - iv. From case (3.e.iii) and Lemma 4.3.14.23, we obtain:
$$\vDash_{consistent} \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle + \theta_{ua} \circ \theta_{uf} \langle |\Sigma|^e \rangle + \theta_{ua} \langle \tau_f \rangle + \tau_a.$$
- (f) From case (3.e.iv) and Lemma 4.3.14.21, we conclude that $\exists \theta_e$ such that $\theta_e \vDash_{sol} \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle + \theta_{ua} \circ \theta_{uf} \langle |\Sigma|^e \rangle + \theta_{ua} \langle \tau_f \rangle + \tau_a$.
- (g)
 - i. We know that $|\Sigma|^{e_f} \subseteq |\Sigma|^e$. Therefore, from cases (3.e.iii), (3.f), and Lemma 4.3.14.29 (weakening), we have: $\theta_e \vDash_{sol} \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle + \theta_{ua} \circ \theta_{uf} \langle |\Sigma|^{e_f} \rangle + \theta_{ua} \langle \tau_f \rangle$.
That is, $\theta_e \vDash_{sol} \theta_{ua} \langle \theta_{uf} \langle \Gamma \rangle \rangle + \theta_{ua} \langle \theta_{uf} \langle |\Sigma|^{e_f} \rangle \rangle + \theta_{ua} \langle \theta_{uf} \langle \tau_f \rangle \rangle$. Note that $\theta_{uf} \langle \tau_f \rangle = \tau_f$
 - ii. From (3.b.i), (3.d.i), and Lemma 4.3.14.41, we can conclude that $\exists \theta'_{sf}$ such that:
 - A. $\theta'_{sf} \vDash_{sol} \theta_{uf} \langle \Gamma \rangle + \theta_{uf} \langle |\Sigma|^{e_f} \rangle + \theta_{uf} \langle \tau_f \rangle$
 - B. $\theta'_{sf} \langle \theta_{uf} \langle \Gamma \rangle \rangle; \theta'_{sf} \langle \theta_{uf} \langle |\Sigma|^{e_f} \rangle \rangle \vdash e_f : \theta'_{sf} \langle \theta_{uf} \langle \tau_f \rangle \rangle$.
 - iii. It is clear that $\text{dom}(\theta'_{sf}) \cap \text{dom}(\theta_e \circ \theta_{ua}) = \text{MTVS}(\theta_{uf} \langle \Gamma \rangle + \theta_{uf} \langle |\Sigma|^{e_f} \rangle + \theta_{uf} \langle \tau_f \rangle)$, because the derivation [II] occurs in an environment that already contains the substitutions θ_{sf} (and thus θ'_{sf}), and it works on a different tape Π' , and thus uses fresh type variables when new type variables are introduced.
- (h) From cases (3.g.ii.A), (3.g.ii.B), (3.g.i), (3.g.iii), and Lemma 4.3.14.49, we have:
$$\theta_e \circ \theta_{ua} \langle \theta_{uf} \langle \Gamma \rangle \rangle; \theta_e \circ \theta_{ua} \langle \theta_{uf} \langle |\Sigma|^{e_f} \rangle \rangle \vdash e_f : \theta_e \circ \theta_{ua} \langle \theta_{uf} \langle \tau_f \rangle \rangle.$$
That is, $\theta_e \circ \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle; \theta_e \circ \theta_{ua} \circ \theta_{uf} \langle |\Sigma|^{e_f} \rangle \vdash e_f : \theta_e \circ \theta_{ua} \circ \theta_{uf} \langle \tau_f \rangle$.
- (i) Since $|\Sigma|^e \supseteq |\Sigma|^{e_f}$, we have $\theta_e \circ \theta_{ua} \circ \theta_{uf} \langle |\Sigma|^e \rangle \supseteq \theta_e \circ \theta_{ua} \circ \theta_{uf} \langle |\Sigma|^{e_f} \rangle$. From case (f) and Lemma 4.3.14.21 and Lemma 4.3.14.29, we can obtain $\mathbf{N}(\theta_e \circ \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle) = \theta_e \circ \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle$ and $\mathbf{N}(\theta_e \circ \theta_{ua} \circ \theta_{uf} \langle |\Sigma|^e \rangle) = \theta_e \circ \theta_{ua} \circ \theta_{uf} \langle |\Sigma|^e \rangle$. Now, from case (h) and Lemma 4.3.14.30, we have
$$\theta_e \circ \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle; \theta_e \circ \theta_{ua} \circ \theta_{uf} \langle |\Sigma|^e \rangle \vdash e_f : \theta_e \circ \theta_{ua} \circ \theta_{uf} \langle \tau_f \rangle.$$

- (j) Similarly, we have $\theta_e \circ \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle; \theta_e \circ \theta_{ua} \circ \theta_{uf} \langle |\Sigma|^e \rangle \vdash e_a : \theta_e \circ \theta_{ua} \circ \theta_{uf} \langle \tau_a \rangle$.
- (k) i. It is evident that $\models_{cst} \beta \downarrow ([\delta] \rightarrow [\alpha])$
 ii. Since α , β , and δ are fresh type variables, we can write (using case (3.e.iii)):
 $\models_{cst} \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle + \theta_{ua} \circ \theta_{uf} \langle |\Sigma|^e \rangle + \theta_{ua} \langle \tau_f \rangle + \tau_a + \beta \downarrow ([\delta] \rightarrow [\alpha])$.
 iii. Due to Lemma 4.3.14.28 (commutativity) and case (3.k.ii), we can write:
 $\models_{cst} \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle + \theta_{ua} \circ \theta_{uf} \langle |\Sigma|^e \rangle + \tau_a + \theta_{ua} \langle \tau_f \rangle + \beta \downarrow ([\delta] \rightarrow [\alpha])$.
 iv. From case (3.k.iii), [III], and Lemma 4.3.14.34, we have:
 $\models_{cst} \theta_{vf} \circ \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle + \theta_{vf} \circ \theta_{ua} \circ \theta_{uf} \langle |\Sigma|^e \rangle + \tau_a + \theta_{vf} \langle \theta_{ua} \rangle + \theta_{vf} \langle \beta \downarrow ([\delta] \rightarrow [\alpha]) \rangle$.
- (l) Again, due to case (3.k.iv) and Lemma 4.3.14.23, and Lemma 4.3.14.21, we conclude that $\exists \theta'_e$ such that
 $\theta'_e \models_{sol} \theta_{vf} \circ \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle + \theta_{vf} \circ \theta_{ua} \circ \theta_{uf} \langle |\Sigma|^e \rangle + \tau_a + \theta_{vf} \circ \theta_{ua} \langle \tau_f \rangle + \theta_{vf} \langle \beta \downarrow ([\delta] \rightarrow [\alpha]) \rangle$.
 Similar to the argument in cases (g) through (j), we obtain:
 i. $\theta'_e \circ \theta_{vf} \circ \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle; \theta'_e \circ \theta_{vf} \circ \theta_{ua} \circ \theta_{uf} \langle |\Sigma|^e \rangle \vdash e_f : \theta'_e \circ \theta_{vf} \circ \theta_{ua} \circ \theta_{uf} \langle \tau_f \rangle$.
 ii. $\theta'_e \circ \theta_{vf} \circ \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle; \theta'_e \circ \theta_{vf} \circ \theta_{ua} \circ \theta_{uf} \langle |\Sigma|^e \rangle \vdash e_a : \theta'_e \circ \theta_{vf} \circ \theta_{ua} \circ \theta_{uf} \langle \tau_a \rangle$.
- (m) Since $\theta_{vf} \circ \theta_{ua} \circ \theta_{uf} \langle \tau_a \rangle = \tau_a$ and $\theta_u = \theta_{va} \circ \theta_{vf} \circ \theta_{ua} \circ \theta_{uf}$, similar to cases (k) and (j) above, for the unification performed in step [VI], we conclude that $\exists \theta''_e$ such that:
 i. Let
 A. $\omega_1 = \theta_{va} \circ \theta_{vf} \circ \theta_{ua} \circ \theta_{uf} \langle \Gamma \rangle$
 B. $\omega_2 = \theta_{va} \circ \theta_{vf} \circ \theta_{ua} \circ \theta_{uf} \langle |\Sigma|^e \rangle$
 C. $\omega_3 = \theta_{va} \circ \theta_{vf} \circ \theta_{ua} \langle \tau_f \rangle$
 D. $\omega_4 = \theta_{va} \langle \tau_a \rangle$
 E. $\omega_5 = \theta_{va} \circ \theta_{vf} \langle \beta \downarrow ([\delta] \rightarrow [\alpha]) \rangle$
 F. $\omega_6 = \theta_{va} \langle \gamma \downarrow [\theta_{vf} \langle \delta \rangle] \rangle$
 ii. $\models_{cst} \omega_1 + \omega_2 + \omega_3 + \omega_4 + \omega_5 + \omega_6$.
 iii. $\theta''_e \models_{sol} \omega_1 + \omega_2 + \omega_3 + \omega_4 + \omega_5 + \omega_6$.
 iv. $\theta''_e \circ \theta_u \langle \Gamma \rangle; \theta''_e \circ \theta_u \langle |\Sigma|^e \rangle \vdash e_f : \theta''_e \circ \theta_u \langle \tau_f \rangle$.
 v. $\theta''_e \circ \theta_u \langle \Gamma \rangle; \theta''_e \circ \theta_u \langle |\Sigma|^e \rangle \vdash e_a : \theta''_e \circ \theta_u \langle \tau_a \rangle$.
- (n) i. From [III], we have: $\theta_{vf} \vdash_{unf} \theta_{ua} \langle \tau_f \rangle = \beta \downarrow ([\delta] \rightarrow [\alpha])$
 ii. From cases (3.m.ii), (3.m.iii) and Lemma 4.3.14.29 (weakening), we have: $\theta''_e \models_{sol} \omega_3 + \omega_5 \langle \rangle$. That is,
 $\theta''_e \models_{sol} \theta_{va} \circ \theta_{vf} \circ \theta_{ua} \langle \tau_f \rangle + \theta_{va} \circ \theta_{vf} \langle \beta \downarrow ([\delta] \rightarrow [\alpha]) \rangle \langle \rangle$.
 iii. The case (3.n.ii) can be re-written as: $\theta''_e \models_{sol} \theta_{va} \circ \theta_{vf} \langle \theta_{ua} \langle \tau_f \rangle \rangle + \theta_{va} \circ \theta_{vf} \langle \beta \downarrow ([\delta] \rightarrow [\alpha]) \rangle$.
 iv. From cases (3.n.i), (3.n.iii) and Lemma 4.3.14.36, we have:
 $\frac{\theta''_e \circ \theta_{va} \circ \theta_{vf} \langle \theta_{ua} \langle \tau_f \rangle \rangle \preceq \nabla (\theta''_e \circ \theta_{va} \circ \theta_{vf} \langle |\delta| \rightarrow [\alpha] \rangle)}{\text{That is, } \theta''_e \circ \theta_{va} \circ \theta_{vf} \circ \theta_{ua} \langle \tau_f \rangle \preceq \theta''_e \circ \theta_{va} \circ \theta_{vf} \langle |\delta| \rightarrow [\alpha] \rangle}$.
 v. The case (3.n.iv) can be re-written as:
 $\frac{\theta''_e \circ \theta_{va} \circ \theta_{vf} \circ \theta_{ua} \circ \theta_{uf} \langle \tau_f \rangle \preceq \theta''_e \circ \theta_{va} \circ \theta_{vf} \circ \theta_{ua} \circ \theta_{uf} \langle |\delta| \rightarrow [\alpha] \rangle}{\text{since these substitutions have no effect.}}$
 vi. Due to case (3.a.ii), the above case (3.n.v) is equivalent to $\frac{\theta''_e \circ \theta_u \langle \tau_f \rangle \preceq \theta''_e \circ \theta_u \langle |\delta| \rightarrow [\alpha] \rangle}$
 vii. The case (3.n.vi) is further equivalent to $\frac{\theta''_e \circ \theta_u \langle \tau_f \rangle \preceq \theta''_e \circ \theta_u \langle \delta \rangle}{\rightarrow [\theta''_e \circ \theta_u \langle \alpha \rangle]}$.
 viii. Similarly, from the unification performed in step [VI], we have: $\frac{\theta''_e \circ \theta_u \langle \tau_a \rangle \preceq \theta''_e \circ \theta_u \langle \delta \rangle}{\rightarrow [\theta''_e \circ \theta_u \langle \alpha \rangle]}$.
- (o) i. From cases (3.m.iv) and (3.n.vii) we obtain:
 $\frac{\theta''_e \circ \theta_u \langle \Gamma \rangle; \theta''_e \circ \theta_u \langle |\Sigma|^e \rangle \vdash e_f \preceq \theta''_e \circ \theta_u \langle \delta \rangle}{\rightarrow [\theta''_e \circ \theta_u \langle \alpha \rangle]}$.
 ii. From cases (3.m.v) and (3.n.viii) we obtain:
 $\frac{\theta''_e \circ \theta_u \langle \Gamma \rangle; \theta''_e \circ \theta_u \langle |\Sigma|^e \rangle \vdash e_a \preceq \theta''_e \circ \theta_u \langle \delta \rangle}{\rightarrow [\theta''_e \circ \theta_u \langle \alpha \rangle]}$.
- (p) i. From (3.m.ii) and (3.m.iii) and Lemma 4.3.14.29, we have $\theta''_e \models_{sol} \omega_1 + \omega_2 + \theta_{va} \circ \theta_{vf} \langle \alpha \rangle$.
 ii. The case (3.p.i) can be re-written as: $\theta''_e \models_{sol} \theta_{va} \langle \omega_1 + \omega_2 + \theta_{vf} \langle \alpha \rangle \rangle$. (Note: ω_1 and ω_2 already contain substitutions from θ_{va}).

- iii. From [IV], we have $\{\bar{\theta}\} \subseteq \text{ftv}(\theta_{vf}\langle\alpha\rangle)$
- iv. From cases (3.p.ii), (3.p.iii) and Lemma 4.3.14.48, we have: $\theta''_e \models_{\text{sol}} \theta_{va}\langle\omega_1 + \omega_2 + \theta_{vf}\langle\alpha\rangle[\bar{\theta}/\bar{\theta}]\rangle$. (Note: Here, we instantiated ω_s of Lemma 4.3.14.48 to \emptyset).
- v. From (3.m.iv), it is evident that: $\theta''_e \models_{\text{sol}} \omega_1 + \omega_2 + \theta_{va}\langle\nabla(\theta_{vf}\langle\alpha\rangle)[\bar{\theta}/\bar{\theta}]\rangle$. That is, $\theta''_e \models_{\text{sol}} \omega_1 + \omega_2 + \theta_{va}\langle\tau_{rh}\rangle$.
- (q) i. Let $\theta_E = \theta''_e \circ [\varepsilon \mapsto \theta_{va}\langle\tau_{rh}\rangle]$. A substitution for ε with any type $\tau \cong \theta_{va}\langle\tau_{rh}\rangle$ will actually work in this case.
- ii. From cases (3.p.v), (3.q.1) Lemma 4.3.14.21, and Definition 4.3.14.12, we can easily obtain: $\theta_E \models_{\text{sol}} \omega_1 + \omega_2 + \gamma\downarrow[\theta_{vf}\langle\delta\rangle]$. That is, $\theta_E \models_{\text{sol}} \omega_1 + \omega_2 + \tau_{ret}$.
- (r) i. Similar to case (3.p.i), we obtain $\theta''_e \models_{\text{sol}} \theta_{va}\langle\theta_{vf}\langle\alpha\rangle\rangle$
- ii. From case (3.r.i) and Lemma 4.3.14.21, we obtain: $\mathbf{N}(\theta''_e\langle\theta_{va}\langle\theta_{vf}\langle\alpha\rangle\rangle\rangle) = \theta''_e\langle\theta_{va}\langle\theta_{vf}\langle\alpha\rangle\rangle$. That is, $\mathbf{N}(\theta''_e \circ \theta_{va}\langle\theta_{vf}\langle\alpha\rangle\rangle) = \theta''_e \circ \theta_{va}\langle\theta_{vf}\langle\alpha\rangle\rangle$.
- iii. From cases (3.r.ii), (3.p.iii) and Lemma 4.3.14.24, we obtain: $\theta''_e \circ \theta_{va}\langle[\theta_{vf}\langle\alpha\rangle]\rangle \preceq \theta''_e \circ \theta_{va}\langle\nabla(\theta_{vf}\langle\alpha\rangle)[\bar{\theta}/\bar{\theta}]\rangle$.
- iv. It is evident that $\theta_{va} \circ \theta_{vf}\langle\alpha\rangle \models_{\text{eqi}} \theta''_e \approx \theta_E$. Therefore, we can also say $\theta_{va}\langle[\theta_{vf}\langle\alpha\rangle]\rangle \models_{\text{eqi}} \theta''_e \approx \theta_E$ and $\theta_{va}\langle\nabla(\theta_{vf}\langle\alpha\rangle)[\bar{\theta}/\bar{\theta}]\rangle \models_{\text{eqi}} \theta''_e \approx \theta_E$.
- v. From cases (3.r.iii) and (3.r.iv), we have: $\theta_E \circ \theta_{va}\langle[\theta_{vf}\langle\alpha\rangle]\rangle \preceq \theta_E \circ \theta_{va}\langle\nabla(\theta_{vf}\langle\alpha\rangle)[\bar{\theta}/\bar{\theta}]\rangle$. That is, $[\theta_E \circ \theta_{va} \circ \theta_{vf}\langle\alpha\rangle] \preceq \theta_E \circ \theta_{va}\langle\tau_{rh}\rangle$.
- vi. From case (3.q.i), we know that $\theta_E\langle\tau_{res}\rangle = \theta_E \circ \theta_{va}\langle\tau_{rh}\rangle$. (We actually only need a compatibility result here, in case we chose any other compatible type in case (3.q.i)).
- vii. From cases (3.r.v) and (3.v.vi), we obtain $[\theta_E \circ \theta_{va} \circ \theta_{vf}\langle\alpha\rangle] \preceq \theta_E\langle\tau_{res}\rangle$.
- viii. The case (3.r.vii) can be written as $[\theta_E \circ \theta_u\langle\alpha\rangle] \preceq \theta_E\langle\tau_{res}\rangle$, since the extra substitutions have no effect.
- (s) i. From case (3.q.i), it is evident that $\omega_1 + \omega_2 + [\theta''_e \circ \theta_u\langle\delta\rangle] \rightarrow [\theta''_e \circ \theta_u\langle\alpha\rangle] \models_{\text{eqi}} \theta_E \approx \theta''_e$. Therefore, from case (3.o.i) and Lemma 4.3.14.42, we have: $\theta_E \circ \theta_u\langle\Gamma\rangle; \theta_E \circ \theta_u\langle|\Sigma|^e\rangle \vdash e_f \preceq [\theta_E \circ \theta_u\langle\delta\rangle] \rightarrow [\theta_E \circ \theta_u\langle\alpha\rangle]$.
- ii. Similarly, we have: $\theta_E \circ \theta_u\langle\Gamma\rangle; \theta_E \circ \theta_u\langle|\Sigma|^e\rangle \vdash e_a \preceq [\theta_E \circ \theta_u\langle\delta\rangle]$.
- (t) Now, from cases (3.s.i), (3.s.ii), (3.r.viii) and T-APP rule, we obtain: $\theta_E \circ \theta_u\langle\Gamma\rangle; \theta_E \circ \theta_u\langle|\Sigma|^e\rangle \vdash e_f e_a : \theta_E\langle\tau_{res}\rangle$.
- (u) i. From case (3.q.ii) and Lemma 4.3.14.28 (commutativity), we obtain: $\theta_E \models_{\text{sol}} \tau_{ret} + \omega_1 + \omega_2$.
- ii. From case (3.a.iii) and Theorem 4.3.14.18, we have: $\theta_s \models_{\text{sol}} \tau_{res} + \theta_u\langle\Gamma\rangle + \theta_u\langle\Sigma\rangle$
- iii. From cases (3.u.i), (t), (3.u.ii), and Lemma 4.3.14.45, we obtain: $\theta_s \circ \theta_u\langle\Gamma\rangle; \theta_s \circ \theta_u\langle\Sigma\rangle \vdash e_f e_a : \theta_s\langle\tau_{res}\rangle$.
- iv. Finally, from cases (3.a.iv) and (3.a.v), we obtain the desired result: $\theta_{su}\langle\Gamma\rangle; \theta_{su}\langle\Sigma\rangle \vdash e : \theta_s\langle\tau_{res}\rangle$.

4. Cases TI-LET- α : Similar to case (4) (TI-APP), except for the use of Lemma 4.3.14.27.

5. Cases TI-IF, TI-DUP, TI-DEREF, TI-SET, TI-TQEXPR, TI-LET- α -TQ are similar.

LEMMA: Corollary-1 to Soundness of Eager Inference

If:

1. $\Gamma; \Sigma; \Pi \vdash_i e : \tau \parallel \theta_u; \Pi'$
2. $\theta_s \vdash_{\text{sol}} \tau : \gg \tau'$
3. $\theta_e \vdash_{\text{sol}} \theta_s \circ \theta_u\langle\Gamma + |\Sigma|^e\rangle$
4. $\theta = \theta_e \circ \theta_s \circ \theta_u$

Then $\theta\langle\Gamma\rangle; \theta\langle\Sigma\rangle \vdash e : \tau'$.

Proof: Follows from Theorem 4.3.14.50 Definition 4.3.14.4, and Lemma 4.3.14.37.

THEOREM: Corollary-2 to Soundness of Eager Inference

If:

1. $\Gamma; \Sigma; \Pi \vdash_{\tau} e : \tau \parallel \theta_u; \Pi'$
2. $\theta_s \vdash_{sol} \tau : \gg \tau'$

Then $\exists \theta_e$ such that

1. $\theta = \theta_e \circ \theta_s \circ \theta_u$
2. $\theta(\Gamma); \theta(\Sigma) \vdash e : \tau'$.

Proof: Follows from Lemma 4.3.14.51.

Chapter 5

Implementation

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 type system accounts for about 6,894 lines and the implementation of polymorphism accounts for 992 lines.

The bootstrap compiler for BitC implements polymorphism by brute-force polyinstantiation. This simplifies the implementation of overloading (type-classes) at the cost of requiring whole-program compilation. The algorithm is incremental, supporting use in an interactive environment [37]. More sophisticated techniques for implementing polymorphism over unboxed types are explored in the literature [24, 13, 32]. We view the current implementation as experimental, though it does have the practical advantage (important to us) that emitted types and code are directly inter-callable with C.

There are several proposals for implementing polymorphism over unboxed types. For example, by using coercions [24] into a boxed representation when used in a polymorphic context, using dictionaries [13] that is, passing extra type-parameters to functions, hybrid variations of the above [32] or full polyinstantiation (C++ templates). Depending on the option we pick, there are different trade-offs with respect to the amount of RTTI support needed, separate compilation, efficiency, code size, *etc.*

Chapter 6

Related Work

Cyclone [17] supports first class polymorphism and polymorphic recursion for functions definitions. This approach is feasible in Cyclone, where there is a distinction between functions (code) and function pointers (data). In an expression language with inner functions, it is more intuitive to treat all first class values alike. Cyclone supports partial type reconstruction so that so that most types and instantiations of polymorphic types can be automatically inferred. Grossman provides a detailed account of using quantified types with imperative C style mutation and `&` operator in Cyclone [53]. His formalization develops a general theory wherein any expression can be polymorphic, but 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. One contribution of our work (in comparison to [53]) is that we give a formal specification and proof of correctness of the inference algorithm, not just the type system.

Smith and Volpano have proposed an ML-style polymorphic type system for a dialect of C [33]. Their system uses different binding constructs for polymorphic and mutable bindings — `let`, `letvar`, `letarr`. They impose the ML-like restriction that all first class references, `vars` and `arrays` must be mutable, and function arguments and `let`-bound identifiers be immutable, because of which they do not have to deal with copy compatibility issues. Their language does not have structures and union types, and thus does not address complications due to parametrized types, and unboxed composite types with mixed unboxed mutable and immutable types. Our language, while being strictly more expressive, also provides a more natural expression of programs.

A monadic model [20] of mutable state is used in pure functional languages like Haskell [19]. The main advantage of this approach is that the type system provides guarantees not only about the immutability of locations but also distinguishes side effecting computations from others. Therefore, this model is well suited for integration with theorem provers and deduction systems. However, this model is also considerably different from the normal programming idioms used by systems programmers. Hallgren *et al.* have recently proposed a monadic interface to low level hardware and formally specified certain behavioral properties about it [12]. They have also implemented a prototype operating systems in Haskell based on this system, and it would be interesting to see if this approach scales to a full implementation that provides real time guarantees. In BitC, we have considered providing syntactic constructs that guarantee side-effect free computation. For example we could have a defining form `defpure` that is similar to `define`, but allows purely applicative definitions only. This has the advantage of providing a separation of stateful computation from pure ones, as well as the simplicity of staying within the Hindley-Milner type system.

Diatchki *et al* have proposed support for bit-level word types in Haskell [8]. Their solution could be extended to the full `defrepr` mechanism of BitC. Communication with the authors has revealed that there is a proposal for extending their prototype interpreter into a full implementation in the GHC compiler [10]. The VFiasco project aims at formalizing the semantics of C++ and using it directly for verification of the Fiasco microkernel written in C++. They present formal semantics for some datatypes of C++ in [14], but do not model C++ pointers, unions or mutability.

Cqual [9] provides a general framework for inference and use of type qualifiers. One of the applications presented in [9] is inference of (maximal) `const` qualifications for types in C programs, similar to mutability inference in BitC. However, their system does not deal with polymorphism or parametrized composite datatypes. SysObjC [3] extends

the C programming language with object-like value types, but does address type safety in the face of polymorphism.

C# is a safe, high-level language supports mutability and low-level representation, but does not support type inference. Spec# [4] is an extension of C# that also provides an integrated verification framework. Their framework is complementary to our system, and can benefit from BitC's mutability model [34].

Chapter 7

Conclusion

In this paper, we have proposed a well-founded first-class mutability model. It is well founded in the sense that types are definitive about the mutability of all locations, and every location has one and only one type across all aliases. The model is first class in the sense that it supports unboxed objects and mutability of stack variables. This makes a language with this type system suitable for systems programming as well as for integration with a verification framework.

There is a fundamental conflict of goals between the ability to infer principal types and to allow freedom of mutability-compatibility at copy boundaries. We have identified various trade-offs and some design choices in this regard, along with their pros and cons. We have proposed a solution to this problem that uses certain hinting mechanisms to infer types based on the “natural” flow of type information in an expression. We have also provided a formal framework and for our type system and proved it sound. The type system is implemented as part of the BitC language compiler. Source code for the BitC compiler can be obtained from <http://coyotos.org>.

Bibliography

- [1] Thomas Ball, Todd Millstein, Sriram K. Rajamani “Polymorphic predicate abstraction.” *Microsoft Research Technical Report MSR_2001_10* June 2002.
- [2] Thomas Ball and Sriram K. Rajamani. “The SLAM Project: Debugging System Software via Static Analysis.” *Proc. 2002 ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, 2002.
- [3] Ádám Balogh and Zoltán Csörnyei. “SysObjC: C Extension for Development of Object-Oriented Operating Systems.” *Proc. Third ECOOP Workshop on Programming Languages and Operating Systems*. San Jose, CA. October 2006.
- [4] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. “The Spec# programming system: An overview.” *CASIS 2004, LNCS vol. 3362* 2004.
- [5] E. Brewer, J. Condit, B. McCloskey, and F. Zhou. “Thirty Years is Long Enough: Getting Beyond C.” *Proc. Tenth Workshop on Hot Topics in Operating System (HotOS X)*, USENIX, 2005.
- [6] The Coq Development Team “The Coq Proof Assistant Reference Manual” <http://coq.inria.fr/doc/main.html>
- [7] R. Deline and M. Fahndrich “Enforcing high-level protocols in low-level software.” *Proc. of the ACM Conference on Programming Language Design and Implementation* pp 5969. 2001.
- [8] Iavor S. Diatchki, Mark P. Jones, and Rebekah Leslie. “High-level Views on Low-level Representations.” *Proc. 10th ACM Conference on Functional Programming* pp. 168–179. September 2005.
- [9] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. “A Theory of Type Qualifiers.” *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI’99)*. pp. 192–203. 1999.
- [10] The GHC Team “The Glorious Glasgow Haskell Compilation System User’s Guide, Version 6.6” http://www.haskell.org/ghc/docs/latest/html/users_guide/index.html
- [11] D. Grossman, “Quantified Types in an Imperative Language” *ACM Transactions on Programming Languages and Systems* 2006.
- [12] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. “A Principled Approach to Operating System Construction in Haskell.” *Proc. International Conference on Functional Programming (ICFP’05)*, Sep. 2005. Tallinn, Estonia. pp. 116–128.
- [13] R. Harper and G. Morrisett. “Compiling polymorphism using intentional type analysis.” *ACM Symp. on Principles of Programming Languages* pp 130-141, January 1995
- [14] Michael Hohmuth and Hendrik Tews “The VFiasco approach for a verified operating system.” *ECOOP Workshop on Programming Languages and Operating Systems* 2005.
- [15] ISO, *International Standard ISO/IEC 8652:1995 (Information Technology — Programming Languages — Ada)* International Standards Organization (ISO). 1995.

- [16] ISO, *International Standard ISO/IEC 9899:1999 (Programming Languages - C)* International Standards Organization (ISO). 1999.
- [17] 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 275-288, 2002.
- [18] Mark P. Jones “Qualified types: theory and practice.” *Cambridge Distinguished Dissertations In Computer Science* ISBN:0-521-47253-9, 1995
- [19] Simon Peyton Jones (ed.). *Haskell 98 Language and Libraries: The Revised report*. Cambridge University Press. 2003.
- [20] Launchbury, J. and Peyton Jones, S. L. “State in Haskell.” *LISP and Symbolic Computation* **8**, 4 (Dec.), pp 293-341, 1995.
- [21] M. Kaufmann, J. S. Moore. “Computer Aided Reasoning: An Approach” *Kluwer Academic Publishers*, 2000.
- [22] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988
- [23] Xavier Leroy, “The Objective Caml System Release 3.09, Documentation and User’s Manual.” <http://caml.inria.fr/pub/docs/manual-ocaml/index.html>
- [24] X. Leroy, “Unboxed objects and polymorphic typing.” *ACM SIGPLAN Symposium on Principles of Programming Languages* pages 177–188, January 1992. **8**(4):343–355, 1995.
- [25] Robin Milner “A theory of type polymorphism in programming.” *Journal of Computer and System Sciences* pp 348-375, 1978.
- [26] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised* The MIT Press, May 1997.
- [27] G. Necula, S. McPeak, and W. Weimer “CCured: Type-safe retrofitting of legacy code.” *Proc. of Symposium on Principles of Programming Languages* pp 128-139, 2002.
- [28] T. Nipkow, L. C. Paulson and M. Wenzel “Isabelle/HOL — A Proof Assistant for Higher-Order Logic.” *Springer LNCS series volume 2283* 2002.
- [29] Frank Pfenning and Carsten Schürmann. “System description: Twelf — a meta-logical framework for deductive systems.” *Proc. of International Conference on Automated Deduction (CADE-16)* pp 202-206, Springer-Verlag LNAI 1632, 1999.
- [30] Benjamin C. Pierce “Types and Programming Languages” *The MIT Press, Massachusetts Institute of Technology* ISBN 0-262-16209-1, 2002.
- [31] Benjamin C. Pierce and David N. Turner. “Local Type Inference.” *In Proc. of Symposium on Principles of Programming Languages* pp 252-265, 1998.
- [32] Zhong Shao. “Flexible representation analysis.” *Proc. ACM SIGPLAN International conference on Functional programming* pp 85 - 98, 1997.
- [33] 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.
- [34] Spec# team “Spec# 1.0.6404 for Microsoft Visual Studio 2005 Release Notes” <http://research.microsoft.com/specsharp/1.0.6404/relnotes.htm>
- [35] Matthew S. Tschantz and Michael D. Ernst, “Javari: Adding reference immutability to Java” *Object-Oriented Programming Systems, Languages, and Applications* pp 211-230, October 2005.
- [36] A. K. Wright, “Simple Imperative Polymorphism” *Lisp and Symbolic Computation* **8**(4):343–355, 1995.

- [37] Swaroop Sridhar “Implementation of Polymorphism in BitC” <http://www.coyotos.org/docs/bitc/polyinst.html>
- [38] S. Sridhar and J. Shapiro. “Type Inference for Unboxed Types and First Class Mutability” *Proc. 3rd ECOOP Workshop on Programming Languages and Operating Systems (PLOS 2006)* San Jose, CA. 2006.
- [39] J. S. Shapiro, Eric Northup, M. Scott Doerrie, and Swaroop Sridhar. *Coyotos Microkernel Specification*, 2006, available online at www.coyotos.org.
- [40] **NOT USED** Tim Harris, Simon Marlow and Simon Peyton Jones “Haskell on a shared-memory multiprocessor” *Proc. of the 2005 ACM SIGPLAN workshop on Haskell*. pp 49-61, 2005.
- [41] **NOT USED** H. XI “Imperative programming with dependent types.” *Proc. of IEEE Symposium on Logic in Computer Science* pp 375387, 2000.
- [42] **NOT USED** J. S. Shapiro, J. M. Smith, and D. J. Farber. “EROS, A Fast Capability System” *Proc. 17th ACM Symposium on Operating Systems Principles*. Dec 1999. pp. 170–185. Kiawah Island Resort, SC, USA.
- [43] **NOT USED** M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Lauris, and S. Levi. “Language Support for Fast and Reliable Message-based Communication in Singularity OS.” *Proc. EUROSYS 2006*, Leuven Belgium. 2006
- [44] **NOT USED** M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. R. Lauris. “Deconstructing Process Isolation.” *Microsoft Technical Report MSR-TR-2006-43*. Microsoft, Inc. 2006
- [45] **NOT USED** M. Fähndrich and R. DeLine ”Adoption and Focus: Practical Linear Types for Imperative Programming.” *Proc. of the ACM Conference on Programming Language Design and Implementation* June 2002.
- [46] **NOT USED** Bernd Schoeller. “Strengthening Eiffel contracts using models.” *Hung Dang Van and Zhiming Liu, editors, Proceeding of the Workshop on Formal Aspects of Component Software FACS’03*, September 2003.
- [47] **NOT USED** Bart Jacobs, Jan Smans, Frank Piessens, Wolfram Schulte. “A Simple Sequential Reasoning Approach for Sound Modular Verification of Mainstream Multithreaded Programs” *Proc. of Multithreading in Hardware and Software: Formal Approaches to Design and Verification (TV 2006)*, Seattle, 2006.
- [48] **NOT USED** Geoffrey Smith and Dennis Volpano. “Polymorphic typing of variables and references” *ACM Transactions on Programming Languages and Systems* Pages: 254 - 267, May 1996
- [49] **NOT USED** Simon Peyton Jones and Philip Wadler “Imperative functional programming.” *Proc. ACM SIGPLAN Principles of Programming Languages*. 1993
- [50] A. K. Wright, “Simple Imperative Polymorphism” *Lisp and Symbolic Computation* 8(4):343–355, 1995.
- [51] Benjamin C. Pierce “Types and Programming Languages” *The MIT Press, Massachusetts Institute of Technology* ISBN 0-262-16209-1, 2002.
- [52] J. Garrigue, “Relaxing the Value Restriction” *International Symposium on Functional and Logic Programming* 2004.
- [53] D. Grossman, “Quantified Types in an Imperative Language” *ACM Transactions on Programming Languages and Systems* 2006.
- [54] D. Grossman “Safe programming at the C level of abstraction.” *Ph.D. dissertation. Cornell University* 2003.
- [55] J. S. Shapiro, S. Sridhar, M. S. Doerrie, “BitC Language Specification” <http://coyotos.org/docs/bitc/spec.html>
- [56] S. Sridhar, J. S. Shapiro “Design of Type and Mutability Inference in BitC” *Systems Research Laboratory Technical Report SRL2006-01* Johns Hopkins University, 2006. <http://www.coyotos.org/docs/bitc/mutinfer.html>
- [57] Jeff Vaughan “A proof of correctness for the Hindley-Milner type inference algorithm” <http://www.seas.upenn.edu/~vaughan2/docs/hmproof.pdf>