

Design of Type and Mutability Inference in BitC[†]

SRL Technical Report SRL2006-01

Swaroop Sridhar Jonathan Shapiro, Ph.D.
Systems Research Laboratory
Dept. of Computer Science
Johns Hopkins University

September 12, 2006

Abstract

BitC is a call-by-value language that supports unboxed mutability. This enables us to allow some freedom in the compatibility of types with respect to their mutability at copy boundaries – that is, the type of the copy of a value might differ in mutability with respect to the original value. However, this kind of compatibility interacts with type inference in surprising ways, because there is no longer a unique way to type an expression. This introduces a trade-off between the degree of freedom in type-compatibility vs. the amount of user annotations we require, as well as the amount of polymorphism we can preserve. This document explores these trade-offs and their implications as a matter of theory and practice.

[†] Copyright © 2006, Swaroop Sridhar and Jonathan S. Shapiro.

1 Introduction

Modern programming languages such as ML [8] or Haskell [9] provide newer, stronger, and more expressive type systems than traditional systems programming languages such as C[7, 5] or Ada [4]. These features improve the robustness and safety of programs, and it is highly desirable to incorporate them into languages that can be used for high-performance “systems” codes. The key missing features for this are state (mutability) and low-level representation. Languages such as C# support for state and representation, but do not support type inference, higher-order types, or mathematically well-founded mechanisms for abstraction.

While there have been recent efforts to integrate low-level representation features into Haskell [10], modern languages do not provide effective support for low-level representation management, or value types, and explicitly seek to discourage the use of “state” (mutable locations). As a result, these languages are ill-suited to the development of operating systems and high-performance “systems” codes. BitC is an attempt to unify all of these features into a single, consistent language framework.

A key feature of the newer languages is **type inference**: a mechanism by which the compiler automatically assigns the proper types to variables with minimal programmer annotation. Type inference preserves all of the consistency advantages of static typing, but lowers the burden on the programmer, which facilitates more rapid prototyping and development.

The integration of mutability introduces some subtle and tricky issues for type inference. In particular, first-class language support for mutability introduces the need for copy compatibility rules (e.g. a mutable object of type T can be assigned a value that is copied from an immutable object of type T). These rules weaken the unification constraints that allow type inference to proceed, and introduce the possibility that the inference mechanism may not be able to automatically infer a type. The problem here 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. Copy compatibility rules entail a loss of principal typings. This note examines the possible *ad hoc* resolution rules, and identifies why the rule we have selected for BitC is both formally sound and pragmatically practical.

For the same reason, we would like to automatically infer which variables are mutable so that the type inference mechanism knows to assign monomorphic types to these variables (which is necessary to eliminate some very obscure and confusing errors). The BitC compiler provides a limited degree of inference for variable mutability. The issues and tradeoffs involved in achieving this are discussed here.

This paper defines the typing rules for the BitC programming language, and the mechanisms used to infer types in BitC. Once a soundness proof has been constructed, the rules presented here will be integrated into the BitC specification.

The program examples in this document are written in the programming language BitC. Readers may find it helpful to refer to the BitC language specification[11].

2 Notation

2.1 Expressions

A complete specification of BitC expressions is provided in the BitC language specification [11].

2.2 Types

τ	Any type.
$\alpha, \beta, \gamma, \dots$ etc.	Type Variables.
<code>unit, bool, int8, ...</code>	Primitive types.
<code>#X₁, #X₂, ...</code>	Dummy types.
\otimes	Any structure type with no type arguments.
\otimes^s	The structure ‘s’ with no type arguments.
$\otimes^s(\bar{\tau})$	Any structure type with type arguments $\bar{\tau}$.
$\oplus, \oplus^u, \oplus^u(\bar{\tau})$	Union types, notation as described for structures.
\ominus	(Meta-syntax for) either structure or union type.
$\otimes_o, \oplus_o, \ominus_o$	Value structure / union types.
$\otimes_*, \oplus_*, \ominus_*$	Reference structure / union types.
<code>arr(τ)</code>	Array of τ .
<code>vec(τ)</code>	Vector of τ .
$\uparrow\tau$	The type: (<code>ref</code> τ).
$\tau_1 \rightarrow \tau_2$	The type: Function from τ_1 to τ_2 .
$\Psi\tau$	The type: (<code>mutable</code> τ) This represents the mutability of the <i>outermost</i> type constructor only.
$\Upsilon\tau$	The type: (<code>maybe-mutable</code> τ). Similar to $\Psi\tau$, but mutability status undecided.
$\tau \setminus \mathcal{C}$	Constrained type τ with the set of constraints \mathcal{C} .
$[[\tau_1, \bar{h}]]$	τ_1 , with a set of types \bar{h} as <i>hints</i> to resolve the mutability status of τ_1 .
ξ	Exception type.
(τ)	Same as τ , used for explicit paranthesization.

The type $\Upsilon\tau$ is referred to in English text as maybe-mutable- τ or simply maybe- τ (interchangeably) in the rest of the document. Collections of such types shall be referred to as “maybe mutable-types” or simply “maybe types” (interchangeably).

2.2.1 Examples

First, we introduce some commonly used composite types throughout the the document.

```
(defstruct (pair 'a 'b):val fst: 'a snd: 'b)
struct pair: (pair 'a 'b)
```

pair: $\otimes_o^{pair}(\alpha, \beta)$

```
(defunion (list 'a) nil (cons car:'a cdr:(list 'a)))
union list: (list 'a)
```

list: $\oplus_*^{list}(\alpha)$

```
(defunion (optional 'a):val none (some it:'a))
union optional: (optional 'a)
```

optional: $\oplus_o^{optional}(\alpha)$

As an example of the shallowness of mutable and maybe mutable constructors, $\Psi\otimes_o^{pair}(\text{int32}, \Upsilon\text{bool})$ denotes the type: (`mutable (pair int32 (maybe bool))`). That is, a mutable structure of value type ‘pair’, parameterized over immutable int32 and maybe mutable bool.

2.3 Type Expressions

This section introduces some operators that work on types. The meaning and purpose of these operators will become clear in the later sections of the document. All of the following operators have lower precedence than the type-constructors defined in the previous section.

$\tau_1 = \tau_2$ Equality of types.
 $\tau_1 \cong \tau_2$ Copy compatibility of types.
 $\tau_1 \hookrightarrow \tau_2$ Unification Linkage between two types.

3 Location Arguments and Returns

In BitC, the following expressions accept locations (addresses of cells) at positions indicated as *loc*, and return locations as their result (except `set!`, which returns `unit`):

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

Location semantics check ensures that there can (deeply) be no non-location as the first argument of a `set!`. This ensures that the programmer’s intuition of what is being `set!`ed is not violated by compiler transformations (example: by the introduction of temporaries due to transformation into an SSA form). All other forms can have an expression that returns a non-location at location positions *only* if they are not (deeply) within a `set!` expression.

4 Copy Compatibility

Unlike ML, BitC supports unboxed mutability – that is, mutable values need not be wrapped by a “ref cell.” 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 copy boundaries (ex: new binding, actual vs formal arguments). That is, the type of the copy of a value might differ in mutability with respect to the original value. This kind of type-compatibility shall hereinafter be called as copy compatibility, denoted by \cong .

For example:

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

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

Note: In the application `(plus1 10:int32)` above, the type of the actual argument `10` is `int32` and that of the formal argument `x` is Ψint32 , and $(\Psi\text{int32}) \cong (\text{int32})$.

However, this freedom must be constrained at a ref-boundary¹ since only the reference is getting copied and the type of the thing referenced cannot differ. It is imperative that we maintain the invariant that *every location must have exactly one type* at all times (see Appendix A). In this sense, mutability is actually an attribute of the *location* in question, rather than the type associated with that location.

¹ Ref-boundary refers to the fact a type is encapsulated within a reference type constructor (ex: `vec(τ)`).

4.1 Definition

When a value is copied onto a *new location* as the result of a copy operation, the type of the value stored in the new location is *only* required to be copy compatible with the type of the original value. Therefore, we define \cong as:

- $\Psi\tau \cong \tau$ – shallow top level mutability compatibility.
- $\text{arr}(\Psi\tau) \cong \text{arr}(\tau)$ – since arrays are value types, and the entire array is copied by value.
- $\ominus_{\circ}(\overline{\tau}_1) \cong \ominus_{\circ}(\overline{\tau}_2)$ iff $\overline{\tau}_{11} \cong \overline{\tau}_{21}$ and $\overline{\tau}_{12} = \overline{\tau}_{22}$, where $\overline{\tau}_{11} \in \overline{\tau}_1$, $\overline{\tau}_{21} \in \overline{\tau}_2$ are the set of type arguments that are *not* used within another reference type-constructor, and $\overline{\tau}_{12} \in \overline{\tau}_1$, $\overline{\tau}_{22} \in \overline{\tau}_2$ are the rest of the type arguments.

The mutability at the fields of structures and unions are determined by their corresponding definitions. However, we do have some freedom with the compatibility of the type-arguments of *value* structures and unions. For example: $\otimes_{\circ}^{\text{pair}}(\tau_1, \tau_2) \cong \otimes_{\circ}^{\text{pair}}(\Psi\tau_1, \tau_2) \cong \otimes_{\circ}^{\text{pair}}(\tau_1, \Psi\tau_2) \cong \otimes_{\circ}^{\text{pair}}(\Psi\tau_1, \Psi\tau_2)$. However, we cannot allow copy compatibility if the type-argument is used anywhere in the definition *within* another reference type. For example, for the following structure,

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

Two values whose types are parametrized over this structure are copy compatible if and only if their first type-arguments (that satisfy 'a above) are copy compatible and the second type-arguments (that satisfy 'b above) are strictly compatible.

4.2 Application

In theory, it is *sufficient* to require (only) copy compatibility at:

- A new binding.
- The argument and return positions of any expression that does not (respectively) expect or return a location.

The value returned by an expression returning a non-location will invariably end up in an expression that does not expect a location, and will thus (in turn) invoke the copy compatibility rule. The main reason to allow copy compatibility here is to allow explicit type-qualification of the return values of applications with a different but copy compatible type. For example:

```
(define (id x) x)
val id:(fn ('a) 'a)

(define xyz (id 10:int32):(mutable int32))
val xyz: int32
```

5 Impact on Type Inference

Freedom in compatibility of types with respect to mutability at copy-boundaries means we can no longer infer a unique type for an expression that involves copy compatibility. For example, in the following expression,

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

we know that the type of the literal 10 on the RHS is `int32`, but what is the type of `p`? It could either be `int32` or `Ψint32`.

When we cannot ascertain the mutability status of a type, we give it the type $\Upsilon\tau$, which means that it is *undecided* as to whether the actual type is $\Psi\tau$ or the immutable type τ . $\Upsilon\tau$ can later resolve to $\Psi\tau$ or τ due to further unification.

5.1 Unification

This section provides the algorithm for type unification in the presence of copy compatibility. The `Unify()` function will either fail with an error (\perp), or will succeed with a set of *linkages* of the unified types. These unification linkages are denoted by $\tau_1 \hookrightarrow \tau_2$, which can be understood to mean “ τ_1 resolves to τ_2 due to unification.”

$$\begin{aligned}
\text{Unify}(\tau, \tau) &= \{\} \\
\text{Unify}(\Psi\tau, \tau) &= \perp \\
\text{Unify}(\Upsilon\tau_1, \Psi\tau_2) &= \text{Unify}(\tau_1, \tau_2) \cup \{\Upsilon\tau_1 \hookrightarrow \Psi\tau_2\} \\
\text{Unify}(\Upsilon\tau_1, \tau_2) &= \text{Unify}(\tau_1, \tau_2) \cup \{\Upsilon\tau_1 \hookrightarrow \tau_2\} \\
\text{Unify}(\Psi\tau_1, \Psi\tau_2) &= \text{Unify}(\tau_1, \tau_2) \\
\text{Unify}(\Upsilon\tau_1, \Upsilon\tau_2) &= \text{Unify}(\tau_1, \tau_2) \cup \{\Upsilon\tau_1 \hookrightarrow \Upsilon\tau_2\} \\
\text{Unify}(\alpha, \tau) &= \{\alpha \hookrightarrow \tau\} \\
\text{Unify}(\uparrow\tau_1, \uparrow\tau_2) &= \text{Unify}(\tau_1, \tau_2) \\
\text{Unify}(\tau_1 \rightarrow \tau_2, \tau_3 \rightarrow \tau_4) &= \text{Unify}(\tau_1, \tau_3) \cup \text{Unify}(\tau_2, \tau_4) \\
\text{Unify}(\text{arr}(\tau_1), \text{arr}(\tau_2)) &= \text{Unify}(\tau_1, \tau_2) \\
\text{Unify}(\text{vec}(\tau_1), \text{vec}(\tau_2)) &= \text{Unify}(\tau_1, \tau_2) \\
\text{Unify}(\ominus^p(\overline{\tau}_i), \ominus^p(\overline{\tau}_j)) &= \bigcup \text{Unify}(\tau_i, \tau_j) \\
\text{Unify}(\tau_1, \tau_2) &= \perp
\end{aligned}$$

All Unification rules are commutative. The unification linkages are just a mechanism of accumulating equality constraints among types. They are represented as (uni-directional) linkages so that it is easy to visualize the type obtained as the result of unification by just following the unification links. Moreover, we don’t have to solve any equality constraints after unification – they are instead “solved” incrementally, each time we form a linkage. If the set of linkages obtained as the result of unification has two linkages $\tau_1 \hookrightarrow \tau_2$ and $\tau_1 \hookrightarrow \tau_3$ where $\tau_2 \neq \tau_3$ it is a contradiction, and hence a type error².

Note the order and manner in which maybe types are handled. Two types $\Upsilon\tau_1$ and $\Upsilon\tau_2$ must be linked even after τ_1 and τ_2 have unified, in order to ensure that their mutability also resolves the same way. Otherwise, all we have said is that the inner types must be strictly equal, and we might (due to further unification) end up in a situation where we infer $\Psi\tau \hookrightarrow \tau$ or $\tau \hookrightarrow \Psi\tau$. We can, if we prefer, introduce such linkages for all other types as well. The mutability wrappers must be processed even before type-variable linkages are processed. Otherwise, we can end up having types like $\Upsilon\Psi\tau$, that is, (maybe (mutable τ)), which can later “legally” form the type (immutable (mutable τ)), which is a contradiction.

If the type of the *lvalue* argument of a `set!` has type $\Upsilon\tau$, we will infer the constraint (rather, the linkage) $\Upsilon\tau \hookrightarrow \Psi\tau$.

We will now define a function `UnifiedType()`, that unifies two types, and returns the *final* type obtained as the result of unification. That is, it returns the type obtained by transitively following linkages produced as a result of unification, or returns \perp if unification fails.

$$\text{UnifiedType}(\tau_1, \tau_2) = \text{TheType}(\tau_1, \text{Unify}(\tau_1, \tau_2))$$

and

$$\begin{aligned}
\text{TheType}(\tau_1, \tau_1 \hookrightarrow \tau_2 \cup \mathcal{L}) &= \text{TheType}(\tau_2, \tau_1 \hookrightarrow \tau_2 \cup \mathcal{L}) \\
\text{TheType}(\Psi\tau, \mathcal{L}) &= \Psi\text{TheType}(\tau, \mathcal{L}) \\
\text{TheType}(\Upsilon\tau, \mathcal{L}) &= \Upsilon\text{TheType}(\tau, \mathcal{L}) \\
\text{TheType}(\uparrow\tau, \mathcal{L}) &= \uparrow\text{TheType}(\tau, \mathcal{L}) \\
\text{TheType}(\tau_1 \rightarrow \tau_2, \mathcal{L}) &= \text{TheType}(\tau_1, \mathcal{L}) \rightarrow \text{TheType}(\tau_2, \mathcal{L}) \\
\text{TheType}(\text{arr}(\tau), \mathcal{L}) &= \text{arr}(\text{TheType}(\tau, \mathcal{L})) \\
\text{TheType}(\text{vec}(\tau), \mathcal{L}) &= \text{vec}(\text{TheType}(\tau, \mathcal{L})) \\
\text{TheType}(\ominus^p(\overline{\tau}), \mathcal{L}) &= \ominus^p(\text{TheType}(\tau, \mathcal{L})^*)
\end{aligned}$$

² The Union operation on constraint sets must be careful to preserve this property.

$\text{TheType}(\tau, \mathcal{L}) = \tau$
 $\text{TheType}(\tau, \perp) = \perp$

All functions that operate on types (including the Unifier itself)³ shall be understood to operate on types that are obtained after all unification linkages are followed. That is, $f(\tau)$ is a shorthand for $f(\tau, \mathcal{L})$, wherein the *first* rule of the function is:

$$f(\tau_1, \tau_1 \hookrightarrow \tau_2 \cup \mathcal{L}) = f(\text{TheType}(\tau_1, \tau_1 \hookrightarrow \tau_2 \cup \mathcal{L}), \tau_1 \hookrightarrow \tau_2 \cup \mathcal{L})$$

These details are not shown in subsequent functions in the interest of brevity. Also, the operation on constrained types is not explicitly shown. Application (of a function) on a constrained type is the same as applying the function on the unconstrained type, because following unification linkages will have the effect of performing resultant substitutions in the constraint set as well.

5.2 An Example

In this section, we illustrate our inference mechanism with the help of an example. Consider the following definition:

```
(define p:(pair (mutable int32) bool)
  (pair (the int32 1) #t))
val p:(pair (mutable int32) bool)
```

In order to illustrate the types obtained after every step of inference, we will annotate this expression with *[numbers]* and then indicate the unification performed (if any) and the type obtained at each step. The annotations that use quoted numbers as in *[n']* indicate the type inferred for the *copy* of the value at the corresponding unquoted annotation *[n]*.

```
[12](define [6] [1]p: [5](pair [3](mutable [2]int32) [4]bool)
  [11'] [11](pair [9'] [9](the [8]int32 [7]1) [10'] [10]#t))
```

```
[1] =  $\alpha$ 
[2] = int32
[3] =  $\Psi\text{int32}$ 
[4] = bool
[5] =  $\otimes_{\circ}^{\text{pair}}(\Psi\text{int32}, \text{bool})$ 
[6] =  $\text{UnifiedType}([1], [5]) = \otimes_{\circ}^{\text{pair}}(\Psi\text{int32}, \text{bool})$ 
[7] =  $\beta \setminus \{\text{IntLit}(\beta)\}$ 
[8] = int32
[9] =  $\text{UnifiedType}([7], [8]) = \text{int32}$ 
[9'] =  $\Upsilon\text{int32}$ 
[10] = bool
[10'] =  $\Upsilon\text{bool}$ 
[11] =  $\otimes_{\circ}^{\text{pair}}(\Upsilon\text{int32}, \Upsilon\text{bool})$ 
[11'] =  $\Upsilon\otimes_{\circ}^{\text{pair}}(\Upsilon\text{int32}, \Upsilon\text{bool})$ 
[12] =  $\text{UnifiedType}([6], [11']) = \otimes_{\circ}^{\text{pair}}(\Psi\text{int32}, \text{bool})$ 
```

5.3 Principality of Inferred Types

The main idea of maybe types is to keep the mutability status of types “open” to further unification, and thus infer most-general (or principal) types at every step of inference. However, at a let-boundary⁴, it is no longer possible to

³ Except, of course, the function `TheType()`.

⁴ Let-boundary refers to the point where we have typed the bindings of a `let`, `letrec`, or `define` expression, and are about to generalize the types of the bound identifiers to obtain a (possibly) polymorphic type.

keep the mutability status of types “open” to further unification — at least in the case of universally quantified types — due to the value-restriction [12]. That is, in the case of the expression:

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

we cannot give p the type: $\forall\alpha.\Upsilon\oplus_{*}^{list}(\alpha)$ due to the value restriction. We can either give p the polymorphic type $\forall\alpha.\oplus_{*}^{list}(\alpha)$, or the monomorphic type $\Upsilon\oplus_{*}^{list}(\alpha)$. That is, in this system, there is no *principal type* that we can infer for p . Given this, we must “default” these maybe types to either mutable or immutable at a let-boundary. In the next section, we will identify some choices for defaulting these maybe types, and discuss their merits and limitations.

In contrast, ML infers principal types⁵ although it does not infer principal typings[6]. In order to achieve principality of inferred types, ML imposes the restriction that *only* references can be mutated, and *all* references must be mutable. This leaves the inference engine with exactly one choice (of the type to infer) when it looks at an expression. In BitC, we trade principality of inferred types in preference to a more expressive language (and type system). Section 5.5 deals with the various issues that arise due to this non-principality of inferred types.

It might be useful to note that this is not the only case in which we run into the non-principality problem in BitC. We do not infer structure types automatically from the field labels in the case of expressions like

```
s.fld
```

We cannot infer the type of s because, in BitC, we do not require field labels to be unique. So there is no principal type we can infer for the above expression, and it fails to type check. This problem is similar to SML’s record selection using the $\#$ operator.

5.4 Interaction with “Relaxed” Value Restriction

In BitC, we adopt the *relaxed* Value Restriction proposed by Jacques Garrigue[1], which states that even in an expansive expression, type variables that occur purely in non-contravariant (non-argument) positions can be generalized. More formally, this means that, in an *expansive* expression e with type τ , the generalizable type variables are given by:

$$\text{ftv}(\tau) \setminus \text{ftv}(\Gamma) \setminus V^{-}(\tau)$$

where:

$\text{ftv}(\tau)$ is the set of all the free type variables in τ ,

$\text{ftv}(\Gamma)$ is the set of all the free type variables captured in the environment Γ , and

$V^{-}(\tau)$ is the function which returns the set of “dangerous” type variables to be removed from the set of generalizable ones, and is defined (in the context of BitC) as:

$$\begin{aligned} V^{-}(\alpha) &= \{\} \\ V^{-}(\Psi\tau) &= \text{ftv}(\tau) \\ V^{-}(\tau_1 \rightarrow \tau_2) &= \text{ftv}(\tau_1) \cup V^{-}(\tau_2) \\ V^{-}(\ominus(\overline{\tau}_i)) &= \bigcup V^{-}(\tau_i) \end{aligned}$$

However, in a language that has unboxed mutability and unboxed composite values, this restriction turns out to be insufficient to ensure soundness. For example:

```
(define p:(mutable bool, 'a) (#t, none))
val p: (mutable bool, (optional 'b)) ??
```

⁵ Except in some cases of operator overloading in SML.

According to the above algorithm, we see that the value constructor `none` has the type $\oplus_{\circ}^{\text{optional}}(\alpha)$, and the type variable α occurs in non-contravariant position, and can thus be generalized. Now, if we give `p` the type $\forall\alpha.\otimes_{\circ}^{\text{pair}}(\Psi\text{bool}, \oplus_{\circ}^{\text{optional}}(\alpha))$, we can then write:

```
(let ((r1:(optional int8) p.snd)
      (r2:(optional double) p.snd)) ... )
```

which is clearly not OK because `r1` and `r2` have different sizes and this requires the polyinstantiation `p.snd` and thus its container `p`, which has other mutable components. Therefore, we cannot give `p` a polymorphic type here. In Section 6.1.3, we will describe a revised version of Garrigue’s restriction, that is sound in the presence of composite value types. The idea is to make the value restriction *propagate* to enclosing container type until we reach a ref-boundary.

In SML, all type variables in expansive expressions are monomorphed. While this will solve our problem, we did not adopt this solution because we wanted to preserve as much polymorphism as possible (especially in the case of reference types). Many use cases for adapting the relaxed restriction including the need for polymorphism in accessor functions can be found in [1]. Cyclone solves this problem by imposing a stricter rule (rather reminiscent of Haskell’s monomorphism restriction[3]) that *only* functions are given polymorphic types[2]. This is a reasonable thing to do in cyclone where most values are mutable, and where there is a distinction between *functions* and *function values*, but, is too restrictive for our purposes.

5.5 Inference Trade Offs

As we have seen in the previous section, we introduced maybe types in order to provide freedom of copy compatibility. However, in doing so, we “lost” precise mutability information about these types. This, combined with the fact that we must default these maybe types at a let-boundary, can result in inferred types that are quite surprising to the programmer. In this section we will consider the various issues in the trade-off between freedom and precision. In an ideal scheme, we should have:

- *Soundness*: The inferred types must be correct.
- *Completeness*: It should be possible to infer all possible sound types, albeit with qualifications.
- *Less programmer annotations*: The inference must be useful, and must not require excessive annotations in the “common case.” Since the “common case” is a subjective usability issue, we can approximate this to say: “it must retain as much available information as possible.”
- *Contained mutability*: The inference engine must not infer mutable types in ways that are “too surprising” to the user. From the standpoint of good programming practice, this might be stated as “mutability should not be promiscuously inferred.”

5.5.1 Fixing Maybe Types

This section explores the different possible “fixes” to default the maybe mutable types at a let-boundary. We will illustrate their effects using the example:

```
(let ((p (pair n:(mutable int32) (lambda (x) x))
      (q #f)) body)
  [Intermediate] p: (maybe (pair (maybe int32) (maybe (fn ('a) 'a))))
  [Intermediate] q: (maybe bool))
```

where we have obtained the types:

```
p:  $\Upsilon_{\otimes_{\circ}^{\text{pair}}}(\Upsilon\text{int32}, (\Upsilon\alpha \rightarrow \alpha))$ 
q:  $\Upsilon\text{bool}$ 
```

and are now at the step of generalization.

- FIX1: Fix *all* maybe types to immutable types deeply. This is in some sense the most “conservative” approach. It preserves as much polymorphism as possible at the cost of requiring programmer annotation for *every* mutable definition.

Using this fix, we obtain:

$$\begin{aligned} p: \forall \alpha. \otimes_{\circ}^{pair}(\text{int32}, \alpha \rightarrow \alpha) \\ q: \text{bool} \end{aligned}$$

Any attempt to `set ! p` or `q` in `body` will fail to type-check.

- FIX2: In the case of *local* definitions, fix all maybe types to immutable types deeply *only if* the type is polymorphic. This method preserves all of the polymorphism in FIX1, but also the mutability of fully-concrete types undecided (and thus open for further unification). The downside to using FIX2 of course, is that it tends to allow mutability in ways the programmer might be caught by surprise.

Using this fix, we obtain:

$$\begin{aligned} p: \forall \alpha. \otimes_{\circ}^{pair}(\text{int32}, \alpha \rightarrow \alpha) \\ q: \Upsilon \text{bool} \end{aligned}$$

Any attempt to `set ! p` in `body` will fail to type-check, but a `set !` on `q` will work fine, giving `q` the type Ψbool

- FIX3: Same as FIX1, except that if a *locally* defined identifier is used as the target (LHS) of a `set !`, it is given a shallowly mutable type. Since this fix infers shallow mutability, it will reduce the need for explicit type qualifications by the programmer in the common case (ex: iterators). However, it also has the problem of unintentional mutability (as in FIX2), but the spread of mutability is contained shallowly in this case.

Using this fix, if `p` and `q` are `set !`ed in `body`, we obtain:

$$\begin{aligned} p: \Psi \otimes_{\circ}^{pair}(\text{int32}, \alpha \rightarrow \alpha) \\ q: \Psi \text{bool} \end{aligned}$$

otherwise:

$$\begin{aligned} p: \forall \alpha. \otimes_{\circ}^{pair}(\text{int32}, \alpha \rightarrow \alpha) \\ q: \text{bool} \end{aligned}$$

Any attempt to `set ! p` or `q` in `body` will succeed, but an attempt to `set ! p.fst` will fail.

- FIX4: Fix all maybe types to whatever they were obtained from. If this does not completely resolve all maybe types, they are resolved to their immutable variants as the last resort. This requires a little more sophistication in the inference engine and the unifier. The idea here is to preserve the mutability information as a matter of “natural” flow of inference, but will allow the possibility of the user overriding these decisions by explicit annotations.

Using this fix, we obtain:

$$\begin{aligned} p: \otimes_{\circ}^{pair}(\Psi \text{int32}, \alpha \rightarrow \alpha) \\ q: \text{bool} \end{aligned}$$

Any attempt to `set ! p` or `q` in `body` will fail, but an attempt to `set ! p.fst` will succeed.

FIX2 and FIX3 must *only* be used in the case of *local* definitions. If we allow the types of top-level forms to be determined by their use cases, it will create a lot of indeterminism problems as the top level definitions have unlimited scope. Interfaces will no longer be definitive, and programs will start influencing the types of the libraries they use, making the order in which units-of-compilation are processed significant. Even within the same unit of compilation, the order in which top-level forms are written will influence the outcome of types in unpleasant ways. Therefore, in the interest of programmer sanity, we freeze type of a top-level form by the end of the definition.

FIX4 can be used in conjunction with FIX2 to obtain the types:

$$p: \Upsilon_{\otimes}^{pair}(\Psi_{int32}, \alpha \rightarrow \alpha)$$

$$q: \Upsilon_{bool}$$

or, in conjunction with FIX3 to obtain (in the presence of mutation):

$$p: \Psi_{\otimes}^{pair}(\Psi_{int32}, \alpha \rightarrow \alpha)$$

$$q: \Psi_{bool}$$

5.5.2 Where to Introduce Copy Compatibility?

We have already stated that, in principle, we can introduce copy compatibility at the argument or return position of any expression that does not (respectively) expect or return a location. However, in practice, we may want to constrain this freedom to a smaller subset in favor of tighter inference. From a usability standpoint, we must individually make a decision regarding type compatibility about each of the following kinds of expressions:

- New `let` bindings.

Introducing a new binding is one of the most natural forms where copy compatibility can be invoked. For example:

```
(define p:(mutable bool) #t:bool)
val p: (mutable bool)
```

- Arguments of function applications.

Passing arguments to a function is the the other natural form where we think about copy compatibility. For example:

```
(define (getFalse false:(mutable bool)) (set! false #f) false)
val getFalse: (fn (mutable bool) (mutable bool))

(define q (getFalse #t))
val q: bool
```

- Return values of function applications.

Strictly speaking, copy compatibility is not necessary at return locations as the return result must invariably end up in a position that will in turn invoke copy compatibility. The main reason to preserve this is to write expressions like:

```
(define (id x) x)
val id: (fn ('a) 'a)

(define xyz (id n:(mutable int32)):int32)
val xyz: int32
```

where we explicitly qualify the result of an application to a different but copy compatible type. The other reason, of course, is to preserve intuition and symmetry between arguments and return types.

However, we can end up in some interesting situations because of this freedom. For example:⁶

```
(import ls bitc.list)
(define (list->vector lst)
  (make-vector (length lst)
    (lambda (n) (ls.list-nth lst n))))
val list->vector: (forall ((copy-compat 'a 'b))
  (fn ((list 'a)) (vector 'b)))
```

The type of `list->vector` appears to be $\oplus_*^{list}(\alpha) \rightarrow \text{vec}(\alpha)$. However, because of the copy compatibility at the return of the inner lambda, the type is actually $\oplus_*^{list}(\alpha) \rightarrow \text{vec}(\beta) \setminus \{\alpha \cong \beta\}$.

Then if we write:

```
(define (lvm lst:(list (mutable bool))) (list->vector lst))
val lvm: (fn ((list (mutable bool))) (vector bool))
```

the type of `lvm` is $\oplus_*^{list}(\Psi\text{bool}) \rightarrow \text{vec}(\text{bool})$ and not $\oplus_*^{list}(\Psi\text{bool}) \rightarrow \text{vec}(\Psi\text{bool})!$ (because we must default the maybe types at top-level).

The fix for this — as we would expect — is additional programmer annotations in order to clamp the type of `list->vector` to $\oplus_*^{list}(\alpha) \rightarrow \text{vec}(\alpha)$, as in:

```
(define (list->vector lst)
  (make-vector (length lst)
    (lambda (n) (ls.list-nth lst n))):(vector 'a))
val list->vector: (fn ((list 'a)) (vector 'a))
```

- Arguments/return values of value constructors – structure/union constructors, `vector`, `array`, `make-vector`.

It is actually tempting to allow copy compatibility at argument and return positions of constructor applications also, because a copy is anyway being performed, and it would be nice to write things like:

```
(defstruct St f1:(mutable bool) f2:(mutable int32))
struct St: St

(let ((p (St #t 25)))
  (set! p.f1 #f))
```

That is, even though the structure constructor is expecting arguments of type Ψbool and Ψint32 , it can be satisfied by literal arguments `#t` and `25` with types `bool` and `int32` respectively. Otherwise, we will have to write:

```
(let ((mTrue:(mutable bool) #t)
      (m25:(mutable int32) 25)
      (p (St mTrue m25)))
  (set! p.f1 #f))
```

or, we will have to accept mutable type qualification on literals, and then write:

```
(let ((p (St #t:(mutable bool) 25:(mutable int32))))
  (set! p.f1 #f))
```

⁶ `copy-compat` is an internal typeclass that relates any two copy compatible types.

However, if we allow copy compatibility at constructor arguments, the inference of parametrized types becomes a little surprising.

```
(define mTrue:(mutable bool) #t)
val mTrue: (mutable bool)

(define p (cons mTrue nil))
val p: (list bool)
```

As a result of copy compatibility, the type of the RHS is $\Upsilon \oplus_{\star}^{list}(\Upsilon \text{bool})$. Note that we lost the mutability information to allow freedom of compatibility even though $\oplus_{\star}^{list}(\tau)$ is a *reference* type. Now, if we default maybe types to immutable, we will obtain the type $\oplus_{\star}^{list}(\text{bool})$ for `p`, which does not “seem” like what the user intended. Here, we can argue two ways:

- Constructors actually accept a location but internally make a copy of the argument, for which we do not invoke copy compatibility.
- Just because the list was constructed from a mutable value, it does not mean that the user intends to make a list of mutable elements. So, we must invoke copy compatibility here.

In expressions that contain a deep nesting of constructors, copy compatibility will lead to rather ugly qualification requirements. For example:

```
(define r:(vector (list (mutable 'a))) (vector (cons mTrue nil)))
val r: (vector (list (mutable bool)))
```

If the needed mutability is not shallow, the user will have to write out the detailed type up to the level in which the mutability is desired (as in the above example), which is not very pleasant.

Having said this, there is actually another possibility, which is to allow copy compatibility at constructor boundaries, but default the maybe types to what they were before inference threw away their mutability status. The next section explores this case.

- Arguments/return values of other non-location returning expressions like conditionals – `if`, `switch`, `try/catch`, `and`, `or`, `not`, ... etc.

Allowing copy compatibility at these positions will allow us to write expressions like:

```
(define p:int32 (if #t 1:int32 n:(mutable int32)))
val p: int32
```

The rationale here is that the result of these expressions cannot appear in any position expecting a location, and it is therefore “safe” to think of them as if an application to the function `if` had happened.

However, as is usual, this will lead to further loss of mutability information. For example:

```
(define (vectorizer1 x) (vector x))
(define (vectorizer2 x) (vector (if #t x x)))

;; Types obtained without copy compatibility at constructor arguments
val vectorizer1: (fn ('a) (vector 'a))
val vectorizer2: (forall ((copy-compat 'a 'b)) (fn ('a) (vector 'b)))
```

Here, `vectorizer1` has the type $\alpha \rightarrow \text{vec}(\alpha)$ `vectorizer2` has the type $\alpha \rightarrow \text{vec}(\beta) \setminus \{\alpha \cong \beta\}$ even if we say that constructor arguments *do not* invoke copy-inference.

5.5.3 How Much of Copy Compatibility to Accept?

We have already stated that, in theory, we can infer a copy compatible, but different type for every new value created due to a copy-operation. However, in practice, we may choose to allow *only* top-most level of mutability-compatibility in favor of a more precise inference.

5.5.4 Compatibility of Function Types

Consider the following function definition:

```
(define (mut-c c) (set! c #\a) c)
```

From the implementation of the function `mut-c`, we infer the type $\Psi_{\text{char}} \rightarrow \Psi_{\text{char}}$ for it. However, as the caller of this function, it is reasonable to think of `mut-c` as having the type $\Psi_{\text{char}} \rightarrow \Psi_{\text{char}}$ or $\Psi_{\text{char}} \rightarrow \Psi_{\text{char}}$, due to the copy compatibility at function argument and return positions. Moreover, while writing a proclamation, we would like to write the type from the caller's perspective without disclosing the mutability information of the *internal copy* of the arguments or return types. For example:

```
(proclaim mut-c: (fn (char) char))
```

We will hereinafter refer to the type of a function from the implementation viewpoint as the “internal type” and the one from the caller's viewpoint as the “external type” of a function. The external type can be obtained by simply dropping all mutable wrappers at copy compatible positions.

Now, when the user explicitly writes a function type, should this be taken as the internal type, or the external type of the function? That is, should the following definition:

```
(define mut-c:(fn (char) char) (lambda (c) (set! c #\a) c)
```

be type correct? Similarly, can `mut-c` be supplied as an actual argument when the formal argument expects the type $\text{char} \rightarrow \text{char}$?

```
(defstruct St fnxn: (fn (char) char))
```

```
(define xyz (St mut-c))
```

In other words, this leads to a notion of compatibility of function types in terms of the copy compatibility of their argument and return types. It may not be good to call this copy compatibility because it is not compatibility created due to the copy of the function value, but is a consequence of copy compatibility at argument and return positions.

Finally, should the user be ever allowed to write the internal type of a function? One possible solution to this problem is given in Section 6

5.5.5 Type Class Instances

With regard to type class method declarations, it is reasonable to expect that they should also declare external types without constraining the internal types (and this the implementation) of the instances supplied for these methods.

Due to copy compatibility, at method argument and return positions, the compiler must only accept those instances whose methods have different external type signatures. For example:

```
(deftypeclass (CL1 'a)  
  fn1: (fn ('a) 'a))
```

```
(definstance (CL1 bool)
  (lambda (x:bool) x))
(definstance (CL1 (mutable bool))
  (lambda (x:(mutable bool)) x))
```

cannot be permitted because the two instances for CL1 are in-differentiable from the caller’s perspective. However, we can allow:

```
(deftypeclass (CL2 'a)
  fn1: (fn ((vector 'a)) 'a))

(definstance (CL2 bool)
  (lambda (x) (vector-nth x 0)))

(definstance (CL2 (mutable bool))
  (lambda (x) (vector-nth x 0)))
```

6 Proposed Solution

Keeping in mind, the various trade-offs raised in the previous section, we shall now propose a particular policy for handling copy compatibility. This is by no means “the” solution to the problem, but reflects our aesthetic judgment of the best way to capture the programmer’s intuition about the flow of types in the language.

- Allow copy compatibility to the full extent, up to a ref-boundary.
- Allow copy compatibility to be invoked at arguments of all expressions that do not expect a location. Similarly for return types of expressions that do not return locations.
- In order to fix the maybe types, we adopt a hybrid of FIX4 and FIX2.
 - The topmost (shallow) mutability is determined by the FIX2 rule in the case of local definitions. In the case of global definitions, any unresolved topmost mutability is set to immutable.
 - In the case of any other unresolved-mutability, we will try to resolve it by unifying it with — a simplified form of — the original type, so that we can preserve the original mutability information. Intuitively, this does means that we will restore the original mutability status. However, it is done through unification because, in cases like `list->vector` above, we must also ensure that the copy compatibility constraints on new type variables created for the sake of copy compatibility are turned into equality constraints. The next section explains this rule more formally.
 - If there are any residual unresolved maybe types even after applying the above rules, fix them to immutable.
- All function types `(fn . . .)` refer to the external type.⁷ By construction, this means that all functions that are copy compatible at function argument and return positions are *equal*. The copy compatibility of function types refers to the compatibility of shallow mutability with respect to the entire function’s type.

Function types that declare mutable types at copy compatible positions will be rejected as an error. For example:

```
(define abc:(fn ((mutable bool)) 'a) (lambda (x) x))
ERROR!
```

However, any type-qualifications on the arguments of a function within its body (as opposed to within the function’s type) are understood to reflect the internal type. For example, the following definition is type-correct.

⁷ This is a syntactic restriction that does not yet appear in the specification.

```
(define abc (lambda (x:mutable bool) (set! x #t) x))
val abc: (fn (bool) bool)
```

Declarations (proclamations) are required to declare a type that will exactly match external type inferred for the corresponding definition.

6.1 Formalization

6.1.1 Introducing Maybe Types

We need a new enhancement to our notation of types. The pair: $[[\Upsilon\tau_1, \bar{h}]]$ stands for the type $\Upsilon\tau_1$ with the understanding that the maybe-ness in $\Upsilon\tau_1$ may be resolved by unifying with one of the types in the *hint-set* \bar{h} if necessary. *Only* maybe types are permitted to have hints.

At a copy-boundary, if the type of the original expression is τ , we give the type $\text{TypeOfCopy}(\tau)$ to the value that was created due to the copy. TypeOfCopy is defined as:

```
TypeOfCopy( $\Psi\tau$ ) =  $[[\text{TypeOfCopy}(\tau), \{\Psi\tau\}]]$ 
TypeOfCopy( $[[\Upsilon\tau, \bar{h}]]$ ) =  $[[\text{Core}(\text{TypeOfCopy}(\tau)), \{[[\Upsilon\tau, \bar{h}]]\}]]$ 
TypeOfCopy( $\alpha$ ) =  $[[\beta, \{\alpha\}] \setminus \{\alpha \cong \beta\}]$ 
TypeOfCopy( $\uparrow\tau$ ) =  $[[\Upsilon\uparrow\tau, \{\uparrow\tau\}]]$ 
TypeOfCopy( $\tau_1 \rightarrow \tau_2$ ) =  $[[\Upsilon(\tau_1 \rightarrow \tau_2), \{\tau_1 \rightarrow \tau_2\}]]$ 
TypeOfCopy( $\text{arr}(\tau)$ ) =  $[[\Upsilon\text{arr}(\text{TypeOfCopy}(\tau)), \{\text{arr}(\tau)\}]]$ 
TypeOfCopy( $\text{vec}(\tau)$ ) =  $[[\Upsilon\text{vec}(\tau), \{\text{vec}(\tau)\}]]$ 
TypeOfCopy( $\ominus_*(\bar{\tau})$ ) =  $[[\Upsilon\ominus_*(\bar{\tau}), \{\ominus_*(\bar{\tau})\}]]$ 
TypeOfCopy( $\ominus_o(\bar{\tau})$ ) =  $[[\Upsilon\ominus_o(\text{TypeOfCopy}(\bar{\tau}_i), \bar{\tau}_j), \{\ominus_o(\bar{\tau})\}]]$  where,  $\bar{\tau}_i \in \bar{\tau}$  are the set of type arguments that are not used within another reference type-constructor, and  $\bar{\tau}_j \in \bar{\tau}$  are the rest of the type arguments.
TypeOfCopy( $\tau$ ) =  $[[\Upsilon\tau, \{\tau\}]]$ 
```

and:

$$[[\Upsilon\tau, \bar{h}]] = \Upsilon\tau$$

The function $\text{TypeOfCopy}(\tau)$ increases the maybe-ness of a type τ to the maximum permissible limit, so that the type of the copy can unify with a different but copy compatible type.

We will also have to change our unifier to suite the new system of types. Instead of the rule:

$$\text{Unify}(\Upsilon\tau_1, \Psi\tau_2) = \text{Unify}(\tau_1, \tau_2) \cup \{\Upsilon\tau_1 \leftrightarrow \Psi\tau_2\}$$

we now have:

$$\begin{aligned} \text{Unify}([[\Upsilon\tau_1, \bar{h}]], \Psi\tau_2) &= \text{Unify}(\tau_1, \tau_2) \cup \{[[\Upsilon\tau_1, \bar{h}]] \leftrightarrow \Psi\tau_2\} \\ \text{Unify}([[\Upsilon\tau_1, \bar{h}_1]], [[\Upsilon\tau_2, \bar{h}_2]]) &= \text{Unify}(\tau_1, \tau_2) \cup \{[[\Upsilon\tau_1, \bar{h}_1]] \leftrightarrow [[\Upsilon\tau_2, \bar{h}_2]] \leftrightarrow [[\Upsilon\tau_1, \bar{h}_1 \cup \bar{h}_2]]\} \\ \text{Unify}([[\Upsilon\tau_1, \bar{h}]], \tau_2) &= \text{Unify}(\tau_1, \tau_2) \cup \{[[\Upsilon\tau_1, \bar{h}]] \leftrightarrow \tau_2\} \end{aligned}$$

and instead of

$$\text{TheType}(\Upsilon\tau, \mathcal{L}) = \Upsilon\text{TheType}(\tau, \mathcal{L})$$

we have:

$$\text{TheType}([[\Upsilon\tau_1, \bar{h}]], \mathcal{L}) = [[\Upsilon\text{TheType}(\tau_1), \bar{h}]]$$

Note that the unifier preserves as much hint information as possible. When time comes to resolve these maybe types, we can pick the most “appropriate” hint according to any desired policy.

6.1.2 Resolving Maybe Types

At a let-boundary, if we have an expression like:

```
(let (([1]p:[2]qual [3]expr)) body)
```

and if we let:

- [1] τ be the type of p before generalization.
- [2] τ_1 be the type obtained from explicit qualification `qual`
- [3] τ_2 be the inferred type for the *copy* of `expr`,

then, we have:

$$\tau = \text{adjMaybe}(\text{TopMutAdj}(\text{UnifiedType}(\tau_1, \tau_2)))$$

where:

$$\begin{aligned} \text{adjMaybe}([\Upsilon\tau_1, \hbar]) &= \text{Ground}(\text{UnifiedType}(\text{adjMaybe}(\tau_1), \wp([\Upsilon\tau_1, \hbar]))) \\ \text{adjMaybe}([\Psi\tau_1, \tau_2]) &= \Psi\text{adjMaybe}(\tau_1) \\ \text{adjMaybe}([\tau_1, \tau_2]) &= \text{adjMaybe}(\tau_1) \\ \text{adjMaybe}(\alpha) &= \alpha \\ \text{adjMaybe}(\uparrow\tau) &= \uparrow\text{adjMaybe}(\tau) \\ \text{adjMaybe}(\tau_1 \rightarrow \tau_2) &= \text{adjMaybe}(\tau_1) \rightarrow \text{adjMaybe}(\tau_2) \\ \text{adjMaybe}(\text{arr}(\tau)) &= \text{arr}(\text{adjMaybe}(\tau)) \\ \text{adjMaybe}(\text{vec}(\tau)) &= \text{vec}(\text{adjMaybe}(\tau)) \\ \text{adjMaybe}(\ominus^p(\overline{\tau})) &= \ominus^p(\text{adjMaybe}(\tau)^*) \\ \text{adjMaybe}(\tau) &= \tau \end{aligned}$$

and:

$$\begin{aligned} \wp([\Upsilon\tau_1, \hbar]) &= \tau_2 \text{ if } \exists \text{ an } \textit{immutable} \text{ type } \tau_2 \text{ in } \text{map}(\wp, \hbar); \tau_3 \text{ if } \exists \text{ a } \textit{maybe} \text{ type } \tau_3 \text{ in } \text{map}(\wp, \hbar); \text{ the} \\ &\text{mutable type } \tau_4 \text{ otherwise.} \\ \wp(\Psi\tau) &= \Psi\wp(\tau) \\ \wp(\alpha) &= \alpha \\ \wp(\uparrow\tau) &= \uparrow\alpha \\ \wp(\tau_1 \rightarrow \tau_2) &= \alpha \rightarrow \beta \\ \wp(\text{arr}(\tau)) &= \text{arr}(\alpha) \\ \wp(\text{vec}(\tau)) &= \text{vec}(\alpha) \\ \wp(\ominus^p(\overline{\tau})) &= \ominus^p(\overline{\alpha}) \\ \wp(\tau) &= \tau \end{aligned}$$

and:

$$\begin{aligned} \text{Ground}(\Psi\tau) &= \Psi\tau \\ \text{Ground}(\Upsilon\tau) &= \tau \\ \text{Ground}(\tau) &= \tau \end{aligned}$$

and:

$$\begin{aligned} \text{TopMutAdj}(\Psi\tau) &= \Psi\tau \text{ if this is a local definition that is mutated in } \textit{body}, \tau \text{ otherwise.} \\ \text{TopMutAdj}([\tau_1, \tau_2]) &= [[\text{TopMutAdj}(\tau_1), \tau_2]] \\ \text{TopMutAdj}(\tau) &= \tau \end{aligned}$$

6.1.3 Value Restriction

The previous section explained how to calculate a type that is a candidate for generalization. In this section, we will actually explain the generalization step, and the algorithm for value-restriction. We will have a new function V^{--} which takes a type and a mode parameter. The mode can be one of “remove” or “keep” depending on whether we come from a value type or a reference type context.

Now, in an expression e with type τ , the set of generalizable type variables in τ is given by:

$$\begin{aligned} & \text{ftv}(\tau) \setminus \text{ftv}(\Gamma) \text{ if } e \text{ is a } \textit{value}, \text{ and} \\ & \text{ftv}(\tau) \setminus \text{ftv}(\Gamma) \setminus V^{--}(\tau, \text{remove}) \text{ if } e \text{ is } \textit{expansive}, \end{aligned}$$

where:

$$\begin{aligned} V^{--}(\alpha, \text{remove}) &= \{\alpha\} \\ V^{--}(\alpha \setminus \{\text{IntLit}(\alpha)\}, \text{remove}) &= \{\alpha\} \\ V^{--}(\alpha \setminus \{\text{FloatLit}(\alpha)\}, \text{remove}) &= \{\alpha\} \\ V^{--}(\alpha, \text{keep}) &= \{\} \\ V^{--}(\Psi\tau, \text{mode}) &= \text{ftv}(\tau) \\ V^{--}(\uparrow\tau, \text{mode}) &= V^{--}(\tau, \text{mode}) \\ V^{--}(\tau_1 \rightarrow \tau_2, \text{mode}) &= \text{ftv}(\tau_1) \cup V^{--}(\tau_2, \text{mode}) \\ V^{--}(\text{arr}(\tau), \text{mode}) &= V^{--}(\tau, \text{remove}) \\ V^{--}(\text{vec}(\tau), \text{mode}) &= V^{--}(\tau, \text{keep}) \\ V^{--}(\Theta_o(\overline{\tau}_i), \text{mode}) &= \bigcup V^{--}(\tau_i, \text{remove}) \\ V^{--}(\Theta_*(\overline{\tau}_i), \text{mode}) &= \bigcup V^{--}(\tau_i, \text{keep}) \end{aligned}$$

Note that the $\uparrow\tau$ *does not* explicitly pass the keep-mode. This is because such types can later be *derefed*, exposing the inner value (recall that *deref* returns a location, and not a copy). Therefore, *only* those type variables which are enclosed in reference types that can never be dereferenced are safe to be generalized in an expansive expression. The *IntLit* and *FloatLit* rules above (only) expressively state that these primitive integer/floating point types are unboxed value types.

6.2 Some Examples

In this section, we will illustrate our scheme with some examples.

```
(define xyz:(mutable int32) 10)
val xyz:(mutable int32)

(define p:(mutable 'a) (pair xyz #t))
val p: (mutable (pair (mutable int32) bool))
```

Inn the above example, *xyz* clearly has the type Ψint32 . Here is the annotated expression for the definition of *p* followed by the types inferred at at every step of inference:

```
[10] [9] [8](define [4] [1]p:[3](mutable [2]'a)
      [7'] [7](pair [5'] [5]xyz [6'] [6]#t))
```

Here are the types obtained after each step of inference:

```
[1] =  $\alpha$ 
[2] =  $\beta$ 
[3] =  $\Psi\beta$ 
```

```

[4] = UnifiedType([1], [3]) =  $\Psi\beta$ 
[5] =  $\Psi\text{int32}$ 
[5'] = TypeOfCopy([5]) =  $[[\Upsilon\text{int32}, \{\Psi\text{int32}\}]]$ 
[6] = bool
[6'] = TypeOfCopy([6]) =  $[[\Upsilon\text{bool}, \{\text{bool}\}]]$ 
[7] =  $\otimes_{\circ}^{\text{pair}}([[ \Upsilon\text{int32}, \{\Psi\text{int32}\}]], [[\Upsilon\text{bool}, \{\text{bool}\}]])$ 
[7'] = TypeOfCopy([7]) =  $[[\Upsilon\otimes_{\circ}^{\text{pair}}([[ \Upsilon\text{int32}, \{\Psi\text{int32}\}]], [[\Upsilon\text{bool}, \{\text{bool}\}]]),$ 
 $\{\otimes_{\circ}^{\text{pair}}([[ \Upsilon\text{int32}, \{\Psi\text{int32}\}]], [[\Upsilon\text{bool}, \{\text{bool}\}]])\}]]$ 
[8] = UnifiedType([4], [7']) =  $\Psi\otimes_{\circ}^{\text{pair}}([[ \Upsilon\text{int32}, \{\Psi\text{int32}\}]], [[\Upsilon\text{bool}, \{\text{bool}\}]])$ 
[9] = TopMutAdj([8]) =  $\Psi\otimes_{\circ}^{\text{pair}}([[ \Upsilon\text{int32}, \{\Psi\text{int32}\}]], [[\Upsilon\text{bool}, \{\text{bool}\}]])$ 
[10] = adjMaybe([9]) =  $\Psi\otimes_{\circ}^{\text{pair}}(\Psi\text{int32}, \text{bool})$ 

```

In the rest of the examples, we will only show the expressions and the final type obtained. In the following expression:

```

(define r (vector (cons mTrue:(mutable bool) nil)))
val r: (vector (list (mutable bool)))

```

`r` gets the “expected” type $\text{vec}(\oplus_{\star}^{\text{list}}(\Psi\text{bool}))$. However, it is still possible to get the type $\text{vec}(\oplus_{\star}^{\text{list}}(\text{bool}))$. for `r` through explicit qualification if one desires so.

```

(define r:(vector (list bool))
  (vector (cons mTrue:(mutable bool) nil)))
val r: (vector (list bool))

```

Finally, our `list->vector` also gets the “correct” type, without any type qualifications.

```

(import ls bitc.list)
(define (list->vector lst)
  (make-vector (length lst)
    (lambda (n) (ls.list-nth lst n))))
val list->vector: (fn ((list 'a)) (vector 'a))

```

The type inferred for `list->vector` is $\oplus_{\star}^{\text{list}}(\alpha) \rightarrow \text{vec}(\alpha)$, as is apparent from the definition, and not $\oplus_{\star}^{\text{list}}(\alpha) \rightarrow \text{vec}(\beta) \setminus \{\alpha \cong \beta\}$.

We will now give some examples that illustrate our value-restriction scheme. In these examples, we will explicitly write the universal quantification of type variables in the case of polymorphic types. First, the example we considered in Section 5.4:

```

(define p:(mutable bool, 'a) (#t, none))
val p: (pair (mutable bool) (optional #X255))

```

is now well typed, because `p` is now given the non-polymorphic type $\otimes_{\circ}^{\text{pair}}(\Psi\text{bool}, \oplus_{\circ}^{\text{optional}}(\#X_{255}))^8$ and not $\forall\alpha.\otimes_{\circ}^{\text{pair}}(\Psi\text{bool}, \oplus_{\circ}^{\text{optional}}(\alpha))$.

All the permissible cases in original (Garrigue) restriction are still preserved for reference types :

```

(define q:(mutable bool, (list 'a)) (#t, nil))
val q: (pair (mutable bool) (list 'a))

```

$q: \forall\alpha.\otimes_{\circ}^{\text{pair}}(\Psi\text{bool}, \oplus_{\star}^{\text{list}}(\alpha))$

Type-variables wrapped in a reference type are OK even if they lie within an encompassing value type.

⁸ Monomorphic types are forced to dummy types at top-level.

```

(define r:(mutable bool, 'c) (#t, (nil, nil)))
val r: (pair (mutable bool) (pair (list 'a) (list 'b)))

r:  $\forall \alpha. \beta \setminus \otimes_{\circ}^{pair} (\Psi_{bool}, \otimes_{\circ}^{pair} (\oplus_{\star}^{list}(\alpha), \oplus_{\star}^{list}(\beta)))$ .

```

Type-variables wrapped in a value type must be removed even if they lie within an encompassing reference type.

```

(define m:(mutable bool, 'b) (#t, (vector none)))
val m: (pair (mutable bool) (vector (optional #X143)))

m:  $\otimes_{\circ}^{pair} (\Psi_{bool}, \text{vec}(\oplus_{\circ}^{optional}(\#X_{143})))$ 

```

Otherwise, one could write:

```

(let ((mm1:(optional int8) (vector-nth m.snd 0))
      (mm2:(optional double) (vector-nth m.snd 0))) ... )

```

Accessor-like functions will work OK for reference types but will not work for value types.

```

(define (car x:(list 'a)) ... )
val car: (fn ((list 'a)) 'a)

(define fstElem (car (cons nil nil)))
fstElem: (list 'a)

car:  $\forall \alpha. \oplus_{\star}^{list}(\alpha) \rightarrow \alpha$ 
fstElem:  $\forall \alpha. \oplus_{\star}^{list}(\alpha)$ 

(define (fst x:( 'a, 'b)) x.fst)
val fst: (fn ((pair 'a 'b)) 'a)

(define fstPart: (fst (nil, nil)))
val fstPart: (list #X432)

fst:  $\forall \alpha. \beta \setminus \otimes_{\circ}^{pair}(\alpha, \beta) \rightarrow \alpha$ 
fstPart:  $\oplus_{\star}^{list}(\#X_{432})$ 

```

The right thing to do in this case is to define `fst` as a *term* and not as an ordinary functions whose application will be considered expansive.

6.3 Limitations

The algorithm proposed above is not a panacea. First, there is a peculiar kind of generality we have lost, which is impossible to recover even with explicit qualification (that is, with the facilities for qualification present in the current language specification). For example, there is no qualification for `list->vector` that we can use, in order to obtain the type $\oplus_{\star}^{list}(\alpha) \rightarrow \text{vec}(\beta) \setminus \{\alpha \cong \beta\}$. That is, even if we re-write the definition as

```

(import ls bitc.list)
(define list->vector:(forall ((copy-compat 'a 'b)) (fn ((list 'a)) (vector 'b))))
  (lambda (lst)
    (make-vector (length lst)
      (lambda (n) (ls.list-nth lst n))))))
val list->vector: (fn ((list 'a)) (vector 'a)))

```

we obtain the type $\oplus_{\star}^{list}(\alpha) \rightarrow \text{vec}(\alpha)$ for `list->vector` according to the above algorithm since a statement copy compatibility does not preclude equality.

6.4 Future Considerations

Function types can be generalized regardless of whether they contain any mutable or maybe types within them. Therefore, if an expression is a value (that is, not expansive), there is no reason to fix the maybe types contained within a function type in order to retain soundness or polymorphism. This way, we can retain the principality of types for functions definitions. This approach will — among other things — resolve the problem with the type of `list->vector` identified in the previous section.

```
(import ls bitc.list)
(define (list->vector lst)
  (make-vector (length lst)
    (lambda (n) (ls.list-nth lst n))))
val list->vector: (forall ((copy-compat 'a 'b)) (fn ((list 'a)) (vector 'b)))
```

The type of `list->vector` will now be $\forall \alpha. \beta \in \bigoplus_*^{list}(\alpha) \rightarrow \text{vec}([\beta, \alpha]) \setminus \{\alpha \cong \beta\}$. That is, we obtain the fully generic type for `list->vector`, but also save a hint to resolve the types at use occurrences. Therefore, any use of `list->vector` will yield “expected” types, as in:

```
(define (lvm lst:(list (mutable bool))) (list->vector lst))
val lvm: (fn ((list (mutable bool))) (vector (mutable bool)))
```

However, we must *only* retain those hints that rely on the function arguments. Otherwise, this will lead to violation of abstraction across the function boundary as the types obtained at application will depend on the implementation of the function.

There is a further relaxation to the way we fix maybe types that can be considered. Whereas we need to fix the maybe-ness in the type itself, we may not have to fix the maybe-ness of the constraints on the type, in order to retain sound typing. This will facilitate cases like:

```
(deftypeclass (CL2 'a)
  zeroth: (fn ((vector 'a)) 'a))

(definstance (CL2 bool)
  (lambda (x) (vector-nth x 0)))

(define p:(vector (mutable bool)) (vector (zeroth (vector #t))))
val p:(vector (mutable bool))
```

to type-check even though there is no instance `CL2(Ψbool)` because, for `p`, we initially obtain the type $\text{vec}(\Psi\text{bool}) \setminus \{\text{CL2}([\Upsilon\text{bool}, \text{bool}])\}$ ⁹, and the constraint `CL2([\Upsilonbool, bool])` can be satisfied by the instance `CL2(bool)`, and we finally get the unconstrained type `vec(Ψbool)`.

7 Acknowledgments

Scott Smith of Johns Hopkins University provided extensive ongoing comments, advice, and guidance in the course of this work. Mark Jones was kind enough to educate us on type classes, which provided an essential basis for integrating these ideas.

8 Conclusion

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

⁹ Originally, we would have obtained the type $\text{vec}(\Psi\text{bool}) \setminus \{\text{CL2}(\Psi\text{bool})\}$.

with their pros and cons. We have also selected a strategy based on our aesthetic judgment of the best way to capture the programmer’s intuition about the flow of types. By default, our strategy infers types based on the “natural” flow of type information in an expression, but preserves the possibility of obtaining other permissible types through explicit qualification in most cases.

A Other Considerations

While we are thinking about the various trade-offs and possible freedom in compatibility of types with respect to mutability, we might be tempted to relax this rule beyond copy compatibility. For example, it feels natural that any value that has a deeply mutable type must be able to flow into an argument that expects an immutable value. For example, if we have a function `vector->string` with type `vec(char) → string` we would think that it should be possible to pass a value with type `vec(Ψchar) → string` as the actual argument to `vector->string`. However, this mechanism is not sound.

While the `vector->string` case will work, it does not generalize. To see the problem, consider the procedure:

```
(define (vector-return vec:(vector char)) vec)
```

`vector-return` has the type `vec(char) → vec(char)`. If we allow mutable values to arbitrarily flow into immutable arguments, the application of `vector-return` in

```
(let ((mv (vector (mutable #\a) (mutable #\b))))
  (let ((imv (vector-return mv)))
    (== (vector-nth imv 0)
        (begin (set! (vector-nth mv 0) #\c)
              (vector-nth imv 0)))))
```

is legal, and the result of this expression is `#f!`

Since `vec(char)` is a reference type, if we allow this use of `vector-return`, we would end up with `imv` and `mv` pointing to identical content, but disagreeing about the types of the cells. If this were allowed, the compiler would never be entitled to believe that vector cells (or more generally, the types of fields within value types) are constant (even if so declared)! In particular, the compiler cannot safely eliminate the extra index into `imv` in this example unless it can determine that the first expression of the `begin` form cannot mutate it – the “constness” of `imv` is merely a local aberration rather than a statement of true immutability. We will in essence, lose the mathematical notion of constness if we allow this application. Therefore we impose the “one location, one type” rule.

References

- [1] J. Garrigue, “Relaxing the Value Restriction” *International Symposium on Functional and Logic Programming* 2004.
- [2] D. Grossman, “Quantified Types in an Imperative Language” *ACM Transactions on Programming Languages and Systems* 2006.
- [3] S. P. Jones, I. Hughes, *et. al.* “Haskell 98 Language and Libraries: the Revised Report” *Journal of Functional Programming* January 2003.
- [4] ISO, *International Standard ISO/IEC 8652:1995 (Information Technology — Programming Languages — Ada)* International Standards Organization (ISO). 1995.
- [5] ISO, *International Standard ISO/IEC 9899:1999 (Programming Languages - C)* International Standards Organization (ISO). 1999.

- [6] Trevor Jim, “What are principal typings and what are they good for?” *ACM Symposium on Principals of Programming Languages* 1996.
- [7] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988
- [8] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised* The MIT Press, May 1997.
- [9] Simon Peyton Jones (ed.). *Haskell 98 Language and Libraries: The Revised report*. Cambridge University Press. 2003.
- [10] Iavor S. Diatchki, Mark P. Jones, and Rebekah Leslie. “High- level Views on Low-level Representations.” *Proc. 10th ACM Conference on Functional Programming (ICFP 2005)*. pp. 168–179. Tallinn, Estonia, September 2005. Published as *SIGPLAN Notices* 9(40). ACM Press. September 2005.
- [11] J. S. Shapiro, S. Sridhar, M. S. Doerrie, “BitC Language Specification” <http://coyotos.org/docs/bitc/spec.html>
- [12] A. K. Wright, “Simple Imperative Polymorphism” *Lisp and Symbolic Computation* 8(4):343–355, 1995.