

# Type Inference for Unboxed Types and First Class Mutability

Swaroop Sridhar Jonathan Shapiro, Ph.D.

*Systems Research Laboratory*

Dept. of Computer Science

Johns Hopkins University

October 22, 2006

## Abstract

Systems programs rely on fine-grain control of data representation and use of state to achieve performance, conformance to hardware specification, and temporal predictability. The robustness and checkability of these programs could be greatly improved if modern type systems and programming language ideas, such as polymorphism and type inference, could be applied to these programs.

BitC is a higher-order programming language in the tradition of ML and Haskell, extended to incorporate both state and the expression of unboxed and low-level datatypes. State and unboxed value types interact in subtle ways with polymorphic type-inference. Unless handled with care in the language design, interactions of these features can lead to unsoundness or results that are counter-intuitive to the programmer. Because instances of value types may have mutable components, a decision must be made concerning their compatibility at copy boundaries: should structurally equivalent types that differ only in their mutability be considered compatible. The choice impacts both the amount of polymorphism that the language can preserve and the burden of type annotation imposed on the programmer.

This paper presents some of these challenges and our design for how to address these issues.

## 1 Introduction

Modern programming languages such as ML [9] or Haskell [10] provide newer, stronger, and more expressive type systems than systems programming languages such as C [8, 6] or Ada [5]. 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. C# supports state and representation, but does not support type inference, higher-order types, or mathematically well-founded mechanisms for abstraction.

A key feature of the newer languages is **type inference**, a mechanism by which the compiler automatically assigns types to variables with minimal programmer annotation. Type inference preserves the consistency advantages of static

typing, but lowers the burden on the programmer, facilitating more rapid prototyping and development.

BitC [12] is a higher order programming language for low-level systems programming. It is type safe, and supports polymorphic type inference. It exposes machine-level representation of types, and provides precise programmer control over data layout. It supports first class polymorphism over all types (including unboxed types). It also supports “first class” mutability in the sense value of any type can be mutable — mutable values need not be wrapped by a “ref cell” as in ML.

Since BitC is a call-by-value 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. 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 document explores these issues, and their implications as a matter of theory and practice.

Program examples in this document are written in the programming language BitC. Readers may find it helpful to refer to the BitC language specification [12].

## 2 Location Semantics

BitC is a stateful language. Variables and fields may be given explicitly mutable type by the programmer, or may be assigned a mutable type by the type inference mechanism. The following syntactic forms in BitC 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)
```

These forms may also be used as expressions. In expression context, the location is implicitly dereferenced to obtain the appropriate value.

In systems programming languages, the programmer depends critically on a mental model of the relationship between locations and storage. An assignment is an update to the content of a particular memory cell. The *identity* of that cell — not just its denotation — is semantically significant. In consequence, mutable identifiers must not be multiply instantiated, and therefore cannot be given polymorphic type.

Mutable value types have implications for type compatibility. In C, it is legal to write:

```
const int *cpi = ...;
int *pi = cpi;
*pi = 5;
```

Because of this, the compiler cannot rely on the alleged const-ness of aliased identifiers and structures. In BitC, mutability is part of the field type, and the language design takes care to ensure that a location never has more than one type. This is enforced in the type system.

### 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,  $\text{int32} \cong (\text{mutable int32})$ .

Copy compatibility need not be restricted to the shallow case, but must not extend past a reference boundary, because the target of the reference is not copied. The required invariant is that every location must have exactly one type at all times, and allowing copy compatibility to extend past a reference boundary creates aliasing opportunities that violate the “one location, one type” constraint. In this sense, mutability is actually an attribute of the location in question, rather than the type associated with that location.

Therefore, we can define copy compatibility as follows:

- $\tau \cong (\text{mutable } \tau)$  — **direct compatibility** ( $\tau$  denotes any type).
- $(\text{array } \tau) \cong (\text{array } (\text{mutable } \tau))$  — **array element compatibility**. Arrays are value types, and the entire array is copied by value.
- **Compatibility of composites**. Composite types are copy compatible if and only if their fields are copy compatible. To achieve this, a restriction must be imposed

for parametric types: any type variable in the composite type definition that appears within a reference type is required to match exactly at all positions where it appears. Because of this rule, instantiations of `'a` are subject to copy compatibility but instantiations of `'b` must match exactly in:

```
(defstruct (St 'a 'b):val
  f1:'a f2:(ref 'b))
(St char char)  $\cong$  (St (mutable char) char)
(St char char)  $\not\cong$  (St char (mutable char))
```

In addition to argument passing, assignment, and binding, there is a wide range of language design choices about where to permit copy compatibility. For example, we might also consider the return type of expressions that do not expect or return locations. We might also weaken the compatibility requirements of `if` expressions to say that the *then* and *else* clauses need only have copy-compatible return types. Each of these decisions — but especially the ones entailing control flow constructs — has impact on the effectiveness of type inference.

### 4 Impact on 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 means that it is undecided as to whether the actual type is `int32`, or `(mutable int32)`. The choice may 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)))
```

#### 4.1 Principality of Inferred Types

The key idea of maybe types is to defer commitments about the mutability status of types, and thus infer most-general (or principal) types wherever possible. BitC is a

let-polymorphic language. This means that maybe type decisions for universally quantified types cannot be deferred past their bindings because of the so-called “value restriction” [13]. 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, in this system, 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 infers principal types<sup>1</sup> although it does not infer principal typings [7]. 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 inference for every expression. In BitC, we trade principality of inferred types for a more expressive language (and type system). Section 4.2 deals with the issues that arise due to this non-principality of inferred types.

## 4.2 Inference Trade Offs

The previous section argued that we “lose” precision of inferred types (with respect to mutability) by the introduction of copy compatibility. In this section we will consider the various issues in the trade-off between freedom and precision of inferred types. An ideal inference scheme should be sound and complete (it should be possible to infer all sound types, albeit with qualifications). It must not require excessive programmer annotations in the “common case.”

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.

### 4.2.1 Resolving Maybe Types at Let Boundaries

One possibility is to fix all types to immutable. For example:

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

<sup>1</sup> Except in some cases of operator overloading in SML.

This scheme will preserve all polymorphism possible, but will mandate a programmer annotation for every mutable location. The alternative would be to fix everything to mutable, 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.

### 4.2.2 Where to Apply Copy Compatibility

We can choose whether to introduce copy compatibility at various constructs like new bindings, function application/return, constructors, conditional expressions, etc. If we introduce copy compatibility promiscuously, the inferred types are sometimes very surprising. For example:

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

The type of `list->vector` appears to be:

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

but is actually:<sup>2</sup>

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

If we default maybe types that are ultimately unresolved to immutable, in the following definition we obtain:

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

which is a correct typing, but is counter-intuitive.

### 4.2.3 How Much of Copy Compatibility to Use?

Should we use copy compatibility to the maximum permissible limit (as defined in Section 3), or should we restrict it to top-level shallow mutability compatibility only? Here again, we will be trading expressiveness for preciseness of mutability information.

### 4.2.4 What expressions Can be Polymorphic?

Should we allow every possible use of polymorphism, or should we restrict it to functions only? For example, in Cyclone only function definitions are polymorphic [4]. This is a reasonable thing to do in Cyclone, where there is a distinction between functions (code) and function pointers

<sup>2</sup> `copy-compat` is a special type class that relates two copy compatible types.

(data); but in doing so, Cyclone loses the “first class” notion of function values.<sup>3</sup> Another 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.

## 5 Design Choices in BitC

Having identified the various trade-offs in the previous section, we shall 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 aesthetic 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 programmers. If, as the designers of the language, *we* are surprised at the behavior of the compiler, it seems reasonable to expect that others will be too.<sup>4</sup>

- 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.
- If a locally defined 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.
- Every time we form a maybe type due to a copy operation, we remember “hints” to resolve this maybe-ness in the type. At a let boundary, we resolve any maybe-ness in the inferred type by unifying with (a simplified form of) 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.

Hints must be accumulated at the merge of branching expressions and we pick the most immutable of all hints to resolve the maybe type. This ensures that inferred types are deterministic. If there are any residual unresolved maybe types even after unifying with hints, we fix them to immutable.

- We allow all values and not just functions to be polymorphic. In BitC, we enforce a modified form of Jacques Garrigue’s Relaxed Value restriction [3], with some extra restrictions to deal with unboxed polymorphic types.

<sup>3</sup> The formalization in [4] uses a generalization of this rule that preserves first class notion of functions, but nevertheless requires explicit annotation of every polymorphic expression.

<sup>4</sup> We have politely ignored one user’s proposal that we host an annual obfuscated BitC typings contest.

- All function types (`fn ...`) must declare immutable types at copy compatible positions.<sup>5</sup> That is,

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

The intuition is that type of a function must be described from the caller’s perspective (the external type), and must hide the mutability information.<sup>6</sup> By construction, this means that all functions that are copy compatible at function argument and return positions are equal.

- No two instances of a type class that cause the external type of any method to have indistinguishable external type must co-exist at any point.

A detailed explanation of these design choices can be found in our accompanying technical report [14].

### 5.1 Examples

```
(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.

Moreover, the `list->vector` example described in Section 4.2 now gets “expected” type:

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

In the case of conflicting hints, we pick the most-immutable type by default. For example:

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

Since we restrict copy compatibility at a ref-boundary, type safety cannot be circumvented by aliasing as in the C code shown in Section 2. For example, the following expression will fail due to a type error.

```
(let ((x:(ref (mutable int)) (dup 10)))
  (let ((y:(ref int) x)) ;; ERROR !!
    (set! (deref y) 200)))
```

<sup>5</sup> This is a syntactic restriction that does not yet appear in the specification.

<sup>6</sup> However, any type-qualifications on the arguments of a function within its body correspond to the internal types, and may contain any mutable qualification.

## 6 Related Work

Diatchki *et al* have implemented support for bit-level word types in Haskell [11]. Their solution could be extended to the full `defrepr` mechanism of BitC, but preserves the Haskell idiom of restricting all use of state to the IO monad.

SysObjC [1] extends the C programming language with object-like value types, but does address type safety in the face of polymorphism.

Cqual [2] infers `const` types in C programs, but does not handle polymorphism.

## 7 Conclusions

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 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.

## 8 Acknowledgments

We are grateful for the assistance of Scott Smith, who 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.

## References

[1] 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.

[2] 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.

[3] J. Garrigue, "Relaxing the Value Restriction" *International Symposium on Functional and Logic Programming* 2004.

[4] D. Grossman, "Quantified Types in an Imperative Language" *ACM Transactions on Programming Languages and Systems* 2006.

[5] ISO, *International Standard ISO/IEC 8652:1995 (Information Technology — Programming Languages — Ada)* International Standards Organization (ISO). 1995.

[6] ISO, *International Standard ISO/IEC 9899:1999 (Programming Languages - C)* International Standards Organization (ISO). 1999.

[7] Trevor Jim, "What are principal typings and what are they good for?" *ACM Symposium on Principles of Programming Languages* 1996.

[8] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988

[9] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised* The MIT Press, May 1997.

[10] Simon Peyton Jones (ed.). *Haskell 98 Language and Libraries: The Revised report*. Cambridge University Press. 2003.

[11] 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.

[12] J. S. Shapiro, S. Sridhar, M. S. Doerrie, "BitC Language Specification" <http://www.coyotos.org/docs/bitc/spec.html>

[13] A. K. Wright, "Simple Imperative Polymorphism" *Lisp and Symbolic Computation* 8(4):343–355, 1995.

[14] 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>