# The Origins of the BitC Programming Language[†]

SRL Technical Report 2008-04

*Warning: Work in Progress*

Jonathan Shapiro, Ph.D. Swaroop Sridhar M. Scott Doerrie

*The EROS Group, LLC* *Systems Research Laboratory*

Dept. of Computer Science

Johns Hopkins University

UNPUBLISHED

## Abstract

The process of programming language creation is a subject of too little reflection and retrospection. Newcomers to the field (including, in some measure, us) regularly propose new languages without quite understanding what they are getting into or how big the task is. Those of us who have already crawled down the language design rathole rarely have time to describe either the depth of the hole or the true majesty of the rats involved.

This paper describes the motivation and early design evolution of the BitC programming language, making our excuses in hindsight and providing a compilation of some of the things that we learned along the way. It describes the problems we were attempting to solve, and where the effort has taken us so far. It also discusses some of the balance points that we attempted to maintain in the language design process, and where (at least in our view) we think that we succeeded or failed. Some of the problems we were trying to address now have other solutions, but whether BitC *per se* succeeds or not, we still feel that a language like this remains motivated.

## 1 Introduction

Designing a coherent and sensible programming language is hard. Even if technically successful, a new language is preposterously unlikely to achieve broad penetration. Because of this, it has become fashionable for programming language designers to explain from the comfortable perspective of hindsight why they thought to undertake such an absurd task, what they hoped they might accomplish, and perhaps what they (re)discovered along the way.

In this paper we describe the original motivation and early evolution of the BitC programming language. BitC started as an attempt to address a specific problem, and morphed rapidly into a full-blown programming language effort. Along the way our ideas and thoughts about languages suffered some severe setbacks and some interesting (at least to us) turns. We will try to address some of those here.

## 2 Motivation

The original motivation for BitC emerged from Shapiro's long-standing interest in robust systems, which dates back to his earliest involvement in capability-based operating systems in 1990. After 7 years of focusing on microkernel engineering and research on the EROS system [26], it occurred to Shapiro that it might be possible to formally verify the confinement property that had been working in practice for years. This led to a verification proof with collaboration with Sam Weber in 2000 that did so [27], and then to a course on high assurance systems co-taught with David Chizmadia in 2001. David had served as a high assurance evaluator under the TCSEC and ITSEC standards.

The course effort generated surprises for both participants. Four things soon became apparent: (1) none of the practically usable security assurance techniques address *code* in a comprehensive way. In consequence, none of them are iterable at reasonable cost. (2) The only substantive tool and language for building high-confidence systems [2] is the proprietary Spark/ADA tool set, which is syntactically very restricted. Ada does not effectively exploit the advances in programming language theory that have been made over the last two decades. (3) None of the widely available general-purpose proof systems such as Isabelle/HOL, PVS, Coq, or ACL2 had ever been connected to any programming language that might be useful for building a real, general-purpose system — or at least not publicly. (4) The current evaluation schemes do not adequately take into account the ability of modern programming techniques and tools to automate property checks and enforce software development disciplines.

In hindsight, we would add a fifth observation: there is a need for a continuous range of repeatable assurance approaches requiring less than overwhelming effort. Today, the options available are basically "none" (human reviewed, therefore not repeatable and minimal confidence) or "too much" (formally verified, therefore not repeatable but potentially high confidence). At least one example now exists where a verification efforts have proceeded successfully using a subset of the C language (SEL4 [15]). The effort required was truly herculean, and it is a *tour de force* of verification. At the time of this writing (September, 2008), the SEL4 team is still working on connecting their low-level formal design to the code, and also on connecting their high-level formal design to their security specification.

The point that we really want to make about the SEL4 approach is that it does not generalize to any *other* effort to verify programs in C. Having achieved a verification of the SEL4 kernel, and assuming that all of the verification libraries associated with that result were made public, the advance in the available verification art that can be applied to other programs is very limited. Verifications using C as the source language are expensive, "one off" efforts.

We emphasize that this is primarily a criticism of C rather than a complaint about the SEL4 results. C is somehow bipolar: Hercules apparently *can* verify selected programs, but even fairly modest checks on general programs remain very hard to achieve. To make this area sustainable, we need both more generalizable verification results than C permits and languages in which more modest but useful levels of confidence do not require such Herculean labors. That is: we need more options on the spectrum between fully verified and wishful assertion. A type safe systems language with some degree of property tracking is useful for that whether we can verify programs in that language or not.

Our own efforts with the Coyotos system proceeded from the opposite end of the problem. We first built a formal high-level system model, and verified that the our key security policy was enforced [27]. Without this, the effort to build a low-level design would be unmotivated. We also had an fairly unusual implementation to work with:

- EROS is an atomic action system. Processes do not retain kernel stacks or other state while asleep. On restart, the pending system call is re-initiated from scratch. This eliminates a wide range of concurrency analysis problems that arise even in single-threaded implementations.
- Aliasing is one of the primary sources of complexity in reasoning about stateful programs, because it can lead to precondition violations. Because of the atomic action design in EROS, there are almost no cases in the EROS implementation where aliasing *matters*. The implementation is very conservative; in all situations where aliasing might occur, it either checks explicitly or it restarts the operation from scratch. We had significantly positive results confirming the correctness of the implementation using the MOPS static checker [5], so we were fairly confident about this issue.
- Also because of the atomic action design, all memory in the EROS kernel is type-stable. While objects are allocated and deallocated, we use a typed heap whose partitioning is established at startup and thereafter does not change. This gives us the same kind of memory safety that is exploited by SAFECode [7, 16].
- Finally, we had a system that had been designed primarily by processor architects rather than software architects, with the consequence that many checkable invariants of the system design had been explicitly stated and were periodically checked in the system. Each of these informally tested some important consistency property, and failures had become very rare. This gave us some degree of hope that a prover might be able to confirm them given a suitable implementation.

What we *didn't* have was an implementation that we felt we could verify. EROS had been implemented in C before any consideration of software verification entered the picture, and it used idioms from C that seemed unlikely (to us) to be explainable to a verifier. In hindsight, our group may have been pessimistic about what was possible; certainly we did not know the literature of dependent type analysis as well as we do now. By the time we did, the BitC effort was well underway, and our comment above about needing a more general solution for safe systems code construction at varying levels of automatable confidence remains (in our view) valid.

## 3 Alternatives to C

Having concluded that C wasn't a sustainable programming language for safe system construction, we remained reluctant to design a new language. Shapiro's group at Bell Laboratories had built the first commercial application ever constructed in early C++, which provided an opportunity to watch Bjarne's efforts at relatively close hand. Shapiro had later served as a turnaround CEO over a company that was drowning in recursive complexity — in part because they had committed themselves to a proprietary language. These experiences and concerns about adoption made us was very reluctant to undertake building a new programming language if there was any means to avoid it.

Connecting a proof system to a systems programming language entails modeling the language precisely within the prover so that the behavior of a program can be analyzed. Success at this type of endeavour has three requirements: a language that has a well-defined, unambiguous semantics, a thorough understanding of both formal programming language theory and prover technology, and a program whose construction is carefully tailored to the requirements of proof (ideally co-implemented with the proof). We could learn the theory, and for a variety of reasons (including some noted above) we thought that we might already have the program (EROS), but the choosing a language presented a problem.

**C, C++** For performance and expressiveness reasons, most production systems (including EROS) are written in C or C++. Neither of those languages has anything remotely like a solid mathematical foundation, and neither language is memory safe. In consequence, the *meaning* of a program written in C or C++ isn't well defined. It is possible to program in something else (typically the prover) and treat a small subset of a particular C implementation as a well-defined assembly language, as the L4.verified project has since done [15]. The practical feasibility of that approach was uncertain when we started the BitC work.

The "C as target language" approach carries most of the costs of developing a new programming language with none of the advantages — the problem being that the programming language emerges implicitly rather than from a coherent design effort, and never becomes "first class." In our view, it is unlikely that this approach can scale to larger systems or larger development groups. Perhaps more important, it is not a "continuous solution," in the sense that it provides no help for projects that require a substantial improvement in software robustness in quality, but for which the burden of formal specification, proof, and severe constraints on programming idiom cannot be tolerated.

**Java, C#** Languages like Java or C# don't really address the problem space of systems programs. Neither language offers performance that is competative with C or C++ — not even when statically compiled. While dynamic translation is the most often cited source of overhead, another significant source of overhead is the tendancy for C# and Java to syntactically and philosophically discourage unboxed objects and encourage heap allocation. Heap allocation in turn induces garbage collection at unpredictable times, which flatly rules these languages out for systems requiring hard timing guarantees.

A less obvious issue is the absence of first-class union value types in the managed subset of the Common Language Runtime (CLR) or the corresponding parts of the Java Virtual Machine (JVM). These are absolutely necessary for low-level systems programming, so one must either abandon Java/C#/Spec# to implement these low-level objects (thereby abandoning the foundation for checking), or one must find a more appropriate language.

In addition to the problems of expressiveness, neither Java nor C# was designed with formal property checking in mind. Spec# [3], a language developed by Microsoft Research to retrofit formal property checking to C#, has been forced to introduce some fairly severe language warts to support precondition and postcondition checking, but the language does not attempt to address the underlying performance issues of C#.

**Haskell, ML** Research programming languages like Haskell [22] and ML [20] didn't seem to offer any near-term solution. Diatchki's work on fine-grain representation in Haskell [25] is not yet main-stream, and had not yet started when we began work on BitC. Support for state in Haskell exists in the form of the I/O monad [23], but in our opinion the monadic idiom does not scale well to large, complexly stateful programs,[1] and imposes constraints that are unnatural in the eyes of systems programmers.

Ultimately, the problem with Haskell and ML for our purposes is that the brightest and most aggressive programmers in those languages, using the most aggressive optimization techniques known to the research community, remain unable to write systems codes that compete reasonably with C or C++. The most successful attempt to date is probably the FoxNet TCP/IP protocol stack, which incurred a 10x increase in system load and a 40x penalty in accessing external memory relative to a conventional (and less aggressively optimized) C implemenation. [4, 6]

It is instructive that no performance evaluation of the hOp and House [10] systems was attempted five years later. The operative research question in both systems was whether systems programming is *feasible* in Haskell, not whether the result is practically *useful*. In the eyes of the systems community, it is inconceivable that these two questions might be separated, and in the absence of a performance evaluation the work has limited credibility. From a programming language perspective, we view the hOp and House results as positive first indicators that must be viewed skeptically. Given the ratio of C to Haskell

[1] Monad composition is a recognized problem in the literature, and is an area of active research.

code in the House implementation, it does not seem to us that the case for Haskell as a systems programming language has been sustained (yet). The primary contribution of these efforts from our perspective is to experimentally confirm many of the design decisions that we have made in BitC.

**SPARK/Ada** The language and proof system best suited to what we were attempting is probably the SPARK/Ada system. [2] Perhaps we should have used it, but one of our goals was to construct an "open exemplar" that could be examined, modified, and re-proved. In this context SPARK/Ada raised three objections. Ada, whatever its merits, is a dying language. The SPARK Examiner is not openly available, so we could not build an open-source engineering effort around it. Finally, the Ada subset supported by the SPARK Examiner is unnecessarily severe in its omitted features – enough so that we plainly could not express even the carefully restricted initialization practices of the EROS or Coyotos kernels.

**ACL2** Given our desire to formally verify the Coyotos kernel (the successor to EROS), the absence of any suitable programming language led us to consider creating a new one. But by this point we had developed a naïve enthusiasm for ACL2. One of the beauties of ACL2 is that it maintains a successful pun between programs and terms in the prover, and simultaneously provides very powerful automation. It also provides an abstraction (the state object, or STOBJ) that offers an illusion of nearly-stateful programming to the developer while maintaining a pure language under the covers. Given this, we decided to see if we might not be able to encode our kernel more or less directly in ACL2 and implement some form of translation to C. This resulted in "Pre-BitC, an attempt to retrofit types to ACL2 in an overlay language from which we could directly generate C code (in the "C as assembly code" view).

We would eventually run into serious troubles with ACL2, but that is getting a bit ahead of the story.

# 4 BitC/ACL2

ACL2 [13] is a first-order prover that is built on an applicative subset of Common LISP. It was very attractive to us, primarily because it offered a very high degree of automation. As newcomers to software verification that was very appealing. Unfortunately, the absence of static types started to get in our way quickly.

Like LISP, ACL2 is dynamically typed. While preconditions can be expressed that describe argument type constraints, the prover requires that all functions be complete over their domains, not merely complete when inputs satisfy preconditions. This leads to a NaN-like idiom in which most functions return some out of band value for invalid inputs, and call sites are forced to check for and propagate this value. Property checks then get re-written as "show that property X is true if procedure F got sensible arguments". Then you have to prove that the wierd value never actually emerges anywhere in your particular program.

In essence, this amounts to imposing static typing through abuse of the prover. The first problem is that the ACL2 code involved rapidly becomes unreadable. BitC got started as a preprocessor to inject the required junk into our ACL2 programs automatically. The second problem is that a *huge* number of useless and irrelevant proof objectives are generated this way, which presents proof scalability problems.

At first the problem did not seem insurmountable. LISP has a rich macro system, and it seemed likely that we could construct a meta-language on ACL2 that would automatically insert the extra checks. This language came to be known as BitC, because it was attempting to implement a little bit of C on top of ACL2. The attempt rapidly bogged down.

The BitC implementation itself used macros that executed code written in ACL2. Because ACL2 is first order, common idioms like `map` are impossible. This is particularly painful in systems like compilers that walk AST trees recursively. Each of our recursions over ASTs had to be laboriously unwound to avoid higher-order procedures — invariably in cases that could be handled by first order code if only sufficient inlining were performed. This was very frustrating.

As we started to work around these issues, it became clear that the transform being performed by our macros was not a simple transform. The entire strategy for "BitC as Macros" relied on the transform being simple, because ACL2 was going to report problems using the post-transform code, and we needed to be able to see from that how to fix the original. At some point it became clear that we had lost the usability battle. Also, by this point, we knew a bit more about language embeddings and we had concluded that a deep, small-step embedding was going to be required in any case. This weakened the case for using any sort of direct overlay language.

Even if `map` had not put an end to BitC/ACL2, we slowly realized that we were building a real programming language, but one that was too restrictive for general use. There are a lot of applications that want safety and speed, and perhaps some property checking, but don't really care about full-strength formal correctness proofs. Many of those programs use constructs that simply are not legal in a first-order stateless language like ACL2, and therefore could not be legal in BitC/ACL2.

Eventually, we gave up on BitC/ACL2, and decided to build a real programming language, mainly because we seemed to be doing it anyway. Almost imperceptably, our goal changed from a specialized, provable systems language to a general-purpose safe systems language having a provable subset language inside it. Perhaps we decided that if we were building a programming language anyway we might as well build one that was broadly useful. From the ACL2 experience, we thought we might understand how to preserve a provable subset language, but the proof of that is in the doing, and the existence of a provable BitC subset has not yet been demonstrated.

# 5 Goals for BitC

Perversely, a strong goal for BitC was *not* to innovate.[2] We hoped to restrict ourselves to integrating first-class handling of mutability and representation into an ML-style language. We liked ML for several reasons. We wanted the abstractive power that comes with polymorphic types and the reduction of maintenance overheads and errors that come with type inference. This pushed us strongly in the direction of a Hindley-Milner style of inference scheme and type system. We didn't necessarily think that we would make much use of higher-order functions in a kernel, but the language needed to be general purpose.

One advantage we had was the decision to set aside source compatibility with any existing language, and settle for linkage compatibility. Another advantage was that some very good and easily retargeted compiler back ends existed for us to build on, most notably LLVM [17].

As Diatchki would demonstrate before we could publish the first work on BitC, adding representation to a functional language actually isn't that hard [25], because Value types don't actually alter the core semantics of a functional programming language. Diatchki's initial scheme wasn't general (it didn't handle pointers), and it therefore evaded some minor formalization challenges, but both groups (proceeding independently) eventually nailed those details down.

We also wanted a language base that grew from a rigorously specified semantics. If we wanted to prove things about programs, being able to automate the formal semantics of our language was critical. We saw very early that handling representation added no change in the core semantics of the language. We won't fully understand the impact of mutability on the core semantics of BitC until the semantics is properly formalized, but the impact in the formalization of the type system has been surprisingly small.

Of our goals, the controversial one in the eyes of the programming language community is generalized support for state. Several people advocated that we adopt the Haskell *monads* alternative. We rejected this option early, largely because we didn't think we could explain it to general systems programmers, and it imposes constraints that we found idiomatically restrictive. Later we would come to feel that monads do not scale to large programs well. In ML, the use of state in the language is very carefully constrained to simplify the core language semantics. When state is introduced generally, the language is suddenly forced to adopt a rich, first-class semantics of locations into both the core semantics and the core type system. When let-polymorphism is present, adding core mutability threatens the ability of the type inference engine to generate principal types. We wanted type inference, because Shapiro had witnessed an unending stream of examples in UNIX where failures to keep types synchronized across a code base led to errors, and because useable abstraction is hard to get in this class of language without both type variables and type inference.

Being blissfully ignorant (at least when we started) of formal type theory, all of this looked hard but straightforward, which describes what kernel researchers do pretty much all the time: navigate hard engineering compromises. As Bjarne put it: “modest, but preposterous.” ([24], p. 1). In fact, we soon learned that nobody had ever discovered a sound and complete type system incorporating polymorphism and general mutability at the same time.

We decided very early that the ML surface syntax should not survive. It is hopelessly ambiguous, which is more or less inexcusable, but after all it's just a yucky pragmatics issue. The beauty of the ML family really does go more than skin deep, but given the skin, it needs to.

## 5.1 Programmer Compatibility

A primary concern was usability *in the eyes of systems programmers*. There is a strong view in the PL community that issues of representation are non-semantic. In systems programs this is incorrect, because the hardware representation is prescriptive and cache effects have significant impact on concurrency and performance. It is therefore terribly important to a systems programmer to know what the data representation is. BitC needed to bring the ML family out of the ivory tower a bit.

[2] This sort of “let's do a conservative, evaluable experiment” approach makes project funding nearly impossible to obtain in academic circles, which may help to explain why computer scientists tend to step on each other's toes rather than standing on each other's shoulders.

But the other usability concern is what might loosely be termed “transparency.” A systems programmer reading C code is able to maintain a useful intuition about what is going on at the machine level. In particular there is a fairly precise notion about control flow sequencing, individual costs of machine-level operations, and (very informally) the presence or absence of instruction scheduling opportunities that the compiler or the hardware might exploit or barriers that might present restrictions. This is particularly true in concurrent code, where barriers are critical to correctness. These sorts of intuitions are almost entirely lost in the presence of extensive pointer indirection or aggressive optimization. Because of its preponderence for heap allocation, extracting any sort of decent performance from an ML program requires a level of optimization that is well beyond aggressive. We needed a language that could preserve the illusion that “what you see is what you get.” This, among other issues, drove us to choose eager rather than lazy evaluation.

Something we *didn’t* want was an object-oriented language. OO languages remain a popular fad, but our experience using C++ in the EROS system was that it actively got in the way of understanding what was going on. We also think it’s a bad sign that the C++ language evolution has adopted a policy of “extension by *ad hoc* complexity.” When writing the first book on reusable C++ coding in 1991 [29], Shapiro hoped that compilers might catch up with the C++ language specification within 5 to 8 years, but every time that threatens to happen somebody moves the language. Setting aside the resulting complexity of the C++ language and continuing instability of its implementations, this doesn’t result in a stable language semantics.

Initially, our planned feature set could be stated as “ML with explicit unboxing, exceptions, and first-class mutability.” That plan didn’t last very long.

### 5.2 Type Classes

One reason that that ML inference system works well subjectively is that the language does not include multiple integral or floating point types. There is a kludge in ML to disambiguate int vs. float by fiat. In O’Caml, the same problem is resolved by using different operators for integer and floating-point operations. Neither approach fares well when more ground types are added to the language’s number system. We needed the operator “+” to be able to operate over arbitrary integer types. Haskell presented a ready-made solution: type classes. These simultaneously allowed us to generalize operators and introduce a form of overloading into the language.

But type classes raise a problem that we will come back to later. They introduce overloading and matching ambiguities that need to be addressed: given two valid choices of specialization, which to choose? In consequence, they have an unfortunate tendancy to break separate compilation schemes. Another issue with type classes is that the major implementations all violate our code transparency objective. The dictionary-based implementation technique is not well-suited to languages having value types of multiple sizes; we wanted an implementation that operated more in the style of C++ templates — not least because of the importance of linkage compatibility with C.

This would turn out to be the most problematic feature of the language.

### 5.3 Looking Forward

Two other issues would emerge to catch us by surprise. The first was initializtion, and the second was existential types.

The existential type issue is one that we recognized at the beginning. In operating systems, it is common to wish to store a (pointer, procedure) pair, where the procedure expects that pointer and it is nobody else’s business what the pointer points to. This is a problem of existential type, and we assumed initially that a full existential type system would be needed. That proved to be quite invasive. Ironically (and especially so given our initial rejection of it), introducing C++-like objects provided the functionality we required without explosive new complexity. Eventually, we relented on objects for this reason.

One important thing that we did not study carefully enough early on was the problem of initialization order and dependencies (Section 9.1). This would later force us to introduce effect types into the language.

## 6 Reading BitC Code

Before turning to issues and examples, it may be helpful to present a brief synopsis of the BitC syntax. BitC is a Scheme-like [14, 28] language. Top level definitions are introduced by `define`, the language is higher-order, and procedure application is written in the customary LISP prefix style:

```
(define c #\c)  ; c is a character

(define (make-adder x)
  (lambda (y) (+ x y)))
```

The lambda convenience syntax for `define` is not merely convenience syntax as in Scheme; it introduces recursive procedure defintions. BitC function types are n-ary; argument arity is part of function type. We initially adopted this approach in the interest of C compatibility. It would later become apparent that this wasn’t necessary, but the use-case checking advantages of n-ary functions have led us to retain this decision.

Recursive data defintions are disallowed by BitC’s symbol resolution rules. BitC is strongly typed, let-polymorphic, and implements Haskell-style type classes [11]. The type of `make-adder`, above, is:

```
(forall ((Arith ’a))
  (pure fn (’a) (pure fn (’a) ’a)))
```

Strong typing means that BitC occasionally requires type annotations:

```
(define i (+ 1 1)) ;;OK
i: (forall ((IntType ’a)) ’a)

;; Type error: mutable locations must
;; have a single type
(define i:(mutable ’a) (+ 1 1))

;; OK -- annotation disambiguates
(define mi:(mutable ’a) (+ 1 1:int32))
```

Annotations are usually required only where state and numeric literals appear in combination, or in interfaces where declarations of external symbols are given.

Local bindings are introduced by `let` or `letrec`:

```
(define (id-with-let x)
  (let ((tmp x))
    tmp))
id-with-let: (pure-fn (’a) ’a)
```

An effect type system is used to type pure vs. impure computation:

```
(define (map f lst)
  (case l list
    (nil nil)
    (cons (cons (f l.car)
                (map f l.cdr)))))
map: (’%e fn ((’%e fn (’a) ’b) (list ’a))
             (list ’b))
```

The `case` construct dispatches over union leg types. Thankfully, most functions are definitively pure or impure, with the result that effect variables do not often make an appearance.

Structures and unions are introduced by `defstruct` and `defunion`:

```
(defunion (list ’a)
  nil
  (cons car: ’a cdr: (list ’a)))

;; :val indicates an unboxed type.
(defstruct (pair ’a ’b) :val
  fst: ’a
  snd: ’b)
```

Additional syntax supporting the definition and instantiation of type classes and modules. These should be self-explanatory as we encounter them.

# 7 Polymorphism and State

Like ML and Haskell, BitC is a let-polymorphic language. This is why procedures like `add-one` and `map` can be generic: each is specialized at need at each point where it is called. The `map` procedure can be called twice in a row with arguments whose types are unrelated from one invocation to the next. As long as all of the consistency requirements expressed in the type of `map` are observed, each of these calls is okay, and neither call has any impact on the other.

Another way to say this is that a binding in a let-polymorphic language is usually a term (in the sense of formal logic) that can be textually replaced by its defining expression wherever it appears:

```
(let ((x 1))
   (+ x x))  =>subst
(let ()
  (+ 1 1))  =>simp
(+ 1 1)
```

Each of these replacements is independent, and each has an independently decided type:

```
(let ((Nil nil))
   ;; Nil has type (list char):
   (cons #\c Nil)
   ;; Nil has type (list string):
   (cons "abc" Nil))
```

This is all quite beautiful from the perspective of formal language design, because term substitution is something that we know how to reason about formally and precisely. Unfortunately, term substitution doesn’t work so well if the binding is assignable (that is: it is a variable), and assignment is something that systems programmers would find very difficult to do without.

## 7.1 Perils of Mutation

Variables and polymorphism do not get along. If assignable values are permitted to be polymorphic, the result of the following expression is four, not five:

```
(let ((i 1))
   (set! i (+ i 1:int32))
   (set! i (+ i 3:int64))
   i:int64)
```

What has happened here is that `i` has been instantiated twice, once with type `int32` and the second time with type `int64`. It is exactly as if two completely different variables were declared, both having the same name. This, of course, is very confusing. The solution adopted in ML and BitC is to impose something called the *value restriction*. Any modifiable variable is considered to name a value (as opposed to a term), and a value may only be given a single type within any given execution of its containing procedure. In BitC, we also require that top-level (mutable) values must have a single type over the whole program, which is why the definition of `mi` in Section 6 required a type annotation. BitC has multiple integer types, so the compiler could not decide a unique type automatically.

The problem here is that we want the type inference mechanism in BitC to produce types that are so-called *principal types* or *most general types*. We would also like to have a type inference algorithm that is both sound (no invalid programs are accepted) and complete (every valid program is accepted). The problem with the assignable let binding above is that we cannot yet see at the point of binding whether `i` will later be assigned, and we will need to make some type decisions that depend on the type of `i` before we discover that `i` is assigned.

In this particular case, it is easy to detect syntactically that `i` is assigned by observing that it appears as a target of `set!`. The first implementation of BitC worked in exactly this way, giving `i` a polymorphic (therefore non-assignable) type in all other cases. Unfortunately, this scheme is sound but not complete. To see why, we only need to modify the example very slightly:

```
(let ((p (pair 1 2)))
   (set! p.first 3))
```

In this example, `p` will not be detected as assignable by our trick. It will be given a polymorphic type, and when the `set!` is discovered the compiler will declare a type error. This particular counter-example can be finessed with a bit of thuggery, but more general cases cannot.

Reviewers of the early BitC papers felt that this was a dodgy hack (which it was). The strength of their distaste was enough to conclude that the first pragmatically viable combination of polymorphism, type inference, and state didn’t warrant publication. Having been burned by Shapiro’s dodgy hack, Sridhar went back to his graduate student office to see if he could come up with a general solution.

## 7.2 Maybe Types

Sridhar’s first approach was to introduce something he called a “maybe type.” The concept is simple enough: when we see something like `i` introduced, assign it a form of union type. Since it has to be compatible with the integer literal 1 it must be constrained by `(IntType ’a)`. Beyond that, we will simply declare that it is either an immutable polymorphic identifier or it is a mutable monomorphic value binding – a decision to be resolved later. If a `set!` is discovered within the `let` body that modifies an object of a given maybe type, the maybe type is resolved to mutable. If we get to the end of the `let` binding form (that is: before the body is considered) without coming to any definitive conclusion that `i` must be mutable, then no assignment can have occurred while `i` is in scope, and we will “fix” the type to the polymorphic immutable choice, on the grounds that maximizing abstractness (therefore generality) is the right choice. This approach avoids the pitfalls of the `pair` example (and some other examples).

This approach is also incomplete. Given:

```
(define (id x) x)
(let ((x 1))
  ((id (lambda ()
      (set! x 2))) ))
```

it cannot determine that `x` should be mutable. The assignment within the `lambda` is effectively shielded from view.

Reviewers greeted this variant with about the same enthusiasm as the original. In academia, dodgy hacks do not pay.

## 7.3 Mutability Polymorphism

The current scheme [30, 31], also due to Sridhar, extends maybe types to a general form. Given: `(let ((x e))...)`, where *e* has type `’a`, it initially assigns `x` the union type:

```
(forall ((CopyCompat ’a ’b))
  ’a || (mutable ’b))
```

Which means: "if we decide that `x` is a term, then it is substitutable, and its type is exactly the type of the expression *e*. If we decide that `x` was required to be mutable, then it is some mutable type `(mutable 'b)` satisfying the requirement that a location of type `(mutable 'b)` can legally be assigned by copy from a value of type `'a`."

The problem with this assigned type is that it is unsound. If the type inference engine does not ensure that all of these maybe types converge to one answer or the other, the program as a whole may be accepted without being valid. Showing that our type rules do, in all cases, generate a unique decision is the subject of a pair of papers containing a depressing amount of dense mathematics. [30, 31] Such papers are not unusual in the programming language theory world, and they are the fodder of which programming languages dissertations are constructed. As system builders, we would have been amply content with the incomplete but pragmatically workable solution.

Given our verification objectives, this wasn't really an option. It is surprisingly easy for these "quick fix" solutions to come back to haunt you later. As new features and/or checks get added to a type system, you may discover at some point that you really needed a mathematically consistent answer in order to achieve a later goal. This would eventually happen to us when the time came to re-visit the meaning of `mutable` and `const`, which we hadn't gotten quite right. Through their insistence on a complete type system, the reviewers delayed the first publication on BitC and the project as a whole by almost two years. The combination of stubborn insistence from the reviewers with stubborn persistence on the part of Sridhar *did* result in a better language, and *did* allow us to repair two fairly bad problems later almost as quickly as we discovered them.

The mutability inference scheme in BitC today is both sound and complete. A side effect of our struggle to achieve this is a proper formalization of the core BitC type system. This will eventually make formal reasoning about the full type system much simpler.

### 7.4 Copy Compatibility

General mutability introduces a small but interesting wrinkle into the semantics of the language: copy compatibility. Values are copied at initializers, argument passing, and procedure return, so it is not necessary that they agree fully about their mutability. It is perfectly valid to initialize an immutable `int32` value from a mutable location of type `int32`. In fact, such an initialization can ignore mutability *up to reference boundaries* for purposes of determining assignment or initialization compatibility.

We were initially concerned that this would weaken type inference, because most of the program points where ML and Haskell can simply unify types are points that the BitC inference scheme must handle explicitly. In practice, we have not found this to be the case. The only issue we have found where inference failure becomes intrusive is in inferring the type of literals. The literal `1` may be any of eight types. When it appears alone, the compiler cannot tell which.

## 8 Mutability and Aliasing

It didn't take us long to hit an irritation. Most systems languages provide some form of elective call-by-reference. When used correctly, this significantly reduces run-time data copy overheads, which are one of the primary sources of performance loss in high-performance systems codes. Call-by-reference becomes particularly important in the presence of separate compilation, where the compiler may not be able to do interprocedural analysis in order to optimize out the normal requirements of the calling convention. This is particularly true for dynamic libraries, where the calling convention alleged at the language-level declaration is prescriptive. The implementation *must* match the declared interface.

In C, it is possible to have simultaneously a pointer to constant X and a pointer to X, both of which reference the same location. This means that the compiler must be conservative when procedure calls are made. In general, it cannot assume that supposedly constant objects are unable to change. In the interests of optimization, the ANSI C standard [1] actually permits the compiler to make exactly this assumption. Since there is no way for a C programmer to check whether they have complied with the expectation of the compiler, this is an invitation to error. Since the result of computation depends on the optimization decisions of particular language implementations, bugs of this sort cannot be reliably eliminated through testing.

In BitC, we decided very early to implement the notion of *immutability* rather then *constantness*. A BitC location that is declared to have immutable type cannot be modified through *any* reference. In C, the "const" in `const char` is a *type qualifier* that can be stripped away. In BitC it is an integral part of the type. Actually, there wasn't any `const` type constructor in early BitC. Variables and fields were constant unless declared mutable.

It is difficult to construct a sound type system in which *both* mutable and const declarations are possible. What does:

```
(mutable (pair (const int32) (const char)))
```

actually mean? Is this a structure that can be modified as a whole but cannot be modified at any particular constituent field? The answer in BitC was initially ”yes”. How about:

```
(const (pair (mutable int32) (mutable char)))
```

This is a pair that is mutable at every field but cannot be mutated as a whole. Both interpretations, of course, are complete nonsense, but they are mathematically consistent, and we decided at first to live with them. The problem is that *mutable* and *const* can be viewed as positive and negative constraints.

In type inference systems, bad things can happen when positive and negative constraints interact. The problem, at root, is that if you have both `const` and `mutable`, then:

```
    int32
    (const int32)
    (mutable int32)
```

mean three different things, and you have to decide what things like:

```
    (mutable (const char))
    (const (mutable char))
    (const (const char))
    (mutable (mutable char))
```

mean. For concrete cases this isn’t so bad, but for things like `(mutable (const ’a))` where the type variable will only be resolved later, matters become sticky rather quickly. Until our sound and complete inference strategy emerged, it wasn’t obvious how to resolve these properly, and we decided to leave well enough alone by omitting `const` from the language. Introducing `by-ref` ultimately changed our minds.

### 8.1 BY-REF

We had flirted with a fairly general pointer concept at various points, but internal pointers are problematic. There really isn’t a good way to give them the same representation that they have in C, and if you allow pointers into the stack you get into a requirement for full region typing to preserve safety, which has been a source of overwhelming complexity in the eyes of many Cyclone [9] programmers. We decided to keep our pointers pointing into the heap where they belonged, but the absence of by-reference parameter passing created this horrible itching sensation. You really cannot do a good job with object initializers or account for the typing of `set!` without them.

Thankfully, the particular type of region scoping introduced by `by-ref` is safe so long as `by-ref` can appear only on function parameters. Without much thought, we added `by-ref` into the BitC implementation, with the rule that the type of the reference had to strictly match the type of the thing that it aliased. So obviously:

```
(define id-by-ref (x: (by-ref ’a)) x)
id-by-ref: (fn ((by-ref ’a)) ’a)

(define p (mutable (pair 1:int32 #\c))
p : (mutable (pair int32 char))

(id-by-ref p.first)
1: int32
```

Right?

Wrong. While the `int32` field of the pair is indeed constant, the *location* occupied by that `int32` is not, and the type `(by-ref int32)` is a contract that the location will remain unchanged. Because of this contract:

```
(define (bad x)
  (set! p (5, #\d))
  x)
id-by-ref: (fn ((by-ref ’a)) ’a)

p.first
1: int32  ;; meaning (const int32)

(bad p.first)
1: int32  ;; permitted, but nonsensical

p.first
5: int32  ;; oops!
```

### 8.2 Const Dominates Mutable, Right?

In order to get this sort of thing right, we need to deal with how mutability and constantness compose. The simple and sensible answer is that a mutable thing is only mutable where its constituent parts are mutable, and consequently is only mutable as a whole if *all* of its constituents are mutable. Under this interpretation, the assignment performed in `bad` is illegal. So far so good, but if we flip the example around:

```
(define p (pair 1:(mutable int32) #\c))
p : (pair (mutable int32) char)

(define set-arg (arg:(by-ref ’a) val:’a)
   (set! arg val))
set-arg: (forall ((copy-compat ’a ’b))
           (fn ((by-ref (mutable ’a)) ’b) ())

(set-arg p.first 3) ;; oops!
```

We need to ensure that as addressing paths are traversed, constantness will dominate over mutability. But this is very annoying, because in:

```
(defstruct (pair ’a ’b):val
  first: ’a
  second: ’b)
```

writing `(mutable (pair int32 char))` will get us something that isn’t mutable *anywhere*, because the field types are constant. Programmers will soon start writing `mutable` on all field types, which will tend to make state promiscuous and simultaneously defeat our desire to exploit abstraction over types. If `pair` is defined as:

```
(defstruct (mutpair ’a ’b):val
  first: (mutable ’a)
  second: (mutable ’b))
```

things are not so bad, because

```
(define p (mutpair 1 2))
```

remains constant everywhere, and therefore polymorphic. Unfortunately, the same is not true for `refpair`:

```
(defstruct (refpair ’a ’b):ref
  first: (mutable ’a)
  second: (mutable ’b))

(define rp (mutable (refpair 1 2)))
```

Because what the programmer is getting for `rp` is *mutable reference to constant aggregate containing mutable fields dominated by the constantness of the aggregate*. If constant-ness dominates mutability naively, no reference type can ever contain a mutable field!

That problem has an apparently obvious fix: perhaps an aggregate should be considered mutable exactly if all of its fields are mutable. But then a variable of type `mutpair` can *never* be immutable, and therefore can never be polymorphic. Somehow not quite what we had in mind. We could strip that mutability away again if we had a `const` type constructor, but positive and negative constraints generally don’t combine well in type inference systems. Drat! That const vs. mutable dilemma really needs a solution. Forcing programmers to write `mutable` at all fields violates usability in the eyes of dysfunctional — er, um — systems programmers. In the eyes of the functional programming camp, that sometimes seems to make us a bunch of stateists and deviants.

Thankfully, those pesky reviewers had forced us to a sound and complete inference system. with the result that reconciling all of this was ultimately straightforward.

*Sketch HOW.*

# 9 Modules and Initialization

One area where we were forced to make language design decisions was in the choice of a module system. Large software systems require multiple source units for maintainability. Modules are tied to this requirement, but they remain an area of active investigation in the research literature, so there was no well-established solution to use.

The C approach was certainly out. Textual inclusion is about as elegant as Perl syntax (and can be implemented therewith). The ML module system [18] is fully understood only by David MacQueen, and only on alternating weeks. The Scheme module system [8] required the combined brilliance of Matt Flatt and Matthias Felleisen (and six revisions of the language standard) to achieve. From the perspective of a mere operating systems weenie, it’s all rather discouraging, but what is the reluctant programming language designer to do? We absolutely needed some form of separate compilation.

Actually, designing a module system *per se* isn’t that hard. You have import, export, and compilation units. It’s convenient if interface units bear some clear relationship to things you can find in the file system, but really that’s not so hard. The problems with module systems in stateful languages are all about global variable initialization order. Java ducks this by eliminating both non-trivial and mutable global variables. Which works, but it is one of the reasons that Java isn’t terribly well suited to systems programming. One thing does seem clear: using things before you define them is problematic in a safe programming language.

There are two language features that create problems for initialization order: external declarations and assignment. External declarations intentionally allow global identifiers to be used before they are defined in ways that cannot be checked when modules are compiled separately. Assignment can cause initialized values to change between uses. This either leads to a much finer dependency ordering on initializers or to throwing up your hands and leaving ordering of effects during initialization undefined. That may be type safe, but you get no guarantee about what values global variables may hold when `main()` begins to run. From a verification perspective this is problematic. It is also makes the operation of a macro system (which we intend to add in the future) undefined.

Well-formed programs invariably satisfy an implicit lattice-structured dependency relationship for their initializers (or at least don’t get caught violating one). The problem is that the ordering built by the compiler is dictated by computational dependency, while the organization of the program required for manageability (the module relationships) is dictated by conceptual relationships that exist in the minds of the developer. The problem of initialization order is to satisfy both constraints simultaneously.

### 9.1 Imposing an Initialization Ordering

In BitC, we solved the initialization ordering problem by declaring that interfaces are initialized first according to the lattice defined by their import dependencies. Within an interface, initialization proceeds from top to bottom, and use of undefined forward references is prohibited. It is source units of compilation that present the problem.

A BitC interface can declare a procedure that is implemented by some exporting source unit of compilation. In the absence of this feature, we could declare that source units got initialized last in unspecified order. Because source units can only export their identifiers through interfaces, there cannot be further ordering dependencies once the interface units have initialized in a defined order.

In the presence of export, this doesn't quite work. What we do instead is to require that any definition that is exported by a source unit may rely (transitively) only on those symbols that were in scope at the point of declaration *in the interface*. The provided definition can rely on other procedures and variables in the source unit, and initialization proceeds as if all of those procedures and variables had been temporarily copied into the interface unit of compilation.

But what about assignment and side effects? And what if some of the procedures executed during initialization get run multiple times?

### 9.2 Initialization vs. State

For reasons previously discussed, permitting assignment during initialization is problematic. Empirically, it isn't enough to require that initializers be designed sensibly by the programmer — in the presence of shared libraries this requirement is nearly unachievable, and the problem becomes more challenging in a higher-order programming language. We cannot preclude the use of `set!` in initializers altogether. This is too strong; it would prevent *any* occurrence of `set!`, even if the containing procedure is not called at initialization time. And strictly speaking, we *can* allow assignments in initializers as long as the results of those assignments do not escape.

The position taken in BitC is that initializers must be "pure," meaning that they may not modify any globally reachable values. Implementing this forced us, with considerable hesitation, to adopt an effect type system (Section 10).

### 9.3 Interaction with Macros

The initialization ordering problem also interacts with any later decision to add a macro system. Macro execution happens at initialization time, and it is necessary to define the initialization environment within which a macro executes. In particular, it must be defined which previously initialized variables the macro can legally reference.

In fact, the problem is a bit more constrained even than that, because it must be possible to reference those values *at compile time*. A corrolary to this is that *all* initializations must be computable at compile time. Pure initializers do meet this requirement.

## 10 Effect Typing

The last foundational addition to BitC was the addition of an effect type system. Effect typing is closely related to region typing, which we had consciously avoided after experiencing it in Cyclone [9]. Region types are powerful when used correctly, but they impose a significant burden on the programmer, clutter type declarations that really need to be clear to the reader, and increase the difficulty of declaring cross-language interfaces correctly. We have chosen to adopt a very restricted form of effect type, and we remain concerned that this may have pushed BitC beyond the boundary of usability. Time will tell.

### 10.1 Effect Types in BitC

The idea behind the BitC effect type system is simple. We associate with every function type and every expression an effect type variable (which I will hereafter call an effect variable) which can take on three states: *pure*, *impure*, and *unfixed*. If a function performs an assignment, its effect variable is fixed to "impure." If function `f` is called in some expresion, the expression is pure exactly if the function is pure. If the execution of function `f` in turn calls function `g`, and `g` is impure, then `f` is impure. And so forth.

In most use-cases, effect systems are not invasive, but complexities arise where procedures are passed or returned as parameters. In a procedure like `map`, the best we can say is:

```
map: ('%e fn (('%e fn ('a) 'b) (list 'a))
             (list 'b))
```

which means: "the purity of an application of `map` depends on the purity of some unknown procedure `f` that will be passed to `map` at the point of application."

Matters can become even more convoluted. Fortunately, it is rare for this to become visible to the BitC programmer. The reason for this is that BitC has intentionally adopted a limited use of effect systems. The only questions we can ask of our effect system are “pure” or “impure,” which tends to limit the exposure of effect variables. The biggest place where they become visible is at typeclass usage. Thankfully, most of the core type classes of BitC require that their method implementations be pure.

We are still working on a satisfactory syntax for this in the general case.

## 10.2 Other Benefits

One benefit of effect types is that we can preserve a notion of pure subprograms in BitC, and we can express this in the BitC type system. It has allowed us to introduce a new expression into the language:

```
(pure expr)
```

whose effect is to require that the computation of *`expr`* be effect-free. A programmer wishing to ensure that their entire program uses only pure constructs need only write their main procedure as:

```
(define (bitc.main argc argv)
  (pure body))
```

This ensures that the entire execution is stateless. A corollary is that it is possible to express the constraint that input data is deeply immutable. The two constructs taken together admit a surprising degree of opportunity for parallel execution. In particular, computation over deeply immutable data structures need not consider any impact from aliasing, and pure computation over impure data structures need not consider aliasing if impure computation is not permitted simultaneously.

## 10.3 Concernes about Effect Types

One concern about the introduction of effect types into BitC is that it is syntactically confusing. The typing of `map`, for example, is visually complicated. While LISP-style syntax certainly is not helping, the experiences of languages providing region types suggest that surface syntax is not the real issue here. To the developer, the meaning of

```
(pure fn (’a) ’b)
(impure fn (’a) ’b)
```

are clear, but the meaning of:

```
map: (’%e fn ((’%e fn (’a) ’b) (list ’a))
             (list ’b))
```

is less so. Too much is going on here in too many domains at once. If these types occur frequently, and particularly if they appear in type classes (where the effect variable gets lifted into the type class type), they may make the language difficult to use.

For systems programmers, who are coming from languages with less interesting type systems, this raises adoption concerns. For this reason, we are considering defining

```
(fn (’a) ’b)
```

to mean

`(fn ’%e`$_{anon}$` (’a) ’b)`

that is: a function whose effect variable is not related to any other effect variable and therefore need not be named explicitly. It remains to be seen whether this will help or hinder clarity.

# 11 Existential Types vs. Objects

Our original plan for BitC called for existential types. The motivating use case for this is driver interfaces, which require a degree of existential encapsulation. In particular, the driver publishes a pointer to its state and a collection of procedures that operate over that state. Except where this pointer is passed back to the procedures, its type is opaque. This is the same type of encapsulation that is accomplished by closures, but closure construction requires dynamic storage allocation where the existential construction does not.

Having hit the effect type roadblock, and having seen the impact of region types in general, we were very reluctant to add generalized existential types to the language. The problem with existential types is that they tend to escape into the type system in exactly the kind of way that makes types unreadable. After a lot of thought, we concluded that all of the use-cases we could identify for existential types could be satisfied very well by a C++-style object construct. This construct introduces a restricted form of existential type that does not escape too visibly into the rest of the type system, and which is familiar to users already.

Somewhat to our surprise, the object concept dropped into the language very nicely, requiring only the introduction of a `method` type that is essentially similar to, but not copy compatible with, functions in general.

# 12 The Price of Polymorphism

The implementation of parametric types in BitC raises several issues. Parametric types, explicit unboxing, and separate compilation do not get along.

## 12.1 Implementation in ML and Haskell

**Parametricity in ML** Parametric types in ML place only simple requirements on the implementation. If ML assigns a parametric type, we already know that the representation of that type's representation occupies a single machine word and that no type-dependent procedure call will be made on that type. ML's record polymorphism complicates this slightly. Record polymorphism can be viewed as a form of type class mechanism (abstraction over "."), but it induces parameterization over fixed-precision integers (structure offsets) rather than over variant type sizes. The compiler never needs to generate code for a single procedure more than once, because structure offsets can be re-encoded as hidden parameters.

**Parametricity in Haskell** Matters in Haskell are slightly more complicated, because type classes introduce unresolved procedure references. The customary implementation method is to pass "dictionary" pointers as hidden parameters [21], this is not really required if units of compilation boundaries can be violated by the compiler [12]. Note, however, that a satisfactory implementation of type classes in Haskell relies on whole-program optimization.

**Parametricity in BitC** BitC has essentially the same problem that Haskell suffers. But in BitC we adopted the view that it *must* be possible for an implementation to preserve separate compilation and optimization *without* introducing hidden parameters. Hidden parameters violate C linkage and carry fairly a large performance burden. Where whole-program information exists we are not opposed to using it. Our current research compiler uses demand-driven whole-program polyinstantiation to avoid any need for hidden parameters. What we want to avoid is *requiring* a whole-program approach, because that does not scale to multi-million line systems.

If whole-program or whole-system code generation is performed, as seems likely in deeply embedded systems, the inlining issue evaporates at the cost of higher compile delays. These can be often be off-loaded by a sophisticated programming environment. When run-time code generation is used, as in CLR or JVM, a similar sort of whole-program view exists. BitC seeks to preserve this implementation option, but in high-confidence contexts we have serious reservations about assuring the dynamic code generator. The approach sketched below has not yet been tested.

## 12.2 Instantiation by Cloning

The presence of unresolved structure offsets does not require link-time or run-time code generation, and for the most part need not impede optimization. With fairly modest additions to the relocation information available to the linker, it is possible to fully optimize procedures that use record polymorphism while still viewing them as *templates*. At link time, we can simply clone the template procedure and resolve the offsets as needed. Sophisticated re-optimization and code generation need not be done. Inlining remains possible in the absence of whole-program compilation, but only if the *definition* of the procedure being inlined is visible at static compile time.

Where inlining and separate compilation are simultaneously required, the simplest strategy in BitC is to migrate procedures that need to be inlined into interface units of compilation, making their definitions accessable to the compiler. In practice, good candidates for inlining are relatively small and simple. Some degree of type and code encapsulation is lost in this practice, but it is certainly no worse than the corresponding practices in C and C++ today.

Type classes and parameter sizes can be handled by link-time cloning similarly. Both unresolved procedure references and type sizes can be viewed as problems of literal instantiation that need not interact with optimization, register allocation, instruction selection (ignoring spills) and the like. Both can be resolved by demand-driven link-time cloning and constant reference resolution. A mixed strategy is also possible in which selected procedure template instantiations are done at static link time and others are deferred to the dynamic linker.

## 12.3 Other Type Class Issues

The original specification of type class instantiation in BitC followed Haskell, in which instances violate lexical scoping. Haskell requires that type class instances be globally non-overlapping w.r.t. the types that they match, and this introduces a requirement for whole-program checking at link time. Because Haskell objects are boxed, this requires a consistency check rather than full link-time code generation. But later versions of Haskell have introduced various mechanisms for specializtion of type classes. These *do* require whole-program compilation, because the correct choice cannot be made without a whole-program view.

In BitC, we ultimately concluded that type class instances should be resolved in the lexical scope at which specialization occurs. This preserves separate compilation (modulo template cloning with literal instantiation), but at the cost that type class specialization becomes context-sensitive.

# 13 Other Thoughts

There are a couple of questions that come up repeatedly as we discuss BitC.

## 13.1 State vs. Functional Programming

From the programming language community, we are often informed that we didn't fully understand the I/O monad, or we are asked whether state is truly necessary. To some degree this question is a matter of theology, but state is unquestionably awkward from a verification standpoint.

We have not "drunk the cool-aid" concerning pure programming and multiprocessing, because the "join" operation in real multiprocessing applications is inherently stateful and serializing (therefore eager). A careful mix of stateful and stateless idioms is required in that domain, and the `pure` construct offered by BitC's effect type system seems to provide exactly the mix required.

Whether our surmise about multiprocessing is correct or not, the origins and focus of BitC lie in core ”systems” applications such as operating systems and databases. While pure programming has a significant role in producing more robust versions of these kinds of systems, they are *entirely* about state, and the proposition that state can be removed from a language designed to support these systems is something like the proposition that eliminating bones is the solution to bone disease.

One of the true advantages of Haskell is that as a research language it is possible for Haskell to do "one thing well" as Simon Peyton-Jones has often noted. Simon is unduly modest; Haskell does one thing *superbly*. The challenge of languages designed for production use is that they must do *many* things well, and in consequence they do not usually have the luxury of adopting strongly polarizing design positions. In BitC, we stuck to our position on type safety, but felt that in our application domain a strict approach to purity was not an option. On the other hand, this straddling position may make BitC a useful language for transitioning incrementally to less stateful idioms.

## 13.2 Pointers and Addresses

From systems programmers, we are often asked whether the absence of any `address-of` operation in BitC isn't a significant impediment to systems programming. The answer is "no." In fact, if you examine well-structured systems code written in C, you will discover that the use of `address-of` occurs in three cases: array indices (where the actual indices can be used directly), locking and chaining constructs (where the address is being taken mainly because neither subtyping nor abstraction is available), or by-reference parameter passing (which BitC supports directly). In fact, the only pointer idioms that BitC truly restricts are pointers to stack-allocated objects.

Some readers may object that our statement here is inaccurate, pointing to constructs in Linux that violate our claim. Our response is that Linux isn't a well-structured system. BitC is a tool for building well-structured systems, and it isn't the purpose of BitC to transcode C in general. In spite of this, it is surprising just how much C code *can* be transcoded directly. While programmers are not supported by C in their quest for type safety, their pragmatic desire for safety tends to push them into idioms whose correct typing can be successfully inferred and re-represented directly.

There *is* a case in Coyotos where we are forced to downcast a pointer in the current implementation. We can show that this case is safe, but the type system doesn't presently see that. We currently handle this using a dynamic cast construct, though an alternative implementation would avoid the problem entirely.

## 13.3 Garbage Collection

We are frequently asked whether garbage collection (GC) and systems programs are compatible notions. The answer to this is unequivocally *yes*. The most critical systems codes use *compile time* allocation; no GC is required in such systems. The next layer of system up uses type-stable heaps. Safety does not require GC in such systems, and memory requirements in such designs are often straightforward to verify. The GCC compiler has recently switched to a conservative collector.

The area we see where reliance on GC remains problematic is in highly concurrent systems having large amounts of shared-access state. The problem here is that the GC subsystem runs concurrently in the most general sense, which imposes a severe burden on mutator performance. This is an area of ongoing research, and we expect that some combination of type-stable heaps, region allocation, and *background* collection may ultimately provide a reasonable approach. For the moment, we note that type-stable heaps combined with explicit deallocation remain type safe, and do not carry the costs typically associated with concurrent collection.

Setting that issue aside, we note that objections to garbage collection rarely take into account the quantifiable *economic* cost of manual storage management. The argument about performance is by now fairly clear. The overhead of GC appears in different places and with slightly higher variance. But the difference in *dollar* cost between the two techniques is persuasive. The overwhelming majority of penetration-based losses in 2007 were made possible by memory safety violations. The aggregate cost of those losses is approaching $20 *per PC*, which at today's prices is a noticeable fraction of total PC cost. If the question is framed as "Will you accept an increase in *variance* that is usually not noticeable in order to gain a 4% improvement in real costs?" the answer may well be "yes." While a cost-survivable position might be made for type-stable heaps with manual storage management, the cost of type-*unsafe* memory management is economically prohibitive.

### 13.4 Surface Syntax

The current BitC surface syntax is LISP-like. It grew in part out of BitC/ACL2. Our public claim was that the LISP ”list as AST” pun was very useful in a language where properties would be stated, but the truth is that designing a suitable property language syntax within a human-readable programming language is not really that hard. It is also complete nonsense because the ”list as AST” pun breaks down almost immediately when static typing is introduced into the language.

The real reason we stuck with the LISP-ish syntax is that we didn't want to be fiddling with the parser while designing the language, and (to a lesser degree) because we didn't want to encourage programmers until we thought the language was ready. We are now preparing a more sensible surface syntax, whereupon we will receive rotten egg and tomato complaints from the LISP community. You can't please everyone. Using the LISP syntax for so long let us focus on what was really important.

## 14 Future Work

While BitC set out to be a language with an accompanying prover infrastructure, the prover part of the task has not yet been tackled. A syntax for preconditions, postconditions, and property statement needs to be designed, and a verification condition generator for those statements needs to be created. All of that remains to be done, so few claims seem sustainable. Having a sound type system undoubtedly places us in a better position for static checking than C, but that is hardly unique to BitC. The language that probably offers the greatest pragmatic experience in this area is Eiffel. [19]. In contrast to Eiffel, the safety of the BitC type system offers some advantages, but demonstration of this remains a task for the future.

Though BitC is a safe language supporting efficient compilation, several of its features cannot be implemented within the managed CLR subset or the JVM. This suggests that enhancements to both may be warranted, particularly because a more direct and efficient form of run-time code generation might be enabled by doing so.

We initially avoided an interactive *read-eval-print* loop in Bitc. This was partly because such interfaces have often led in practice to languages that cannot be statically compiled. It was also because we wanted to avoid any inadvertent dependency on whole-program compilation in the design and implementation of the language. Today, an interactive interface would not be difficult to add, and we are preparing to do so.

We also need to explore the template cloning approach sketched in Section 12.2.

It remains too early to tell whether BitC will emerge as a useful and general systems programming language. We can say from experience and measurement that its performance and expressiveness are both up to the job. The question will be whether a language like BitC will be adopted in the face of object orienteerism and unfamiliar syntax.[3] Only time can answer that.

## 15 Conclusion

The design of the BitC language was an effort spanning four years of effort by three people, one Ph.D degree, nearly 42,000 lines of *surviving* code by two very experienced programmers, and many false starts. Only now are we getting to the point where selected users can begin to use the implementation, and thereby provide feedback on the language. Whether the effort has been modest seems dubious, but preposterous does not seem in serious doubt.

It is noteworthy that *none* of this effort was deemed fundable by the National Science Foundation (which is to say: by the academic programming languages community). The work is clearly innovative, and it is somewhat unusual to find a project of this sort that is accompanied by a validating experiment. Part of the problem is "crossover." Reviewers from the programming language community were unable to evalute our credentials as systems researchers. Another part of the problem is scale: the combined effort of BitC and Coyotos was too large to fund in any single grant or contract. Bootstrapping an effort of this magnitude under a single lead investigator is always a challenge.

Curiously, we have seen steadily increasing interest from the commercial world as the project has gained exposure beyond the research community. Senior executives

[3] The term "orienteerism" either is, or will be, a trademark of the Walt Disney Corporation.

at companies who rely on critical software systems seem to understand that developing in the systems languages of the 1960s carries a large cost, and that it may be time to review the state of the art.

For those with an interest in new programming language design and implementation, we encourage you to look at the source code of the BitC implementation. We are working over the next few months to make it documented and approachable. We are also working to produce notes that address some of the key design issues in the compiler. The process of design has been entirely open, which is moderately unusual.

**Acknowledgements** We have received comments, encouragement, and assistance from the many members of the `bitc-lang` mailing list and the *Lambda the Ultimate* blog. We would particularly like to thank Scott Smith of Johns Hopkins University, who was an active participant and advisor in the emergence of the BitC type system and its verification. Mark P. Jones of Portland State University provided several very patient discussion of type classes and their issues and consequences.

## References

[1] —: American National Standard for Information Systems, Programming Language C ANSI X3.159-1999, 2000.

[2] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[3] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. "The Spec# programming system: An overview." *Proc. CASSIS 2004: International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, published as *Lecture Notes in Computer Science*, pp. 49–60, vol. 3362, 2004.

[4] E. Biagioni, R. Harper, and P. Lee "A Network Protocol Stack in Standard ML" *Higher Order and Symbolic Computation, Vol.14, No.4*, 2001.

[5] H. Chen and J. S. Shapiro. Using Build-Integrated Static Checking to Preserve Correctness Invariants. *Proc. 2004 ACM Symposium on Computer and Communications Security*. Oct. 2004.

[6] H. Derby, "The Performance of FoxNet 2.0" *Technical Report CMU-CS-99-137* School of Computer Science, Carnegie Mellon University, June 1999.

[7] D. Dhurjati, S. Kowshik, and V. Adve. "SAFECode: Enforcing Alias Analysis for Weakly Typed Languages." *Proc. 2006 Conference on Programming Language Design and Implementation*. pp 144–157. Ottawa, Canada, 2006.

[8] Matthew Flatt and Matthias Felleisen. "Units: Cool modules for HOT languages." *Proc. ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pp. 236–248, ACM Press, 1998.

[9] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. "Region Based Memory Management in Cyclone." *Proc. SICPLAN '02 Conference on Progrmaming Language Design and Implementation*, Berlin, Germany, June 2002.

[10] T. Hallgren, M. P. Jones, R. Leslie, and Andrew Tolmach. "A Principled Approach to Operating System Construction." *Proc. 10th International Conference on Functional Programming (ICFP 2005)*. pp. 116–128. 2005.

[11] Mark Jones. "Type Classes With Functional Dependencies." *Proc. 9th European Symposium on Programming* (ESOP 2000). Berlin, Germany. March 2000. Springer-Verlag Lecture Notes in Computer Science 1782.

[12] Mark Jones. "Dictionary-Free Overloading by Partial Evaluation." *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. pp. 107–117. 2004

[13] M. Kaufmann, J. S. Moore. *Computer Aided Reasoning: An Approach*, Kluwer Academic Publishers, 2000.

[14] Richard Kelsey, William Clinger, and Jonathan Rees (Ed.) *Revised$^5$ Report on the Algorithmic Language Scheme*, ACM SIGPLAN Notices, 33(**9**), pp 26–76, 1998.

[15] Gerwin Klein, Michael Norrish, Kevin Elphinstone and Gernot Heiser. "Verifying a High-Performance Micro-Kernel." *7th Annual High-Confidence Software and Systems Conference*, Baltimore, MD, USA, May, 2007

[16] S. Kowshik, D. Dhurjati, and V. Adve. "Ensuring Code Safety Without Runtime Checks for Real-Time Control Systems." *Proc. 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. Grenoble, France. pp. 288–297. 2002.

[17] C. Lattner and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation." *Proc. 2004 International Symposium on Code Generation and Optimization.*, p. 75–, 2004.

[18] David MacQueen, “Modules for Standard ML.” *Proc. 1984 ACM Conference on LISP and Functional Programming*, pp. 198–207, 1984.

[19] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

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

[21] J. Peterson and M. P. Jones. “Implementing Haskell Type Classes.” *Proc. 1989 Glasgow Workshop on Functional Programming*, Glasgow, Scotland. pp. 266–286. 1989.

[22] S. L. Peyton Jones (ed.). *Haskell 98 Language and Libraries: The Revised report*. Cambridge University Press, 2003.

[23] S. L. Peyton Jones and P. Wadler “Imperative functional programming.” *Proc. ACM SIGPLAN Principles of Programming Languages.*, 1993

[24] Bjarne Stroustrup. *The Design and Evolution of C++* Addison-Wesley, 1994.

[25] 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.

[26] 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.

[27] Shapiro, J. S., Weber, S.: Verifying the EROS Confinement Mechanism. Proc. 2000 IEEE Symposium on Security and Privacy. May 2000. pp. 166–176. Oakland, CA, USA

[28] M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten, (Ed.) R. Kelsey, W. Clinger, J. Rees Richard Kelsey, William Clinger, and Jonathan Rees (Ed. 5th Edition) R. B. Findler, J. Matthews (Authors, Formal Semantics). *Revised*$^{6}$ *Report on the Algorithmic Language Scheme*, 26 September, 2007.

[29] Jonathan S. Shapiro. *A C++ Toolkit*. Prentice Hall, 1991.

[30] Swaroop Sridhar, Jonathan S. Shapiro, and Scott F. Smith. “Sound and Complete Type Inference for a Systems Programming Language.” *Proc. Sixth ASIAN Symposium on Programming Languages and Systems*, Bangalore, India, Dec 2008.

[31] Swaroop Sridhar, Jonathan S. Shapiro, and Scott F. Smith. “The BitC Type System and Inference Algorithm: Proofs of Soundness and Completeness.” *SRL Technical Report SRL2008-02*, July 2008.