BitC: A Modern Language for Systems Programming

Jonathan S. Shapiro
(PhD Penn CIS '99)

Assistant Research Professor
Department of Computer Science
Johns Hopkins University

Systems Research Laboratory works on a range of basic research in operating systems, programming languages, and software verification.

Managing Director
The EROS Group, LLC

Created to commercialize work on the Coyotos operating system. Has since broadened into high-robustness and high-confidence systems.

Loaded* Trivia Quiz Question

Name a *foundational* advance in systems programming languages in the last 30 years.

*After all, what good is an *unloaded* question?

Trivia Quiz Answer

Foundational advances in systems programming languages in the last 30 years: **none***.

In the last 40:

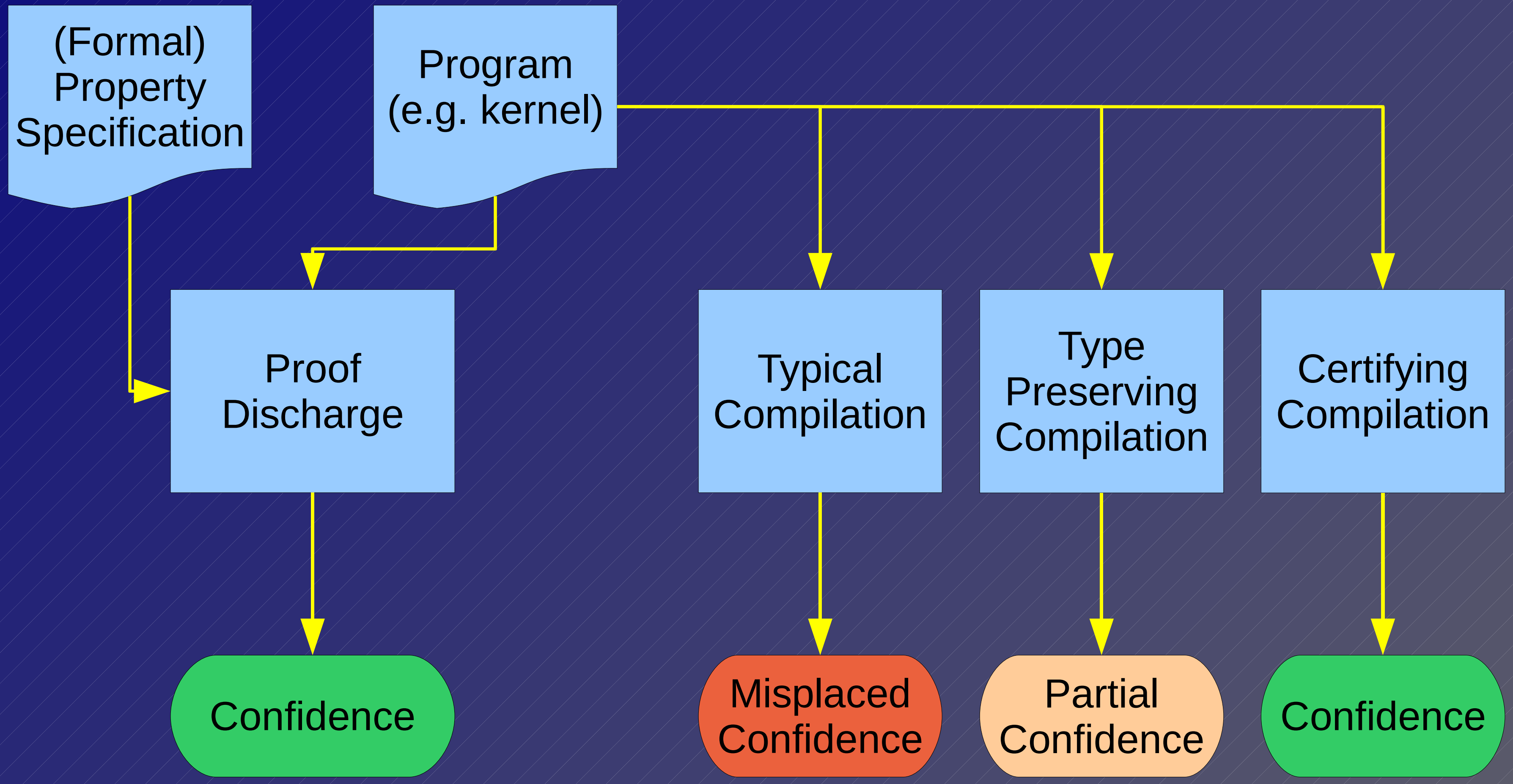
☑ C Programming Language, 1971
☑ Ada, 1975
☒ C++, 1983
... and?

*This is probably unfair to Modula-3

BitC Application Domain

- High-level concern: *repeatable, cost effective* assurance
- Automation relies on formal methods, ranging from type systems to software verification.
- Other goals: broadly higher confidence on general-purpose applications, kill off memory errors.

Solution Structure



Compiler complexity is *not* your friend!

Preconditions

- Programs must be (formally) *meaningful*.
- Can't give up low-level expressiveness to obtain that.
- Anybody who cares about security, robustness and writes in C needs their head examined.
- ... but where's the alternative?

Quiz #2

Name a language that is:

- Safe (not C, C++)
- Expressive about machine-level things (not OCaml, Haskell, C#, Java)
  - Representation, unboxing, state
- Has a defined semantics (not C#, Java, Cyclone)
- Supports *open* proof trails (not SPARK/Ada)
- Verification friendly (not C# => Spec#)
- Nice to have an appropriate mechanism of abstraction (not SPARK/Ada, C, C++, Java, C#, Spec#)

Perhaps...

- Something in the OCaml/Haskell family
- But require:
  - Bit-level structures, machine-level types
  - Non-discretionary unboxing (including unions)
  - First-class treatment of state
  - Allocation-free subprograms (=> no currying)
- This was the original statement of BitC.

Famous Last Words

"Well, that doesn't seem so hard..." (2004)
=> BitC, Sridhar, 2009

We thought this was an integration and follow-through problem.

Curiously, so did members of the PL community...

Reviewer Responses

- Unboxing is a compiler problem
  - Only when discretionary
- State is bad for you
  - How to implement kernel without it?
- This is all non-semantic
  - Who cares?
  - But wrong in any case
- Inference Isn't Important...
  - Users very bad at constraints
- ... but must be complete
  - Can't publish without that.

- It's just a constant factor
  - 100x matters.
  - Actually, 2x matters
- The optimizer can do it
  - When done, call me.
  - Aassurance?
  - Loss of transparency
- It's all been done
  - So where can I get it?
  - Prominent figure, but...
  - ...*his* stuff doesn't compile

Eye off the Ball

A vocal portion of the PL community has lost sight of the fact that ultimately *we build languages to build, validate, and maintain) artifacts*, not to test theories.

In consequence, work bringing good theory to useful systems practice is almost impossible to publish in PL venues.

In consequence, PL *engineering* is academic suicide, even when it involves hard problems.

This does not seem healthy.

About That Optimizer

- In matters of assurance, “clever” is a four letter word
- Pushing complexity from one place to another (like into the optimizer) isn't a win
- *Especially* when the resulting complexity is densely intertwined, impossible to assure, and hard to understand pragmatically

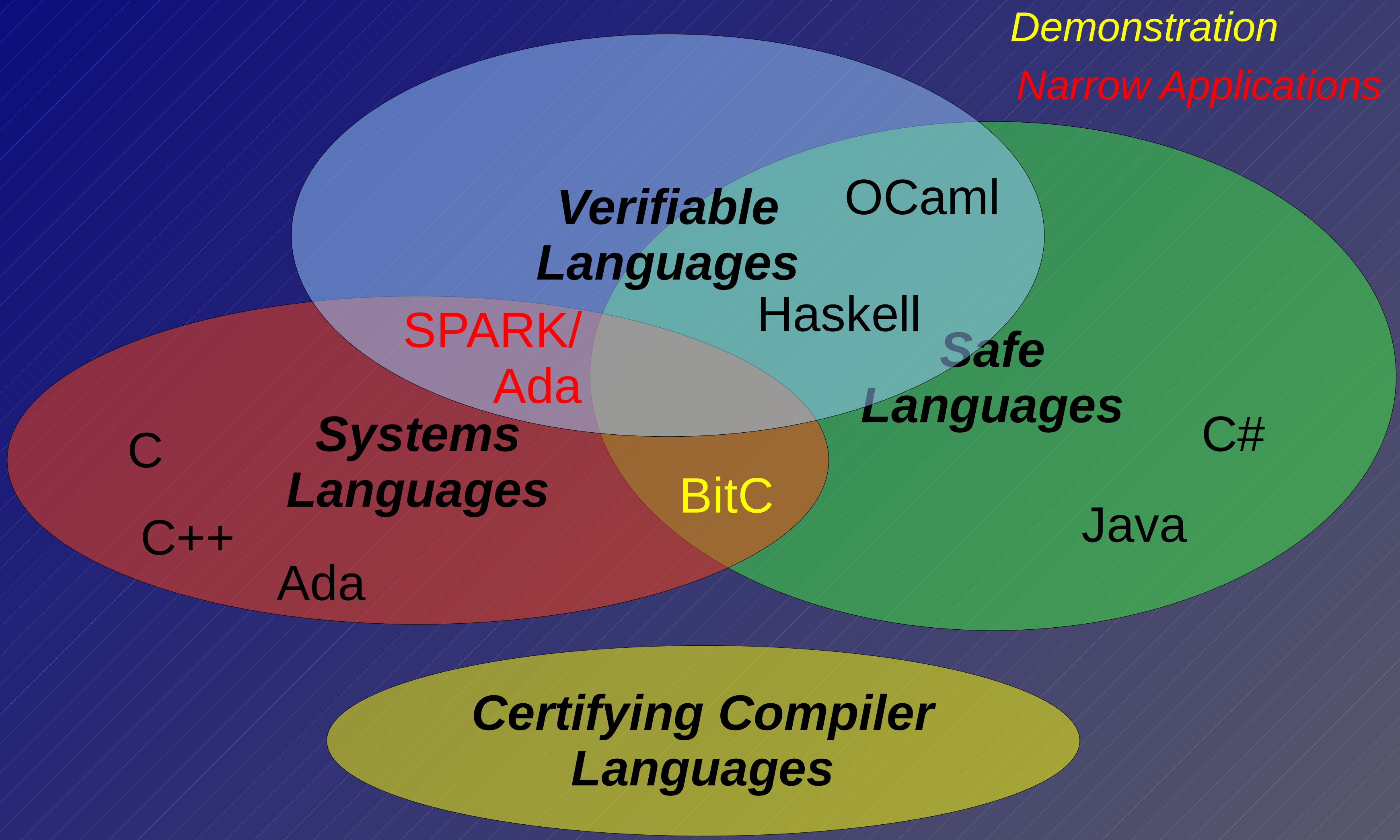
- The “moose under the rug” (Robin Hanson, Xanadu)

Other Responses

- Functional Programming!
  - Dysfunctional problems
  - But there are obviously important lessons here for proof
- OO Programming!
  - Bad model gone worse
  - Well, I share some blame

- Monads!
  - Compose poorly
  - Interleave issues
- Dependent Types!
  - This is attractive, but...
  - Some hard issues
  - Hybrid approach masks errors

PL Universe (Systems View)



What About SPARK/Ada

- 25+ years old (their prover is showing its age)
- Severe expressiveness limit
- How many Ada programmers do *you* know?
- Proprietary tool chain.
  - How can customers trust it?
  - How can customers self-maintain in emergency?
  - Correctness must be *seen* to be done

... So we decided to build a new language: BitC

Assurance and Lord Hewart

"...it is not merely of some importance but is of fundamental importance, that justice should not only be done, but should manifestly and undoubtedly be seen to be done."

Lord Chief Justice Hewart, 1923
Rex v Sussex Justices; Ex parte McCarthy

... In the long haul, BitC is about <u>open</u> and <u>independently reproducible</u> evidence of correctness as a basis for assurance

Integrate, Don't Invent. Hah!

- Prescriptive unboxing + polymorphism => ???
  - Polyinstantiation, but issues with that
- Lots of similar types pretty much forces us to *ad hoc* overloading => Type Classes => Constructor Classes
  - But dictionaries and unboxed aggregates don't get along
  - Dictionaries are a problem for performance in any case
- Sound and complete inference in the presence of general (on-stack) mutability (Sridhar 2008)
  - Inference matters (c.f. SVR4)
- Separate compilation, dynamic linking?
  - Important in large systems

Perils of “Obvious” Research

“Fast capability systems are doable” (1991)
=> EROS, 1999

“I can't see why verifying confinement ought to be difficult” (~1994)
=> SW Verification, Weber 2000

“Dynamic binary translation isn't that complicated” (2002)
=> HDTrans, Sridhar, 2005

“Automating the existing proof ought to be straightforward” (2002)
=> DSW Verification, Doerrie, 2009(?)

Moral

When shap looks innocent, students should either run like hell or dig in for the long haul.

(Also applies to junior faculty.)

BitC

Goals

- ☑ Higher-order programming language
- ☑ Modern type system in the style of ML or Haskell
- ☑ Sound and complete type inference.
  - This was hard
- ☑ First-class handling of state (with pure subset)
- ☑ First-class representation (unboxing, bit reprs)
- ☑ Translucent compilation (c.f. templates)
- ☑ Non-allocating subset
- ☑ Perform comparable to C
- ☑ Near-zero runtime support
- ☑ *Static* safety
  - Except divide, vectors
- ☒ Separate compilation
- ☐ Unambiguous, automated semantics
- ☐ Integrated property meta-language

Examples

```
// Can't say this in ML:

struct A {
  int32 i;
  int32 j;
};

struct B {
  struct A a;
  float f;
};

// no unboxed types
// no concrete int/float types
```

```
// In BitC:

(defstruct A :val
  i:int32
  j:int32)

(defstruct B :val
  a:A
  f:float)
```

Abstraction Failure in C

```
struct list {
  void *elem;
  list *next;
};
...
size_t length(list *l) {
  if (!l)
    return 0;
  return 1+(length(l->next));
}

// not type safe
// types compromised for reuse
// potentially deep recursion
```

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

(define (length l)
  (case tmp l
    (nil 0:word)
    (cons (+ 1 (length l.cdr)))))
length: (fn (list 'a) -> word)

// type safe
// fully abstract
// potentially deep recursion
```

Can Nearly Get There in C++

```
Template <class T>
typedef struct ConsCell<T>
   *List<T>;

template <class T>
struct ConsCell {
  T *elem;
  List<T> next;
};
...
template <class T>
size_t length(List<T> l) {
  if (!l)
    return 0;
  return 1+(length(l->next));
}
```

- Our experience with C++ in the EROS system was *very* discouraging.
- Many languages, no compilers
- Many useful subsets
- *“Giving C++ to ordinary developers is morally equivalent to letting children play with razor blades.”*

— Roger A. Faulkner, 1986

Type Classes, First-Class Procedures

```
// Can't say this in C:
(defclass (Sortable 'a)
  sort: (fn 'a -> 'a)
  sorted: (fn 'a -> bool))

// Can't even say that in C++!
// Intuition: A thing is
// sortable if some means is
// defined to sort it and to
// decide whether it is sorted:

(define (sort-list l) ...)
sort-list:
  (forall ((Ord 'a))
    (fn (List 'a) -> (List 'a)))
(define (sorted-list l) ...)

(forall ((Ord 'a))
  (definstance
    (Sortable (List 'a)
      (sort = sort-list)
      (sorted = sorted-list))))
```

```
// Better:
(forall ((Ord 'a))
  (defclass (Sortable 'a)
    sort: (fn 'a -> 'a)
    sorted: (fn 'a -> bool)))

// Notion: All things that are
// sortable are necessarily comparable
// for magnitude, equality
```

First-Class Procedures

```
// Or this:
(define (add-x x)
  (let ((saved-x x))
    (lambda (y) (+ x y))))
add-x: (forall ((Arith 'a))
           (fn 'a -> (fn 'a ->
  'a)))

((add-x 5) 3:int32)
8:int32
```

```
// Generic map:
(define (map f l)
  (case lst l
    (nil nil)
    (cons
      (cons (f lst.car)
            (map f lst.cdr)))))
map: (fn (fn 'a -> 'b)
         (list 'a) ->
         (list 'b))
```

Refinement of Length

```
// constant space, because
// tail recursion required
(define (aux-len l cnt:word)
   (case tmp l
      (nil cnt)
      (cons (aux-len l.cdr
                  (+ 1 cnt)))))
aux-len:
   (fn (list 'a) word -> word)

(define (list-length l)
   (aux-len l 0))
list-length:
    (fn (list 'a) -> word)
```

```
// local procedures
(define (list-length l)
   (define (aux-len l cnt:word)
      (case tmp l
         (nil cnt)
         (cons (aux-len l.cdr
                     (+ 1 cnt)))))
   (aux-len l 0))
list-length:
    (fn (list 'a) -> word)
```

Type Classes Provide Overloading

```
(defclass (Eq 'a)
  == : (fn 'a 'a -> bool))
(define (!= x y)
   (not (== x y)))
!=: (forall ((Eq 'a))
      (fn 'a 'a -> bool))

(forall ((Eq 'a))
   (defclass (Arith 'a)
     -: (fn 'a 'a -> 'a)
     +: (fn 'a 'a -> 'a)
     *: (fn 'a 'a -> 'a)
     /: (fn 'a 'a -> 'a)))

(forall ((Eq 'a))
   (defclass (Ord 'a)
     <: (fn 'a 'a -> bool)))
...
```

- We introduced these to handle multiple concrete integer types.
- Since then they have become addictive.

Existential Dispatch

```
(defstruct (S 'a 'b 'c)
  m1: (method 'a -> 'b)
  field: 'c)

(define (S.m1 s:S a) ...)
...

(defcapsule (C 'a 'b)
  m1: (method 'a -> 'b))

// Instantiate: methods must
   match
(C (S int32 char char))

// Use:
(define (f o:(C 'a 'b) a)
   (o.m1 a))
// Note that type of S is
// not exposed.
```

- This mechanism is enough to do driver abstraction
- It is closer to Java interface notion than to C++ objects.
- It can be viewed as “objects without inheritance”
- Cleaner re-use is obtained through type classes than inheritance

Practical Result

- Easy to get started
- Nearly as expressive as C, much more powerful
- Not object-oriented, but lots of object things can be done
- Needs a new syntax

Mistakes => Additions

- Noalloc: initially a compile option, later an effect
  - Needs to be part of type; reveals a flaw in currying
- Initialization sanity forced introduction of heap effect typing
  - But once you have this, there are advantages for concurrency
- Too many vector types
  - Dependent typing would solve this, but complicated
  - Decided to give up completeness here
- Now considering region types
  - Functions from locations to locations

More Mistakes => More Additions

- By-ref parameters, array-ref parameters
- In the general areas of temporal extent and vector types, it's clear that we need a more foundational approach.
  - So far, region types seem too complicated for real users
  - Dependent types seem to be a witches brew in a general language.

We believe that we are teetering on the complexity cliff

Some Side Points

- BitC is expressive enough to serve as its own intermediate language within the compiler.
  - Contrast with TIL, whose unboxing transforms cannot be expressed or typed in ML.
  - This was pragmatically useful. Re-running the typer caught many errors.
- Exceptions, constraints, polymorphism are *not* core.
- Some rewrites are type-driven, notably method application.
  - In all of these the type of the rewritten expression is unchanged
  - All of these are motivated by pragmatics
- Polyinstantiator implements “tree shaking” as a side effect
- Amenable to interactive implementation

Why No Separate Compilation

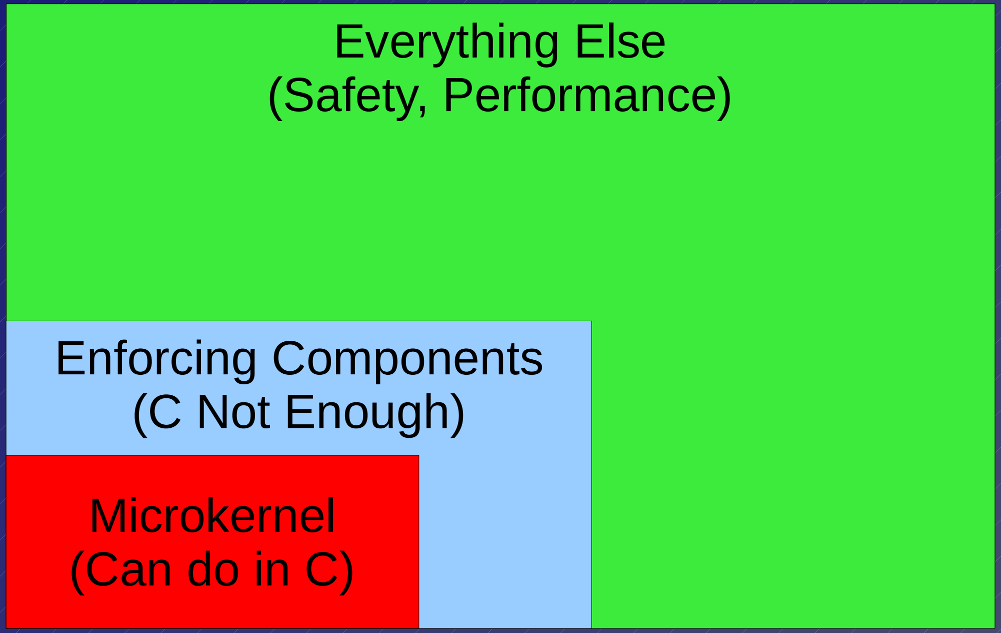
- No single instantiation for both int16 and float
  - `(defunion (list 'a) nil (cons car: 'a cdr: (list 'a)))`
- Language requires “size-based polymorphism”
- We do this by polyinstantiation
  - Similar to template expansion, but well-typed
- Must clone and resolve sizes and offsets at [dynamic] link time.
  - Fortunately, this is *much* simpler than JIT compilation!

BitC in Hindsight

BitC Was Overkill

- Much of the abstraction in BitC can be *annotated* on top of C
- If you are doing software verification anyway, that overhead is negligable.
- C's ambiguities can be avoided in purpose-constructed code, and usually do not matter
  - This is alleged, but I am not convinced.
- You don't prove C programs. You prove idioms.
- So: a fancy new language may not be needed for software verification *of a kernel*
  - *... but for the rest of the system, some of which is critical, C won't do it.*

Systems Complexity Hierarchy



BitC is a Success

- It is proving very friendly to program in
- It can encode the Coyotos microkernel safely
- It uses a fully native-compatible calling convention
- Inference really does provide a practical benefit
  - Ease of prototyping, scripting
- Type safety will improve static checking soundness
- Mostly supports the idioms you know today
  - Takes a while to master the corners
- Almost nothing “new” in the language
  - Can learn from experiences of others

What This Took

- ~40,000 lines of compiler front end
- Three years of serious effort, two people who didn't know modern type systems when we started
  - Swaroop Sridhar did most of the work, and came up with the new type system results
- Several false starts and dead ends

Lots To Do

- Finish the effect type system
- Capture the semantics in Coq
- Libraries:
  - Need to define the core BitC library
  - Need to build interface specifications for various C libs
    - Ideally, want a tool to automate that.
  - Need a decent BigNum implementation

More Information

http://www.bitc-lang.org