BitC (0.11 Transitional) Language Specification[†]

Version 0.11+

Jonathan Shapiro, Ph.D. Swaroop Sridhar Scott Doerrie
The EROS Group, LLC *Systems Research Laboratory*
Dept. of Computer Science
Johns Hopkins University

September 28, 2008

Abstract

BitC is a systems programming language that combines the "low level" nature of C with the semantic rigor of Scheme or ML. BitC was designed by careful selection and exclusion of language features in order to support proving properties (up to and including total correctness) of critical systems programs.

This document provides an English-language description of the BitC semantics. It will in due course be augmented by a formal specification of the BitC semantics. The immediate purpose of this document is to quickly capture an informal but fairly complete description of the language so that participants in ongoing discussions about verifiable systems programming languages have a common frame of reference on which to base their discussions.

Contents

---

1 Overview

The BitC project is part of the successor work to the EROS system [12]. By 2004, it had become clear that a number of important practical "systems" lessons had been learned in the EROS effort. These motivated a re-examination of the architecture. With the decision to craft a revised design and a new implementation came the opportunity to consider methods of achieving greater and more objective confidence in the security of the system. In particular, the question of whether a formally verified *implementation* of the EROS successor might be feasible with modern theorem proving tools. Following some thought, it appeared that the answer to this question might be “yes,” but that there existed no programming language providing an appropriate combination of power, formally founded semantics, and control over low-level representation. BitC was created to fill this gap.

1.1 About the Language

BitC is conceptually derived in various measure from Standard ML, and C. Like Standard ML [10], BitC has a formal semantics, static typing, a type inference mechanism, and type variables. Like C [1], BitC provides full control over data structure representation, which is necessary for high-performance systems programming. BitC also provides support for unboxed mutable locations. The BitC language is a direct expression of the typed lambda calculus with side effects, extended to be able to reflect the semantics of explicit representation.

Versions of BitC up to 0.10 used an s-expression syntax similar to that of Scheme [8]. This allowed us to focus our attention on type inference and semantic issues first. The main goal of version 0.11 is to choose the production surface syntax for the language. For the sake of people who are already familiar with the s-expression syntax, this version of the specification includes *both* syntactic variants.

The transitional syntax is testing a curried style of application syntax so that we can experiment with mixfix operators. In spite of this syntax, BitC application is not curried. Currying encourages the formation of closures that capture non-global state. This requires dynamic storage allocation to instantiate these closures at runtime, and it is difficult for the programmer to determine syntactically when this is happening. Since there are applications of BitC in which dynamic allocation is prohibited, currying is an inappropriate default idiom for this language. We will consider introducing explicit convenience syntax for curried application if this proves to be an impediment to effective use of the language.

In contrast to both Scheme and ML, BitC does *not* provide or require full tail recursion. Procedure calls must be tail recursive exactly if the called procedure and the calling procedure are bound in the same `def`, and if the identity of the called procedure is statically resolvable at compile time. This restriction preserves all of the useful cases of tail recursion that we know about, while still permitting a high-performance translation of BitC code to C code.

As a consequence of these modifications, BitC is suitable for the expression of verifiable, low-level “systems” programs. There exists a well-defined, statically enforceable subset language that is directly translatable to a low-level language such as C. This translation is direct in both the sense that the translation is simple and the result does not violate programmer intuitions about what the program does or the program’s data representation. Indeed, this was a key reason for our decision to move our implementation efforts into BitC.

1.2 Transitional Input Language

We are in the process of migrating from the S-expression syntax to the final BitC surface syntax. S-expression forms are now being incrementally replaced by the corresponding block-style forms, and the compiler no longer accepts some of the older S-expression conventions. I am trying to update this document as the S-expression forms are retired.

1.3 Conventions Used in This Document

In the description of the language syntax below, certain conventions are used to render the presentation more compact.

Input that is to be typed as shown appears in `fixed` font.

Syntactic “placeholders” are shown in italics, and should generally be self-explanatory in context. Variable names, expressions, patterns, and types appear respectively as italic *v*, *e*, *p*, or *T*, with an optional disambiguating subscript. For clarity, the defining occurrence of a name will sometimes appear in the abstract syntax as *nm*.

When a sequence of similar elements is permitted, this is shown using ”,...” or ”;...”, according to whether the elements are separated by commas or semicolons. Such a sequence must have at least one element. For example:

{*e*; ... *e*}

indicates that a block consists of a (non-empty) sequence of expressions separated by “;”. When it is intended that zero elements should be permitted in a sequence, the example will be written:

{[*e*; ... *e*]}

Note that the square braces **[** and **]** have no syntactic significance in the BitC core language after s-expression expansion. When they appear in the specification, they should be read as metasyntax.

1.4 Type Inference

BitC incorporates a polymorphic type inference mechanism. Like SML, BitC imposes the value restriction for polymorphic type generalization. The algorithm for type inference is not yet specified here, and will be added at a future date — we want to be sure that it converges. We currently plan to use a constraint-based type inference system similar to the Hindy-Milner type inference algorithm [10].

The practical consequence of type inference is that explicitly stated types in BitC are rare. Usually, it is necessary to specify types only when the inference engine is unable to resolve them unambiguously, or to specify that two expressions must have the same result type. In this situation, a type may be written by appending a trailing type qualifier to an expression indicating its result type, as in:

```
(a + b) : int32
(+ a b) : int32
```

by similarly qualifying a formal parameter, as in:

```
def fact x =
  if (x < 0) then - (fact (-x))
  else if (x == 0) then 1
  else x * fact (x - 1)

(def (fact x:int32)
  (cond ((< x 0) (- (fact (- x))))
        ((= x 0) 1)
        (otherwise
          (* x (fact (- x 1))))))
```

In general, wherever a type is permitted by the grammar, it is also permissible to write a **type variable**. A type variable is written as an identifier prefixed by a single quote. The scope of a type variable is the scope of its containing definition form. The type inference engine will infer the type associated with the type variable. Within a definition, all appearances of a type variable will be resolved to the same type. This is particularly useful in the specification of recursive types. For historical reasons, `'a`, `'b`, etc. are often pronounced "alpha," "beta," and so forth.

1.5 Documentation Strings

Certain productions in the grammar (Section 15) incorporate an optional documentation string labeled *docstring*. Documentation strings have predefined syntactic positions to facilitate automated extraction by documentation tools. If present, the documentation string must be a syntactically well-formed string, but the string is otherwise ignored for compilation purposes. In certain contexts a documentation string may be followed by an expression syntax, which creates a parse ambiguity. The parser should handle these cases by accepting the expression sequence and then checking to see if it has length greater than 1 and its first element is a string. Note that in such cases the string would be semantically irrelevant in any case. The only point of care here is to note that an expression sequence consisting of a single string is a value, not a documentation string.

I The Core Language

2 Input Processing

The BitC surface syntax is an impure s-expression language. Expressions can be augmented with type qualifiers, and the language provides syntactic conveniences for field reference and array indexing. All of these have canonicalizing rewrites into s-expressions.

Input units of compilation are defined to use the Unicode character set as defined in version 4.1.0 of the Unicode standard [13]. Input units must be encoded using the UTF-8 encoding and Normalization Form C. All keywords and syntactically significant punctuation fall within the 7-bit US-ASCII subset, and the language provides for 7-bit US-ASCII encodable "escapes" that can be used to express the full Unicode character code space in character and string literals.

Tokens are terminated by white space if not otherwise terminated. For purposes of input processing, the characters *space* (U+0020), *tab* (U+0009), *carriage return* (U+000D), and *linefeed* (U+000A) are considered to be white space.

Input lines are terminated by a linefeed character (U+000A), a carriage return (U+000D) or by the two character sequence consisting of a carriage return followed by a line feed. This is primarily significant for comment processing, layout processing, and diagnostic purposes, as the rest of the language treats linefeeds carriage returns as white space without further significance.

2.1 Comments

A comment introduced by the two-character sequence "//" extends up to but not including the trailing newline and/or carriage return of the current line (the end of line markers are significant for purposes of line numbering).

A comment introduced by the two-character sequence "/*" extends up to the following two-character sequence "*/". If an end-of-line sequence is encountered within the comment, the next token encountered after the comment is considered the first token on a new line.

For lexical purposes, comments are considered white space. This implies that the comment syntax cannot be successfully exploited for identifier splicing as in early C preprocessors.

Characters within comments are processed only to determine where the comment ends. Nested comments are not supported.

2.2 Identifiers

Because of BitC's support for user-defined expression syntax (mixfix), the definition of a well-formed identifier is regrettably more complicated than in most languages. We begin by defining three lexical classes:

- Any sequence of code points beginning with an "identifier character" (Unicode 4.1.0 [13] character class XID_Start), followed by any number of optional "identifier continue characters" (Unicode 4.1.0 character class XID_Continue) is a *alphanumeric identifier fragment*.

- Any sequence of the following characters is a valid *operator identifier fragment*:

  `! $ % & * + - / < = > ? ^ | ~`

- An underscore (U+005F, "_"), sharp underscore (U+005F U+0023, "#_"), or the at-sign (U+0040, "@") is a valid *separator*.

A well-formed identifier matches the regular expression:

```
_*(#_)?fragment((_|#_|@)fragment)*(#_|_)?
```

that is: a sequence of alphanumeric or punctuation fragments that are joined by separators with an optional leading and trailing separator, with the whole optionally preceded by an arbitrary number of underscores. Each fragment is either an alphanumeric or a punctuation sequence.

Identifiers are case sensitive. Reserved words are not identifiers. Identifiers starting with *two or more* leading underscores (U+005F, "_") are reserved for use by the runtime system. Non-reservedIdentifiers are valid for use as mixfix identifiers.

The sheer perverse obscurity of the specification for identifiers is motivated by mixfix support, and will hopefully make more sense after a review of that section.

2.3 Interface Names and Identifiers

Interface names consist of a sequence of interface identifiers joined by dots ("."). An interface may start with any interface identifier character, followed by any interface continue character. In addition, an underscore ("_") may appear in any position of an interface identifier, and a hyphen ("-") may be used in any position *other than* the first position.

Interface identifier characters are the Unicode identifier characters (Unicode 4.1.0 [13] character class XID_Start) falling within the 7-bit US ASCII subset (the first 128 Unicode code points). Interface continue characters are similarly the Unicode identifier continue characters (Unicode 4.1.0 [13] character class XID_Continue) falling within the 7-bit US ASCII subset.

The restriction on acceptable character code points in interface identifiers is designed to ensure that interface names can be mapped directly to file names in current file systems. It is expected that the legal namespace for interface identifiers will expand as the capabilities of widely used file system interfaces improve.

Interface names whose leading interface identifier is "bitc" are reserved for use by the BitC runtime system and standard library.

Reserved words are *permitted* as interface identifiers.

While BitC treats interface identifiers as case-sensitive, it is *strongly discouraged* for source code to rely solely on case distinctions to discriminate between interface names. The choice of legal interface identifiers is intentionally chosen to support a direct mapping to file names across a wide range of file systems. At least one pervasively used file system provides only haphazard support for case sensitivity in file names.

2.4 Reserved Words

The following identifiers are syntactic keywords, and may not be rebound:

`->`	`and`	`apply`
`array`	`ArrayRef`	`as`
`begin`	`bitc`	`bitfield`
`block`	`bool`	`boxed`
`ByRef`	`case`	`catch`
`char`	`cond`	`const`
`continue`	`declare`	`def`
`exception`	`deref`	
`disable`	`do`	`double`
`dup`	`enable`	`exception`
`extends`	`external`	`false`
`fill`	`fixint`	`float`
`from`	`if`	`import`

impure	in	instance
int8	int16	int32
int64	interface	is
fn	lambda	let
letrec	MakeVector	member
mutable	not	object
opaque	or	otherwise
pair	proclaim	provide
pure	quad	repr
return	sizeof	
string	struct	suspend
switch	tag	the
throw	trait	try
true	uint8	uint16
uint32	uint64	unboxed
union	use	vector
version	where	until
word		

The following identifiers are reserved for use as future keywords:

assert	break	check
constrain	deep	object
deftype	do*	inner-ref
int	let*	list
location	module	namespace
ptr	read-only	require
ref	sensory	super
tycon	tyfn	typedef
using	value-at	

In addition to the reserved words identified above, all definitions provided in the standard prelude are implicitly imported into the initial top-level environment of every compilation unit.

Note that BitC does *not* permit redefinition of bound variables in the same scope. This guarantees that top-level forms receive the default bindings of these identifiers in their environment.

For the moment, all identifiers beginning with "`def`" are reserved words. This restriction is a temporary expedient that is not expected to last in the long term.

Finally, the identifiers defined as part of the BitC standard runtime environment (described below) are bound in the top-level environment.

2.5 Literals

The handling of literal input and output is implemented by the standard prelude functions `read` and `show`. Source tokenization, requires that foundational literals have a defined canonical form.

2.5.1 Integer Literals

An integer literal takes one of the following forms:

```
decimal digits  // decimal integer literal
0xhex digits  // hexadecimal integer literal
0ooctal digits  // octal integer literal
0bbinary digits  // binary integer literal
0octal digits  // octal integer literal
```

A decimal integer literal may not begin with a leading zero. *Digits* are selected from the characters

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
```

with the customary hexadecimal valuations. The letters may appear in either lowercase or uppercase. It is an error for a digit to be present whose value as a digit is greater than or equal to the specified base.

Integer literals of a particular fixed-precision type may be written by using a type qualifier. The expression:

```
564 : uint32
```

specifies an unsigned 32 bit quantity whose value is 564. It is a compile-time error to qualify an integer literal with a type that is incapable of representing the literal's value. In the absence of explicit qualification, the type assigned to an integer literal will be some subset of:

```
int8  int16  int32  int64
uint8 uint16 uint32 uint64
word
```

Any concrete type that cannot represent the literal value will be omitted from the set of types assigned.[1]

2.5.2 Floating Point Literals

The general form of a floating point literal is:

```
digits.digits[e[-]exponent]
```

where all digits are decimal. Floating point values may not have any radix other than 10. The *exponent* may include an optional initial minus sign. Note that the decimal point is not optional, and must have at least one digit on both sides. Thus, `0.0` is a valid floating point literal, but

[1] There is an issue here: doesn't the initial set need to be the set of all integer field sizes so that initialization can work? Shap thinks that the answer is probably yes, but that it isn't a problem in practice because the arithmetic operators are only defined over homogeneous argument types. Swaroop points out that expanding the set isn't what creates the problem for type inference.

`0.` and `.0` are not. Also `1e11` is not a valid floating point literal; `1.0e11` should be used instead.

As with integer literals, floating point literals of an explicitly stated representation type may be written using a type qualifier. The expression:

```
0.0 : float
```

specifies a 32-bit (single precision) IEEE floating point quantity whose value is zero. As with integer literals, it is a compile-time error to specify a value cannot be represented within the representable range of the qualifying type.[2] In the absence of explicit qualification, the type of a floating point literal is some subset of:

```
float double quad
```

Any concrete type whose representable range cannot express the literal value will be omitted from the assigned set.

Conversion of a floating point literal to internal representation follows the customary IEEE floating point rounding rules when the specified literal cannot be exactly represented.[3]

2.5.3 Character Literals

BitC uses the Unicode character set as defined in version 4.1.0 of the Unicode standard [13]. Characters are 32 bits wide. Character literals can be expressed in two ways.

A character literal may be written as

```
'printable-character'
```

Where *`printable-character`* is any character specified in the Unicode 4.1.0 standard *except* those with general categories "Cc" (control codes) "Cf" (format controls), "Cs" (surrogates), "Cn" (unassigned), or "Z" (separators). That is, any printable character, excluding spaces. Notwithstanding the listed Unicode categories, the characters "\" (U+005C) and single quote (U+0027) are excluded, and the character "space" (U+0020) is considered printable.

An arbitrary character may also be specified numerically in one of the following forms:

```
'\U+hex digits'  // unicode code point
'\decimal digits'  // unicode code point expressed in
'\0xhex digits'  // unicode code point expressed in h
'\0ooctal digits'  // unicode code point expressed in
'\0bbinary digits'  // unicode code point expressed i
'\0octal digits'  // unicode code point expressed in
```

[2] It is *not* an error if conversion of the literal value causes loss of precision in the low-order bits of the mantissa.

[3] A more precise statement is needed for floating point literal conversion, but I don't know enough about floating point conventions to know what that statement should be.

The value supplied must be a valid unicode code point, which is a value in the range 0..10FFFF hexadecimal.

Certain commonly used non-printing characters have convenience representations as character literals:

```
'\space', '\s'
'\linefeed', '\n'
'\return', '\r'
'\tab', '\t'
'\backspace', '\b'
'\formfeed', '\f'
'\backslash', '\\'
'\squote', '\''
'\dquote', '\"'
```

2.5.4 String Literals

BitC strings are written within double quotes, and may contain the previously listed "printable characters" *excluding* backslash "\" (U+005C), or the double quote character.

Within a string, the backslash character "\" (U+005C) is interpreted as beginning an encoding of an escaped character. The character following the "\" (U+005C) is either a single-character embedding from the list given above, or a curly brace character "{" (U+007B) identifying the start of a numeric or named embedding that is terminated by "}" (U+007D).

2.6 Compilation Units

There are two types of compilation units in BitC: interfaces and source compilation units. An interface compilation unit defines or declares types (and consequently the code of type constructors), defines type classes, defines constants, and declares values. A source compilation unit can define types, type classes, constants, and values.

Every valid BitC compilation unit must begin (ignoring comments) with a `bitc version` form. The syntax of the `bitc version` form is:

```
bitc version n.m
```

where *`n.m`* is the version of BitC version (major *dot* minor) to which this program conforms. For the version of BitC described in this document, the proper version is `0.11+`. It is a compile-time error if the language version accepted by the current compiler is not backwards compatible with the version specified by the `bitc version` form.

In an interface compilation unit, the `bitc version` form is followed by exactly one `interface` form (Section 9). In a source compilation unit, the optional `bitc version` form is followed by one or more `module` forms (Section 9).

A source compilation unit may alternatively consist of the `bitc version` form followed by an arbitrary sequence of imports, definitions, declarations, and use forms that are *not* `interface` forms. In this case the forms following the `bitc version` are deemed to be implicitly enclosed by a `module` form, and the compilation unit defines exactly one source module.

2.6.1 Definitions and Declarations

The top level forms that introduce programmatic definitions and declarations are:

`def` Introduces value declarations and definitions, including functions.

`struct` Introduces structure declarations and definitions.

`union` Introduces union declarations and definitions.

`repr` Introduces repr definitions and declarations. Reprs are a special kind of union with fine-grain layout and tag control.

`trait` Introduces traits definitions, which restrict how type variables can be instantiated and provide overloading.

`instance` Introduces trait instance definitions, which identify the types for which a trait is defined.

The `def`, `union`, `struct`, and `repr`, forms support simple recursion. That is, the identifier(s) being defined may be used in their definition. However, identifier(s) being defined are deemed incomplete until the end of the enclosing defining form. Restrictions on the use of incomplete identifiers are described in the sections on types and value binding.

The `proclaim` form is used to provide opaque value declarations. The identifier declared by a `proclaim` form is considered incomplete. If a completing definition is later provided within the same compilation unit, the identifier is considered complete in the balance of the defining compilation unit after the the close of its defining form. An incomplete declaration may be used within a procedure, but may not be used as part of a top-level initializer (see `def`, Section 5.2).

All defining forms are expressions that return a value of type **unit**.

2.7 Layout

The goal of the layout system is to provide programmer convenience by automatically inserting curly braces and semicolons wherever they have not been inserted by the programmer. Left curly braces are conditionally inserted after certain preceding tokens. Semicolons are conditionally inserted based on the indentation level of each line. Right curly braces are conditionally inserted before certain tokens and also based on the indentation level of the curren tline.

2.7.1 Concept

The "trick" to the layout scheme is in two parts:

1. There are several sequencing constructs in the language whose general form is a semicolon-separated sequence of things surrounded by curly braces. This lets us use the same layout rules for multiple purposes.
2. At each point where a left curly brace might be automatically inserted, a preceding token (one of `let`, `do`, or "`=`") signals unambiguously that a left curly brace must follow:

```
def f x y = { ...
def x = { 5 }
struct S 'a 'b = { a : int; ...
let { x = { 5 } } in { ...
while (expr) do { ...
do { ... } while (expr)
```

We rely on the fact that (a) knowing the preceding and current token is enough to know whether to insert a curly brace (b) because blocks are values, all binding expressions can safely be wrapped in blocks, (c) because such wrapping is always safe, it can be done in a way that is invisible to the programmer In fact, the parser insists that this be true. While it is nearly universal practice in Bitc to write code without semicolons in many places, most notably:

```
let x = 5 in body
```

But what the compiler actually *sees* is:

```
let { x = { 5 } } in body
```

We also rely on the fact that correct programs do not under-indent lines capriciously.

2.7.2 Specification

- The term *offset*, as used here, is defined as the number of preceding UCS4 code points that occur to the left of the token on the same line. The first code point on a line is deemed to have offset zero. Offsets are determined without regard to comments. That is, in:

  ```
  a b c /*
   */ x
  ```

  the token x appears at offset 5 and is the first token on its line.

- The term *layout item sequence* is a sequence of layout items that are separated by semicolons and surrounded (bracketed) by curly braces. An opening '{' signals the beginning of a layout item sequence.

- A left curly brace will be automatically inserted after the keywords **let**, **do**, and **=** (binding) if none appears explicitly in the input. Note that a left curly brace is required by the grammar at each of these positions.

- The lexer maintains a record of every left curly brace (whether or not inserted), in a stack of layout contexts. Each entry records the preceding keyword (**let**, **do**, or **=**), whether the left curly brace in question was automatically inserted or not, and the offset of that layout context. The most recent entry on the layout context stack is "popped" whenever an implicit or explicit '}' is encountered.

- On encountering the **in** token, the lexer will insert implicit close braces ('}'), popping layout contexts as it goes, until one of the following conditions holds:

  1. The last layout context popped was associated with the **let** keyword. That is: curly braces inserted before **in** will only balance up to the nearest preceding **let**.
  2. The top entry on the context stack was explicit. That is: implicit closing curly braces will only balance implicit open curly braces.

- On encountering end-of-file, the lexer will insert implicit close braces ('}'), popping layout contexts as it goes, until the top entry on the context stack is explicit.

- Every (implicit or explicit) '{' begins a layout item sequence. After processing the '{', the lexer examines the next token. If it is end-of-file or **in**, processing proceeds as described above. Otherwise:

  – If the offset of the token is *greater than* the current sequence offset, it becomes the current sequence offset and is recorded in the top (most recent) layout context stack entry. Regardless of offset, no implicit semicolon will be inserted before this token.

  – If the offset of the token is *less than or equal to* the current sequence offset, and the most recent open brace was implicit, an implicit close brace is immediately inserted.

- Provided it does not follow an opening brace, the offset of the first token on every line is used to determine whether a closing curly brace '}' or a semicolon should be conditionally inserted as follows:

  – If the offset of the token is *greater than* the current sequence offset, the token is "," or "(", or the preceding token is "," or "(", nothing is inserted. This special-case suppression enables the use of longer string literals as arguments in I/O.

  – If the offset of the token is *less than* the current sequence offset, and the most recent open brace was implicit, an implicit close brace is immediately inserted.

  – If the offset of the token is *equal to* the current sequence offset, the token is not a semicolon (';'), and the most recently returned token was not a semicolon, a semicolon is automatically inserted. *Qualifications to this rule are needed concerning the current and immediately preceding token.*

- An implicit '{' must be matched by an implicit '}'. Similarly, an explicit '{' must be matched by an explicit '}'. If this requirement is violated, an error is signalled.

3 Types

3.1 Overview

To mitigate mitigate the effects of forward reference in the specification, we begin with a high-level overview of the BitC type system. This overview is non-normative.

BitC provides explicit control over data structure representation while preserving a memory-safe and type-safe language design. Where types are specified explicitly by the developer, both the layout and size of the underlying data structures for a given compilation target platform are fully specified. Different target platforms may yield different layouts and sizes due to (e.g.) differences in underlying pointer types. In consequence, the use of constructs such as `sizeof` and `bitsizeof`, or of the machine-dependent type `word`, can render a program target-dependent.

BitC provides a powerful type inference mechanism. In the absence of concrete declaration or specification through type parameters, the type of a binding is *inferred* by type inference based on its initializing expression and their context(s) of use.

3.1.1 Terms vs. Locations

Term Bindings If the initializing expression is pure, and the bound value is never mutated, that it is a *term binding*, and is given polymorphic type. For example, the type reported for the following definition:

```
def i = 1 + 3;
i: 'a where IntLit('a)
```

indicates that `i` is a term binding. It inhabits any type that satisfies the `IntLit('a)` constraint, which is to say: any type that admits both `1` and `3` and as valid literals.[4] The intuition is that a term binding's semantics is defined by substitution at the point of use. The final type of the binding is determined by inference following substitution. The implementation is free to actually perform such substitution or to instantiate a concrete binding for each concrete type of `i` that is actually seen.

With two exceptions, the BitC inference mechanism produces typings that are complete; the derived type will be the *least constraining* type that is permissible given the initializing expression and the context(s) of use. The exceptions are:

- The current inference algorithm does not infer array, ArrayRef, or ByRef types.
- BitC does not (yet) provide polymorphic recursion.

It is expected that both of these limitations will be lifted in upcoming enhancements to the language.

Location bindings If the binding's initializer has side effects, or if the bound variable is mutated (in whole or in part), then the binding is a *location binding*. Location bindings are given monomorphic type and the value occupies some concrete location. Because their values are subject to change, location bindings are not equivalent under term substitution.

The concrete type of a monomorphic binding must be fully known by the close of its defining form. When the binding occurs at top-level, this means that use-occurrences cannot be considered by the inference engine to determine whether the binding is a location binding or a term binding. In the absence of specification, a term binding is inferred:

```
def i = 1 + 3;
i: 'a where IntLit('a)
def mi : mutable('a) = 1 + 3 : int32;
mi: mutable(int32)
def bad_mi : mutable('a) = 1 + 3
error: 'a monomorphic, not fully determined
```

Because it is able to consider the contexts of use, the BitC inference algorithm will infer a location for local location bindings.

3.1.2 Boxed and Unboxed Types

Most languages having advanced type systems have a heap-based value semantics. Conceptually, all values live in the heap and are manipulated through references. BitC has two categories of types — boxed and unboxed — each with associated values.

An unboxed type is one whose values are "contained" within some composite value (which may be a stack frame). The lifetime of an instance of unboxed type is determined by the lifetime of its container, and it is the responsibility of the container to reserve storage for its contained value types. Such types are said to be "unboxed." All of BitC's primary types are unboxed types.

A boxed type is one whose values live in the heap. Every instance of a boxed type has at least one reference that denotes it. The reference is an unboxed value; the value it denotes is boxed.

If *T* is an unboxed type, then `boxed(T)` is the type of a reference denoting a heap-allocated instance of *T*. Unboxed values can be copied to the heap using `dup()`.

Similarly, if *T* is a boxed type whose target is of statically known size, and whose type unifies with `(boxed 'a)`, then `unboxed(T)` is the corresponding value type. *Need to explain why the qualification is needed. Issue is unboxing of boxed unions, but I can't remember the details at the moment.* Unboxed copies of boxed values can be obtained using `deref()`.

At the moment, BitC requires that all structure and union

[4] This particular typing is expected to change when we introduce the `Nat` kind.

declarations explicitly state whether the type defined is boxed or unboxed. We will eventually relax this requirement, making `boxed` the default for union types and `unboxed` the default for struct types. Exception types are boxed by definition; specification that an exception type is boxed is permitted, but not required.

BitC does not provide automatic assignment conversion between value types and reference types.

3.1.3 Type Variables

BitC permits types to be parameterized by means of type variables. Type variables provide a means of specifying that the type of some element will be provided at a later point of specialization. Permitted instantiations may be constrained by *type classes*. For example, the type:

```
def isLessThan(a,b) = a < b;
fn('a, 'a) -> bool
  where Ord('a)
```

indicates that `isLessThan()` is defined over any specialization of `'a` for which a definition of ordering methods has been provided by an instantiation of the `Ord('a)` trait.

3.2 Primary Types

The primary types of BitC are:

unit The unit type, having as its singleton member the unit value, both of which are written as `()`.[5]

bool A boolean value, either `false` or `true`. The representation of this type is a single byte, aligned at a byte boundary.

char A unicode code point. The representation of this type is a 32-bit unsigned integer, aligned at a 32-bit boundary.

word The type `word` is the smallest unsigned integral type whose range of values is sufficient to represent the bit representation of a pointer on the underlying machine. This type is architecture dependent, and is *not* directly assignment compatible with unsigned integral types of the same size. Values of type `word` are aligned at a boundary that is a multiple of their size.

bitfield *This is not a primary type, and it's description needs to move into the section dealing with aggregate types.*

The `bitfield` form describes a fixed-precision integer field:

```
basetype(size)
```

Where *`basetype`* is one of the primary fixed-precision integral types and *`size`* is a literal not exceeding the size in bits of the base type.

The form

```
int32(4)
```

describes a two's complement four bit field placed within a 32-bit alignment frame.

Bitfields may only be used as types of structure, union, or tag fields. The type of a bitfield is deemed to be assignment and binding compatible with its *basetype*. A bitfield over a signed base type is sign-extended as needed when copying to its base type. A bitfield over an unsigned base type is zero extended.

float, double, quad The types `float`, `double`, and `quad` describe, respectively, IEEE floating point values as described in [2][3]. The `quad` type is an extended precision floating point type with a 15 bit exponent and a 112 bit mantissa.

3.3 Type Variables

A type variable is an alphanumeric identifier fragment preceded by a single quote, as in `'a`. A type variable, may appear in any position where a type can appear. This indicates a type parameter whose actual (concrete) type is to be specified later. A type containing type variables within its description must be "specialized" by supplying the type corresponding to each parameter. In the case of parameterized structure and union types, each type variable is specialized by supplying it at the point of use: `list(char)` is an instantiation of `list('a)`. In the case of function types and let/letrec bindings, type variables are specialized automatically during compilation by inferring the required type of each type variable at each point of application.

The scope of a type variable is generally its outermost defining form. In some cases, a `letrec`, `let`, or local definition can introduce a new type variable scope. Because a type variable's scope covers an entire definition, type variables can be used used to annotate that two expressions must have the same type. The expression:

```
myfun(x, y:'a):'a
```

says that the type of `x` is unspecified by the program author (and should therefore be inferred), the return type is

[5] Note that **unit** is not a keyword.

also unspecified (and should be inferred), but the return type and the last argument type are the same. This type of annotation is sometimes useful to assist the inference engine.

3.4 Simple Constructed Types

Constructed types compose existing types into new types. Type equivalence for the simple constructed types is determined by structural equivalence.

3.4.1 Reference Types

If *T* is a value type, then

```
boxed(T)
```

is the type of a reference denoting a heap-allocated instance of *T*.

Storage Layout The representation of a `ref` instance is architecture dependent. It is customarily determined by the size of the machine's integer registers, and aligned at any address that is congruent mod 0 to the integer register size.

3.4.2 Function Types

If t_{arg} and t_{result} are types (including type variables), then:

```
fn (targ_1, ...targ_n) -> tresult)
```

is the type of a function taking *n* arguments of types t_{arg1} through t_{argn}, respectively, and returning a value of type t_{result}. The type of a function taking zero arguments is written as:

```
fn () -> tresult)
```

Storage Layout Function types are considered reference types that denote an object of statically undefined size. The size and alignment of a value of function type is the size and alignment of the underlying architecture's pointer size.

3.5 Sequence Types

BitC provides fixed-length (`array`) and variable-length (`vector`) types.

3.5.1 Arrays

An array is a value type whose value is a fixed product type $T^{i>0}$, all of whose elements are of common type. The type:

```
(array T i)
```

describes the type of fixed-length arrays of element type *T* and length *i*, where *i* is an integer literal of type `word` that is greater than zero.

Storage Layout The value representation of a *k*-element array is laid out in memory as the concatenation of *k* contiguous element cells whose size and alignment are determined by their respective element types. The elements of the array appear at increasing addresses in order from left to right.

3.5.2 Vectors

A vector is a dynamically sized array whose elements are of type *T*. Vectors are reference types. Because they are dynamically sized, there is no corresponding value type. The type:

```
(vector T)
```

describes vectors of element type *T*.

3.5.3 Array References

An array reference is a sequence type whose elements are of type *T* and whose length is dynamic. A parameter or let binding of type `(ArrayRef T)` will accept as its corresponding actual parameter or initializer a value of either type `(ArrayRef T)` or type `(array T len)` for *any* length. There is no value constructor for array reference types.

Array references are not permitted to escape. Pending definition of a standardized escape analysis for BitC, the `ArrayRef` type is not permitted as the return value of a procedure, a non-value expression, a structure field, or a closed-over value. The type:

```
(ArrayRef T)
```

describes array references of element type *T*.

3.6 Named Constructed Types

The named constructed types are types whose compatibility rules are determined by name equivalence. Two values of named constructed types are equivalent if (a) they are instances of the same statically appearing type definition, and (b) their corresponding elements are equivalent.

Unless otherwise qualified, a named constructed type declaration declares a reference type.

3.6.1 Structures

A structure definition defines a named type whose instances are an ordered sequence of named cells. The syntax of a structure declaration is:

```
(struct nm field ...)
(struct (nm tv1 ... tvn) field ...)
```

where each *field* is one of:

```
nm:type
(the type nm)
(fill bitfield-type)
(reserved bitfield-type value)
```

All names *nm* are disjoint identifiers giving the names of the structure fields, and the respective *type* forms are the types of the respective fields. Given a variable `v` that is an instance of a structure type having a field named `f`, the expression `v.f` unifies with the field `f` within that structure.

A `fill` element may be used to support precise specification of alignment. The alignment and storage layout of a fill field follows the alignment and storage layout of its base type, however a fill field has no name or defined value, and cannot be programatically referenced.

A `reserved` element may be used to specify a reserved bit position in a low-level data structure that is required to hold a known value. It is otherwise identical to a `field` element.

An identifier that is bound to a structure type may be used as a procedure to instantiate new values of that structure type. The arguments to this procedure are the initial values of the respective structure fields.

An identifier that is bound to a non-parameterized structure type may be used as a type name. An identifier that is bound to a parameterized structure type may be used in a type constructor application within a type specification. Its arguments are the types over which the newly instantiated structure type should be instantiated. For example, the declarations:

```
(struct ipair a:int32 b:int32)

(struct (tree-of 'a):boxed
  left : (optional (tree-of 'a))
  right : (optional (tree-of 'a))
  height : int8
  value : 'a)
```

define (respectively) the type name `ipair` and the single argument type constructor `tree-of`.

Storage Layout A structure having *k* fields is laid out in memory at increasing addresses from left to right as *k* contiguous cells whose size and alignment are determined by their respective element types. These cells are then packed according to the previously described alignment and layout packing rules. *Did we describe them?*

3.6.2 Unions

The `union` form defines enumerations, discriminated unions, and mixes of these. The type being defined is in-scope within the definition of the type, but is incompletely defined. The syntax of a union declaration is one of:

```
(union nm C1 ... Cn)
(union (nm tv1 ... tvn)
  C1 ... Cn)
```

where each tv_i is a type variable and C_i is a **constructor form**. A constructor form consists of either a single identifier or a parenthesized identifier followed by a sequence of field or fill declarations (see `struct`). All field names appearing in a `union`, including constructor names, must be disjoint.

An identifier bound to a union constructor having no fields denotes the value of the unique corresponding union instance for that union type.

An identifier bound to a union constructor having associated fields is a procedure that may be used to instantiate new instances of that union type. The arguments to this procedure are the initial values of the union fields associated with that union variant.

An identifier that is bound to a non-parameterized union type is a valid type name. An identifier that is bound to a paramterized union type may be used in a type constructor application within a type specification. Its arguments are the types over which the newly instantiated structure type should be instantiated. For example, the declarations:

```
(union contrived
  (asChar c:char) (asInt i:int32))

(union (optional 'a) :unboxed
  none
  (some value:'a))
```

define (respectively) a reference type holding either a `char` or an `int32`, and a value type of optional elements.

The declaration:

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

Defines the reference type of homogeneous lists.

Storage Layout Each variant of a union declaration effectively defines a *k+1* element structure, where the first element contains the tag and the remaining *k* elements are the fields of the constructor leg. In the usual case, the representation of the union leg is arranged as though it had actually been this structure, without regard to the layout of other legs.

In the unusual case of a union whose tag representation can be elided (see below), each individual union leg will be arranged as though it had been the corresponding structure declaration.

In the case of a union having no tag, the union representation will match the size and alignment of reference cells. The storage occupied by a union of value type is the maximum of the storage required for each individual case of the discriminated union (including the type tag, if present).

Type Tag Size and Alignment In the absence of declaration, the union type tag will be given an implementation-defined size and alignment selected to maximize performance efficiency. Explicit control over the size and alignment can be achieved using a `tag-type` declaration. The declaration:

```
(union (list 'a):boxed
  (declare (tag-type uint8))
  nil
  (cons car:'a cdr:(list 'a)))
```

indicates that the tag should be implemented using an unsigned byte. The declared tag type must be an unsigned integral or bitfield type having a sufficient number of distinct values to assign a unique value to each constructor.

Tag Representation The prelude type `nullable('a)` must be implemented in a single pointer-sized machine word, with the `Null` case being tagged by a word value of zero and the `ptr` case being tagged by a non-null word value. This yields a concrete representation that is compatible with the representation of nullable pointers in other languages.

The following representation requirements are required unless they cannot be legally implemented on the underlying machine, as in JVM or CLR.

In the absence of an explicit declaration of the type tag representation, a union type having exactly one union leg whose first element's type `'a` unifies with `RefType('a)` or `nullable('a)`, and all of whose other legs have no fields shall be represented in such a way that the tag word reuses the storage of the ref/nullable field. The ref/nullable leg shall be denoted by a tag field whose least significant bit is zero. The n'th enumeration leg's tag value (in order of appearance) shall be encoded as `n*2+1`. This representation is sometimes known as the Cardelli optimization, because it permits a two-word implementation of CONS cells, as in Scheme or LISP.

If a union type tag is explicitly declared to be of a field type whose size in bits `b` is such that the machine's natural heap alignment restriction for objects is $\texttt{align}>=2^{\texttt{b}}$, the total number of distinct legs of the union does not exceed $2^{\texttt{b}}$, and there is exactly one union leg whose first element's `'a` type unifies with `RefType('a)` or `nullable('a)`, then the tag field shall overlay the least significant bits of the ref/nullable field, the tag value zero shall denote the ref/nullable leg, and all other tag values shall be non-zero.

3.6.3 Reprs

There are examples of low-level hardware data structures for which the unions and structures that can be specified using `struct` or `union` are insufficiently expressive. One example is the Pentium GDT data structure, which has nested union discriminators, but simultaneously has an overall bit-level layout requirement. Another example is data structures where the representation of the tag must appear at a specific location that is not adjacent to the fields guarded by the tag. The `repr` form is included to permit the expression of these data structures.

The following scheme for `repr` is based on the bit-data representation proposed by Iavor Diatchki, *et. al.* [11].

Similar to unions and structures, a `repr` delaration takes the following form:

```
(repr name
  (Ctr1 f11:type f21:type ... fn1:type
    (where (== fp1 v11) (== fq1 v21) ...
           (== fm1 vm1))

  (Ctr2 f12:type f22:type ... fn2:type
    (where (== fp2 v12) (== fq2 v22) ...
           (== fm2 vm2))
```

```
... )
```

The following restrictions apply. For all constructors `Ctrx`, `Ctry`, `Ctrz`, `...`:

- All fields `fpx`, fqx...fmx appearing in the `when` clause of a constructor form `Ctrx` must be described within the body of `Ctrx`. That is, {`fpx`, `fqx`, `... fmx`} ⊆ {`f1x`, `... fnx`}.
- Identically named fields within two different constructor forms must be located at the same bit level offset from the beginning of both the constructor forms. That is, `fpx` = `fpy` implies bit-offset(`fpx`) = bit-offset(`fpy`).
- Identically named fields within two different constructor must have the same type. That is, `fpx` = `fpy` implies type-of(`fpx`) = type-of(`fpy`).
- The fields within the `when` clauses of all constructor forms must uniquely distinguish all constructible values of the union. The compiler will not introduce any more tag bits for a `repr` value.
- Currently, the `repr` form will not accept type arguments over which it can be instantiated. That is, the following definition is not legal.

  ```
  (repr (name 'a 'b ... ) ... )
  ```

- Currently, the discriminating fields `fp1`, `fp2`, etc must have a integer/bitfield type, and the discriminator values `v11`, `v12`, etc must be an integer literal.

We can envision a larger language construct `UNION`, which accepts both type arguments and `when` clauses. The `union` and `repr` are just specializations of this `UNION` construct. However, currently the language only supports `union` (which does not accept the `when` clause) and `repr` (which does not accept type arguments).

3.6.4 Objects

This description is provisional. The feature is a work in progress.

In BitC, an object provides a form of existential dispatch. Objects are declared similarly to structures. The syntax of an object declaration is:

```
(object nm field ...)
(object (nm tv1 ... tvn) field ...)
```

where each field is required to be of method type.

An identifier that is bound to a object type may be used as a procedure to instantiate new values of that object type. The single argument to this procedure must be an instance of some compatible structure type *S*. A structure type *S* is deemed compatible if *S* is of reference type and for every method *m* in the object type, there must be a corresponding method of the same name in *S* whose type is at least as general as the method type declared in the object type.

An identifier that is bound to a non-parameterized object type may be used as a type name. An identifier that is bound to a parameterized object type may be used in a type constructor application within a type specification. Its arguments are the types over which the newly instantiated object type should be instantiated. For example, the declarations:

```
(object O i:int32)
(object (Oparam 'a) x:'a)
```

define (respectively) the type name `O` and the single argument type constructor `Oparam`.

An object occupies two words of storage one of which is a reference to a method table and the other is a reference to an object of corresponding structure type. Construction of an object from a structure instance entails capturing a reference to that instance and a reference to a method table mapping the method declarations of the object onto the corresponding method definitions of the referenced structure type. Invocation of an object method is realized as invocation of the corresponding method of the referenced structure instance.

Storage Layout An object occupies two machine words, the first of which is a reference to a method table and the second of which is a reference to an object of corresponding structure type. The alignment of this structure is dictated by the pointer alignment requirements of the underlying hardware implementation.

3.6.5 Value vs. Reference Types

In the absence of other specification, the `struct`, `repr`, `union`, and `object`, forms declare reference types. The developer may optionally qualify the declaration to make this intention explicit:

```
(struct nm:boxed field ...)
(struct (nm tv1 ... tvn):boxed
  field ...)
(union nm:boxed C1 ... Cn)
(union (nm tv1 ... tvn):boxed
```

```
  c1 ... cn)
(repr nm:boxed (body))
(repr (nm tv1 ... tvn):boxed (body))
```

The qualifier ":boxed" indicates that the type declared (and consequently the type returned by value constructors) is a heap-allocated type, sometimes known as a "reference type." The qualifier ":unboxed" indicates that the type declared is a type whose storage is allocated within its containing data structure (which may be the stack). These are sometimes refered to as "value types." The qualifier ":opaque" indicates that the type declared is a value type whose internal structure is not accessable outside of the defining interface and the providers of that interface. An importer of an opaque type may declare fields and variables of that type and can *copy* instances of that type, but can neither apply the type constructors nor make reference to the contents of instances.

Note that if the type declared is a value type, it cannot be instantiated within the body of the declaration because its size is not statically known. That is, it is legal to have a field that is a *reference* to a value of the type currently being defined, but not a value of that type.

3.6.6 Forward Declarations

The declarations

```
qual struct nm [external];
qual union nm [external];
qual repr nm [external];
qual struct nm(tv1, ... tvn)
          [external]
qual union nm(tv1, ... tvn)
          [external]
qual repr nm(tv1 ... tvn)
          [external]
```

state (respectively) that `nm` is a structure (respectively union) reference type of the stated arity whose internal structure is not disclosed. The qualifier *qual* declares the type to be one of "boxed", "unboxed", or "opaque". In the absence of qualification, the default is "boxed"[6] If present, the *external* portion consists of the keyword `external` followed by an optional identifier (see discussion of external identifiers in `proclaim`).

For example, the following declaration is include in the library `bitc.int` interface to declare the bignum type:

```
unboxed struct int external bitc_int
```

[6] The qualifier is presently mandatory on struct, union, and repr declarations during syntax transition. This requirement will be relaxed in a future revision of the specification.

The structure of these types may optionally be disclosed later in the same compilation unit by a type definition for `nm`. If the declaring form appears within an interface, the corresponding type definition may appear in a providing unit of compilation, in which case the type is opaque to importers of the interface.

Note that a forward declaration of a value type is sufficient to declare *references* to that type, but not *instances* of that type. A complete definition of the value type is required to be in scope in order to declare fields and variables of value type.

3.6.7 Method Types

If `S` is a structure or object type, and t_{arg} and t_{result} are types (including type variables), then a field *m* of `S` may be declared as:

```
(method targ_1...targ_n -> tresult)
```

A method type may only be specified as a field type of a field within a structure or object type. Methods occupy no storage in their associated structure or object.

Structure Methods In a structure type, methods may be viewed as a procedure proclamation that is coupled to a convenience syntax supported by application. The declarations:

```
(struct S
  m: (method int32 -> bool))
(struct T : val
  m: (method int32 -> bool))
```

implicitly proclaim (respectively) the procedures:

```
(proclaim S.m: fn (S, int32) -> bool)
(proclaim T.m:
   fn (ByRef T, int32) -> bool)
```

where the parameter corresponding to the argument of structure type is `ByRef` exactly if the corresponding structure type is a value type. Implementations of these procedures must be provided elsewhere by the developer.

Given these declarations, and an expression *e* returning a value of type `S`, the application:

```
(e.m 3)
```

is a syntactic convenience for:

```
(S.m e 3)
```

Object Methods In an object type, methods may be viewed as a procedure proclamation that is likewise coupled to a convenience syntax supported by application. As with structure methods, they implicitly proclaim corresponding procedures. In *contrast* to structure methods, the implementation of these procedures is provided by the compiler.

3.6.8 Named Type Conveniences

The following types are defined in the BitC standard prelude.

```
(struct (pair 'a 'b) :unboxed
   fst:'a snd:'b)

(union (list 'a) :boxed
   nil
   (cons 'a (list 'a)))
```

Note that `pair` is a keyword that is specially recognized in binding patterns.

The `pair` type is supported by a right-associative infix convenience syntax:

```
(a, b) => (pair a b)
(a, b, c) => (pair a (pair b c))
```

This convenience syntax may be used in types, binding patterns, and value construction.

3.7 Const

The `const` keyword is a type *metaconstructor*. If *T* is a type, then the type `const(`*T*`)` is a type that is copy-compatible (3.10) with *T*, but has had all mutability stripped (recursively) at all shallow constituent fields. This enables a local, shallowly constant copy to be made of a structure containing mutable constituents.

The `const` construct is considered a meta-constructor because of its "sticky" behavior under unification. The type `const(`*'a*`)` does not unify with any type (shallowly) containing a mutable constituent field.

3.8 Mutable

Unless modified by the `mutable` keyword, the preceding types yield immutable instantiations. If *T* is a type, then the type `mutable(`*T*`)` is the type of mutable instances of *T*. If the type *T* is a reference type (including `:boxed` structure types), then the type `mutable(`*T*`)` describes a mutable reference to a memory location in the heap.

3.8.1 Mutability of Aggregates

Array types and by-value structure types are aggregate types. While all fields of an array are of like type, structures may contain a combination of mutable and immutable constituent fields. An instance of aggregate type is mutable as a whole exactly if all of its contained constituent fields are mutable:

```
(def p (pair mutable(#\c) 3:int32))
...
// legal, field is mutable:
(set! p.first #\d)
// illegal
(set! p.second 5)

(def mp (pair mutable(#\c)
          3:mutable(int32)))
// legal, all fields mutable:
(set! mp (pair #\d 4))
```

The test of constituent mutability does not extend across reference boundaries.

If *T* is an aggregate type, then `mutable(`*T*`)` is a valid type exactly if *T* is mutable at all constituents.

3.8.2 Shallow vs. Deep Mutability

If *T* is an immutable type, and if all of its unboxed fields (recursively) are of immutable type up to `ref` boundaries, then *T* is said to be **shallow immutable**. If any of those elements are mutable, then *T* is said to be **shallow mutable**. We use the term **deep mutable** to refer to mutable types that appear *behind* a `ref` boundary.

The type `boxed(mutable('a))` is shallow immutable but deep mutable.

3.9 Exceptions

BitC provides declared exceptions. The type `exception` should be viewed as an "open" union reference type whose variant constructors are defined by `exception`. The syntax of an exception declaration is:

(exception *nm* [$field_1$ ... $field_n$])

where each $field_i$ is a field declaration (see `struct`) whose type is a concrete type.

An identifier bound to an exception name is a procedure that may be used to instantiate new instances of that exception. The arguments to this procedure are the values of the fields associated with the exception.

3.10 Copy Compatibility

The combination of mutability and value types in the BitC type system raises the need to specify what happens at "copy boundaries." Given a value of type T_1 and a location or formal parameter (the receiver) of type T_2, when is the value compatible with the receiver for purposes of argument passing and assignment? We refer to this as **copy compatibility**.

3.10.1 Trivial Copy Compatibility

The types T and `mutable(`T`)` are trivially copy compatible, because they differ only in top-level mutability. A location of type `mutable(`T`)` may be assigned a value of either type, and a parameter of type T may be passed a value of either type.

The intrinsic type class `(top-copy-compat` T_1 T_2`)` describes a relation between all pairs of types T_1 and T_2 that are trivially copy compatible. This type class is rarely the right thing to use in input programs, but may sometimes be seen in the type checker output.

3.10.2 Structural Copy Compatibility

Two structured types T_1 and T_2 are structurally copy compatible if (a) they are trivially copy compatible or (b) they are value types that are fieldwise structurally copy compatible. Note that this definition explicitly does *not* descend recursively across reference types. The conceptual intuition is this: any element that will actually be copied by assignment or argument passing must be compatible ignoring mutability, but any object that is *pointed to* must have exactly matching type in both the value and its receiver.

The intrinsic type class `(copy-compat` T_1 T_2`)` describes a relation between all pairs of types T_1 and T_2 that are structually copy compatible. If you are trying to abstract over mutability, this type class is usually the one that you want. Note that `(top-copy-compat` T_1 T_2`)` implies `(copy-compat` T_1 T_2`)`, but the reverse is not true.

3.10.3 Inner and Outer Procedure Types

A curious consequence of copy compatibility is that functions have two types. Consider the function:

```
(def (inc x:mutable(int32))
  (set! x (+ x 1))
  x)
```

From the perspective of the function's implementation, `x` is a mutable location having type `mutable(int32)`, and since `x` is returned, the return type of this function is also `mutable(int32)`. From this, we would conclude that the type of `inc` should be:

```
inc: fn (mutable(int32) -> mutable(int32)
```

Given the copy compatibility rules, however, the fact that `inc` internally mutates its argument is not something that the caller needs to know in order to call `inc` directly. The externally observable type reported for `inc` therefore strips shallow mutability, giving:

```
inc: fn (int32) -> int32
```

In addition to preserving abstraction, reducing type incompatibilities at function reference types, and providing some degree of separation of concerns, copy compatibility can also be exploited by polyinstantiating implementations to significantly reduce the amount of redundant instantiation that would otherwise be required.

3.11 Restrictions

BitC imposes a value restriction [4] on polymorphism. A binding is only permitted to be of polymorphic type if its defining expression is a syntactic value.

As is usual in let-polymorphic languages, polymorphic function arguments cannot be used polymorphically within the function. For example, the following function is disallowed:

```
(def (foo f)
  (pair
    (f (cons 1 (cons 2 nil)))
    (f (cons true (cons false nil)))))
```

4 Type Classes and Qualified Types

A **type class** defines an n-ary relation on types, and provides a means for specifying *ad hoc* polymorphism. Every type class is parameterized over $n \geq 1$ types, and defines a set of methods over those types. Type classes provide a form of *open* type-directed operations: a user can add a new member to the relation established by a given type class by providing a new instantiation of the type class.

Closely connected with type classes is the notion of **qualified types**. For example, consider the following definition of `list-max`:

```
(def (list-max x)
  (switch tmp x
    (nil (raise ValueError))
    (cons
     (if (null? tmp.cdr)
         tmp.car
         (let ((m (list-max tmp.cdr)))
           (if (>= tmp.car m)
               tmp.car m))))))
```

which is typed as:

```
fn (list('a)) -> 'a
  where Ord('a)
```

This type should be read informally as "`list-max` is a procedure accepting lists of type `'a` and returning a value of type `'a`. It is defined over all types `'a` such that there is an instantiation of the `Ord('a)` type class."

In this example, `Ord('a)` is the type class that describes types having a total order. That is: types over which the procedure >= is defined. Obviously, it not semantically sensible to request the greatest element of a list whose element type does not have at total ordering.

Contrast this example with the following alternative:

```
(def (list-max gte x)
  (switch tmp x
    (nil (raise ValueError))
    (cons
     (if (null? tmp.cdr)
         tmp.car
         (let ((m (list-max tmp.cdr)))
           (if (gte tmp.car m)
               tmp.car m))))))
```

which is typed as:

```
fn (fn ('a, 'a) -> bool,
    list('a)) -> 'a
```

In this second example, the comparison operator is provided as an argument, and there is no requirement for additional type constraints. Note, however, that in practice any comparison function that might actually be passed in this position is likely to depend on the <= operator in some fashion, and is therefore likely to end up having a qualified type.

4.1 Definition of Type Classes

A type class is defined by the abstract syntax:

```
(trait (nm tv ... tv)
  [tyfn-declarations]
  [:closed]
  method-definitions)
```

where a *tyfn-declaration* is a statement of functional dependency between types [6]:

```
(tyfn (tv ... tv) tv)
```

and each method definition takes the form:

```
nm : function-type
```

Each method is an abstract procedure that may be instantiated for some particular type by a later use of `instance`. The method may be invoked prior to the point where the instantiation is visible. Each method defined by a type class is introduced into the scope containing the type class definition.

By providing an instantiation of a class over some particular set of types, the programmer simultaneously proves (by example) that the set of types is a member of the class and defines (by example) how the operations of the class are implemented for that type. If the type class has been marked "closed," the instance definition must appear in the same interface or module that contains the type class definition.

Type functions, when present, indicate that there is a dependent relationship between two or more types of the type class relation. For example, the (incomplete) declaration:

```
(trait (sample 'a 'b 'c)
  (tyfn ('a 'b) 'c)
  ...)
```

states that `sample` is a type relation over three types, but also says that for any pair of types `'a` and `'b` there there is one valid choice of `'c`.

4.1.1 Example: `Eql`

As a first example, consider the equality comparison operations. The type class `Eql` defines a single element type relation on types `'a`: describing whether the type is admissable under equality. Some types — notably function types — cannot be compared for equality. The definition of this type class is written:

```
(trait Eql('a)
  == : fn ('a, 'a) bool)
  != : fn ('a, 'a) bool)
```

which states that `Eq.` is the single element type relation over all types `'a` that can be passed as arguments to `==` and `!=`.

4.1.2 Qualification: `Ord`

A type class can also be introduced in qualified form. The syntax for such a type class definition is:

```
(forall (constraint ... constraint)
  (trait
     (nm tv ... tv))
     [tyfn-declarations]
     method-definitions)
```

where each constraint takes the form:

```
(tc-name tv ... tv)
```

where `tc-name` is a typeclass name. An example of this use is the `Ord` type class:

```
trait Ord('a)
  where Eql('a)
is < : fn ('a, 'a) 'a
```

This type class states that `Ord` is the single element type relation over all types `'a` that can be passed as arguments to <. It also states that the `Ord` relation is only defined for types that are also members of the `Eql` relation (that is: types that admit equality comparison).

Note that in the presence of this definition, the procedures >, <=, and >= can be defined as:

```
(def (> x y)
  (not (or (< x y) (== x y)))
(def (<= x y)
  (or (< x y) (== x y)))
(def (>= x y)
  (or (> x y) (== x y)))
```

all of which will be inferred to have the type:

```
fn('a, 'a) -> bool
  where Ord('a)
```

This may seem like a very long-winded way of saying that an orderable type is any type that can be passed to the operators < and ==. However, type classes are statements about *relations* among types. This may become clearer with the following example.

Note that because `Ord('a)` has `Eql('a)`, the types:

```
fn('a, 'a) -> bool
  where Ord('a), Eql('a)
fn('a, 'a) -> bool
  where Ord('a)
```

are equivalent. The second is stylistically preferred for reasons of brevity. It is also more robust: in the (in this example unlikely) event that the definition of `Ord` should be modified to depend on some other type class in place of `Eql` the future, the first definition will mistakenly retain an additional, unncessary type dependency, while the second will continue to type check as intended.

Restriction: Qualified type relationships must be acyclic.

4.1.3 Example: `tyfn`

Need an example of type functions.

4.2 Instantiation of Type Classes

Whenever a type class method is invoked, the compiler must identify some concrete member of the type class relation that is sufficient to choose an appropriate implementation of that method. This is done by locating an appropriate instantiation.

A type class instantiation is a demonstration by example that some particular set of types satisfies the relation required by the type class. Type class instantiations are defined by the `instance` form. The abstract syntax of `instance` is:

```
(instance tc-instance
  function ... function)
(forall (constraint ... constraint)
  (instance tc-instance)
    function ... function)
```

where *tc-instance* takes the form:

```
(typeclass-name type ... type)
```

For example, the definition:

```
(instance Ord(int32)
  int32-ops.<)
```

states that `int32` is member of the type relation `Ord` because there is an instance function `int32.<` that provides an implementation of the "less than" operation over arguments of type `int32`. If the type class definition is closed, all instance definitions must occur in the same interface or module ad the type class definition.

In practice, this definition is insufficient, because we must first demonstrate that `int32` is a member of the `Eq` relation (which is a superclass of `Ord`) In consequence, two separate instantiations are required:

```
(instance (Eq int32)
  int32-ops.==
  int32-ops.!=)
(instance Ord(int32)
  int32-ops.<)
```

A type class instantiation is deemed to be in scope for purposes of procedure instantiation if it is defined by the end of the outermost unit of compilation.

It is a compile time error to define two type class instances covering the same concrete types unless one instance is "preferred" to the other. Preference is determined by comparing the respective type variable instantiations positionally. Given two instances A and B over type variables $\texttt{tv}_1 \ldots \texttt{tv}_n$, instance A is preferable to instance B if there exists some subset of the respective type variable instantiations such that the instantiation under A is strictly more concrete than the instantiation under B, and the two instantiations are identical (modulo type variable renaming) at all other positions. If this comparison does not (transitively) determine a most preferred instantiation, then no instantiation is preferred and a compile time error is signalled.

4.3 Qualified Types

Constraints are now permitted only as the outermost form. This section needs to be updated accordingly.

It is sometimes necessary to qualify the types of instances, type classes, constructed type definitions, or value declarations explicitly. A qualified type takes the general form:

```
(forall (constraint ... constraint)
        type)
```

Qualified types may appear only as the types of binding patterns; they may not qualify expressions generally. For example:

```
(def add1:(forall ((Num 'a))
                  (fn 'a 'a))
  (lambda (x) (+ (the 'a 1) x)))
```

explicitly states that the `add1` procedure takes arguments whose type admit $+$, and therefore must be members of the `Num` type class.

If multiple qualifications appear in the same binding pattern, they must unify. The following is legal, if somewhat obscure:

```
(def
  (v1:(forall (Eql('a)) 'a), v2:'b)
  : (forall ((Num 'c))
      ((fn ('c) bool), 'b)) ...)
```

with the effect that `v1` receives the qualified type:

```
(forall (Eql((fn ('c) bool))
         (Num 'c))
        (fn ('c) bool))
```

which will ultimately fail to type check, because functions are not admissable under value equality.

Qualifications may also be applied to structure and union declarations, with the abstract syntax:

```
(forall (constraints)
  (struct (struct-name tvars) [:unboxed])
    nm1[:t1] ... nmn[:tn])
(forall (constraints)
  (union (union-name tvars) [:unboxed])
    C1 ... Cn)
```

Qualifications may similarly appear in the binding patterns of structure, union, and value declarations.

4.4 Core Type Classes

BitC defines several core type classes. These classes cover type relations that are required internally by the type checker, or in some cases relations that cannot be expressed within the language. All of these type classes are closed, though not necessarily finite — the compiler implements their membership internally.

4.4.1 `ref-types`

`RefTypes('a)` is the type class consisting of all heap-allocated types: `boxed('a)`, `(vector 'a)`, and `string`. Use of this type class is appropriate when a structure or union should not be instantiated over value types. The `nullable('a)` type is an example of this.

4.4.2 `copy-compat`

`(copy-compat 'a 'b)` is an equivalence relation containing all pairs of types `'a` and `'b` that are "copy compatible". That is: all types for which a value of type `'b` may be assigned to a location of type `mutable('a)`, and all types for which a formal parameter of type `'a` may be passed an actual parameter of type `'b`.

4.4.3 `top-copy-compat`

`(top-copy-compat 'a 'b)` is an equivalence relation containing all pairs of types `'a`, `'b` such that `'a=='b`, `mutable('a)=='b`, or `'a==mutable('b)`. That is: types that are the same ignoring top-level mutability.

5 Binding of Values

5.1 Binding Patterns

Binding patterns are used to bind names to values. They appear in the definition of formal parameters and in binding forms such as `def`, `let`, `letrec`, and `do`. A binding pattern consists of an identifier that is optionally qualified by a type:

```
id
(the T id)
id : T
```

In top-level bindings (those introduced by a top-level `def`, the *id* may be qualified by an interface binding name corresponding to some interface that the current unit of compilation provides (Section 9.3). Thus, if `my.interface` is an interface name, it is legal for a source unit of compilation to contain:

```
// State that we are a provider
// of my.interface:
provide if my.interface
//...
// Define some variable declared
// in the interface:
(def (if.varname x)
  (+ x 1))
```

5.2 def

Variable and procedure bindings are introduced by `def`:

```
(def bp e)
(def (id [bp1 ... bpn])
  e ... e)
```

where each *bp* is a binding pattern. In the first form, the newly bound identifiers are not in scope within the body. The second form permits recursive bindings. Identifiers defined within a recursive `def` are deemed "incomplete" until the end of the enclosing `def` form.

The right hand form of a `def` is evaluated to obtain a value, which is then bound to the identifier on the left-hand side.

Mutually recursive procedure definitions at top level can be achieved either by use of `letrec` or by declaring the procedures ahead of their definitions.

5.3 Local Binding Forms

5.3.1 let

The `let` form provides a mechanism for locally binding identifiers to the result of an expression evaluation. Each identifier bound in a `let` form must appear exactly once among the collection of binding patterns being bound. Evaluation of the initialization expressions occurs in order from e_1 to e_n. The environment in which the expression(s) are evaluated does not contain the identifiers being bound in the current `let` form.

The syntax of `let` is:

```
(let ((bp1 e1)
      ...
      (bpn en))
  ebody-1
  ...
  ebody-n)
```

One common form of these expressions is the one in which the left hand patterns are simple identifier names, as in:

```
(let ((x e1)
      ...
      (y e2))
  ; x, y are bound in:
  ebody-1
  ...
  ebody-n)
```

The value of a `let` form is the value of the last form executed within the body.

In similar languages, `let` is often presented as a form derived from `lamdba`. In BitC, as in other let-polymorphic languages, the value restriction for lambda arguments means that this is not (quite) true.

5.3.2 letrec

The `letrec` form provides a mechanism for locally binding identifiers to an expression value. Each identifier bound in a `letrec` form must appear exactly once among the collection of binding patterns being bound.

Evaluation of the initialization expressions occurs in order from e_1 to e_n. The syntax of `letrec` is:

```
(let ((bp1 e1)
      ...
      (bpn en))
  ; Identifiers in bpi
  ; are bound in:
  ebody-1
  ...
  ebody-n)
```

The environment in which the expression(s) are evaluated contains (via unification) the identifiers being bound in the current `letrec` form. This allows `letrec` to bind recursive procedure definitions:

```
(letrec
  ((odd
     (lambda (x) ; odd
       (cond ((= x 0) false)
             ((< x 0) (odd (- x)))
             (otherwise
               (not
                 (even (- x 1)))))))
   (even
     (lambda (x) ; even
       (cond ((= x 0) true)
             ((< x 0) (even (- x)))
             (otherwise
               (not
                 (odd (- x 1))))))))
  body)
```

The value of a `letrec` form is the value of the last form executed within the body.

Within the defining expressions of a `letrec` form, use of the identifiers being defined is subject to the same restrictions described for `def`. This ensures that cyclical constant data cannot be introduced.[7]

5.3.3 local defininitions

The `def` form may be used to introduce local definitions in any expression sequence, provided the local definition is not the last form of the sequence. For this purpose, the bodies of `begin`, `lambda` (including those implied by derived form rewrites), `let`, `letrec`, `while`, or `do-while` constitute expression sequences.

Local `def` is a derived form. The canonical rewriting of the local `def` form using core language constructs is:

```
(begin ...
  (def id edef) e2 [...]) =>
(begin ...
  (let ((id edef))
    e2 [...]))
(begin ...
  (def (id [args]) edef) e2 [...]) =>
(begin ...
  (letrec ((id (lambda(args) edef)))
    e2 [...]))
```

This rewrite proceeds left to right. Successive definitions are gathered into `let` or `letrec` forms that are progressively more deeply nested, which means that later local definitions of an identifier shadow earlier definitions.

5.4 Value Non-Recursion

In any recursive binding (introduced by `letrec` or `def`) such as:

```
(def bp e)
```

if *`id`* is an identifier that appears in the binding pattern (and is therefore incomplete), free occurrences of *`id`* in *`e`* must occur only within a `lambda` body. This ensures that *`id`* will be initialized before it is used.

This restriction intentionally prevents infinitely recursive data constant definitions.

5.5 Static Initialization Restriction

I continue to look for a more rigorous way to express the following requirement.

Statically declared (global) variables must be initialized before the main entry point is entered. This presents a challenge of specification. The language definition must impose a sufficient ordering constraint on initializations to ensure that no initializer can depend (transitively) on any uninitialized variable. To ensure this, we introduce the notions of "compile-time evaluable" and "compile time applicable" expressions, and the restriction that every initializing expression of a statically declared variable must be compile-time evaluable.[8] Informally: it must be possible for the compiler to evaluate the initializing expression at compile time without (conservatively) referencing any uninitialized variable.

Literals are compile-time evaluable.

[7] Cyclical constants impede termination reasoning in the prover.

[8] This notion is conceptually related to the Standard ML notion of "syntactic constants," and achieves the same goal. The definition of "compile-time evaluable" is slightly richer, and allows for more expressive initializing expressions.

A locally bound identifier is compile-time evaluable exactly if its initializing expression is compile-time evaluable. It is compile-time applicable exactly if the return value of its defining expression is compile-time applicable.

A globally bound identifier is compile-time evaluable provided its definition is lexically observable and compile-time evaluable. By “lexically observable,” we mean that either (a) it appears as a lexically preceding definition in the same unit of compilation, or (b) there exists some chain of interfaces $I_0 \ldots I_n$ such that the global identifier is defined in I_n, the unit defining `out` imports I_0, I_0 imports I_1, I_1 imports I_2 ... and I_{n-1} imports I_n.

A globally bound identifier is compile-time applicable provided it is of function type, it is lexically observable, and all expressions appearing in its defining `lambda` form are compile-time evaluable. For purposes of this analysis, it is assumed that any formal parameter of the function is both compile-time evaluable and (if of function type) compile-time applicable.

Any expression *other than* an application or an assignment is compile-time evaluable provided that all of its free identifiers are compile-time evaluable.

An application is compile-time evaluable provided that (a) the expression in the function position is compile-time evaluable, (b) all of its arguments are compile-time evaluable, and (c) any arguments of function type are compile-time evaluable.

An assignment (as with `set!`) is compile-time evaluable provided its expression is *both* compile-time evaluable and (if of function type) compile-time applicable. This prevents later assignments from altering the compile-time evaluability of previously defined identifiers.

Dangling:

The result of an expression evaluation (including application and constructor application) is observably known if (a) the definitions of all identifiers that are free in the expression are observably known, and (b) any procedure that is applied is observably applicable. Requirement (b) is satisfied by definition for all type constructors.

Note that these definitions are conservative with respect to mutability. Because no initializing expression can reference an observably unknown value, nor perform an application that is not observably applicable, it follows that no assignment performed from within an initializing expression can cause an identifier to transition from observably known to observably unknown.

6 Declarations

The `proclaim` form is used to provide opaque value declarations. The declaration:

```
(proclaim x:int32)
```

states that `x` is the name of a value of type `int32` whose definition and initialization is provided by some implementing unit of compilation. This form can legally appear only at top level within a source unit of compilation or within an interface.

The identifier declared by a `proclaim` form is considered incomplete. If a completing definition is later provided within the same compilation unit, the identifier is considered complete in the balance of the defining compilation unit after the the close of its defining form. An incomplete declaration may be used within a procedure, but may not be used as part of a top-level initializer (see `def`, Section 5.2).

It is occasionally necessary to make reference to procedures or values that are implemented by an externally provided runtime library. This may be accomplished by an `external` declaration:

```
(proclaim proc:(fn (int32) char)
          external)
(proclaim proc:(fn (int32) char)
          external ident)
```

This has the effect of advising the BitC compiler that no definition of this identifier will be supplied in BitC source code. It is primarily intended to support portions of the BitC runtime library. Use of this mechanism for other purposes is strongly discouraged, and we reserve the right to revise this syntax incompatibly in future revisions of the BitC specification.

If a proclaimed external procedure provides an optional trailing `ident`, this identifier will be used verbatim in the generated code in place of the normal identifier name generated by BitC. The trailing identifier is permitted only if the external procedure has non-polymorphic type.

7 Expressions

7.1 Literals

Every literal is an expression whose type is the type of the literal (as described above) and whose value is the literal value itself.

7.2 Identifiers

Every lexically valid identifier is an expression whose type is the type of the identifier and whose value is the value to which the identifier is bound.

7.3 `sizeof`, `bitsizeof`

The `sizeof` and `bitsizeof` forms report the size, in bytes (respectively bits), of a type. When applied to expressions, they report the size of the *type* of that expression. The expression is typed by the compiler, but it is not evaluated.

```
sizeof(e)
sizeof(T)
bitsizeof(e)
bitsizeof(T)
```

The return type of `sizeof`, `bitsizeof` is `word`.

7.4 Type-Qualified Expressions

Any expression *e* may be qualified with an explicit result type by writing either of

```
(the T e)
e : T
```

where *T* is a type. This indicates that the result type of the `the` form is constrained to be of type *T*. The `the` form is syntax, its expression argument is not conveyed by application, and is therefore not subject to copying as a consequence of type qualification.

The result *value* of the expression is not changed by type qualification, except to the extent that a type restriction may lead the inference engine to resolve the types of other expressions and the selection of overloaded primitive arithmetic operators in ways that produce different results.

Syntactic Restriction The `e:T` convenience syntax is not permitted in combination with the member selection convenience syntax ".". The sequence of grammar expansions:

```
expr -> expr.Id
expr -> expr:type.Id
expr -> expr:Id.Id.Id
               ^
```

leads to a shift/reduce conflict at the indicated position. The grammar resolves this by disallowing the helper type-qualification syntax in this context. If required, a type qualification in this context can be obtained using either of the following alternatives:

```
(the T e).Id
(member e:T Id)
```

7.5 Value Constructors

7.5.1 unit

The expression:

```
()
```

denotes the singleton unit value.

7.5.2 MakeVector

The expression:

```
(MakeVector e_len e_init)
```

creates a new vector whose length is determined by the value of the expression e_{len}, which must evaluate to a value of type `word`. The argument e_{init} must be a function from word to some type *T*, where the vector created will be of type `(vector T)`. The initializer value for each cell will be obtained by invoking the procedure e_{init} a total of e_{len} times, passing as an argument the index of the vector position to be initialized. The procedure e_{init} should return the desired initializer value for the corresponding position.

For example, the procedure `list->vector` may be written as:

```
import bitc.list as ls
(def (list->vector lst)
  (MakeVector
    (length lst)
    (lambda (n)
       (ls.list-nth lst n))))
```

Care should be taken to ensure that the type returned by the initializer function is mutable if the slots of the vector are intended to be mutable.

7.5.3 array, vector

The expressions:

```
(array e_0 ... e_n)
(vector e_0 ... e_n)
```

create a new array (respectively, vector) whose length is determined by number of arguments. The first argument expression becomes the first cell of the created array (respectively, vector), the second becomes the second, and so forth. All expressions must be of like type.

7.5.4 Convenience Syntax

Derived forms

The following are right-associative convenience syntax for types defined in the standard prelude:

```
(a,b) => (pair a b)
(a,b,c) => (pair a (pair b c))
```

7.6 Expression Sequences

The expression:

```
(begin e1 ... en)
```

executes the forms e_1 through e_n in sequence, where each form is an expression. The value of a `begin` expression is the value produced by the last *expression* executed in the begin block.

7.7 Labeled Sequences and Escape

The expression:

```
(block ident e1 ... en)
```

executes the forms e_1 through e_n in sequence, where each form is an expression. The value of a `block` expression is the value produced by the last *expression* executed within the block.

Within the body of the `block` form, the identifier *`ident`* is lexically bound as an escape label, and the expression

```
(from ident return e)
```

Causes an immediate return from the `block` with the value computed by the expression e. Control does not continue past the end of this form.

The identifier *ident* must be in scope as an escape label, and the `block` and its associated `return-from` must appear within the body of the same lambda form. That is: the `return-from` may *not* appear within a `lambda` that is in turn nested within a `block`.

7.8 Iteration

Derived form

BitC provides the looping construct `loop`, which conditionally evaluates its body multiple times.

```
(loop ((bp1 einit-1 estep-1)
       ...
       (bpn einit-n estep-n))
   (etest eresult)
  ebody-1
  ...
  ebody-n)
```

Do is an iteration construct taken from Scheme [8]. It specifies a set of variables to be bound along with an initializer expression and an update expression for each variable. Evaluation of the `loop` form proceeds as follows:

The $e_{init\text{-}i}$ expressions are evaluated in order in the lexical context containing the `do` form. In this context, the variables bound by the loop have not yet been bound. All other expressions are evaluated within an inner lexical context that includes the `loop`-bound variables. After all of the initialization values are computed in order, the `loop`-bound variables are bound to the initial results in parallel, and body processing begins.

At the start of each pass over the body, the expression e_{test} is evaluated. If this expression returns **true**, then e_{result} is evaluated and its result returned. Otherwise, the expresions of the body are evaluated in sequence.

At the end of each execution of the loop body, the $e_{step\text{-}i}$ expressions are evaluated in sequence. Once all of the expression values have been evaluated, the `loop`-bound variables are bound to the newly computed results in parallel and a new pass is initiated over the loop body as previously described.

The execution of a given pass of the loop body can be terminated immediately by the:

```
(continue)
```

form. This causes an immediate transfer of control to the end of the nearest enclosing loop body. Note that the initializer, step, test, and result expressions are *not* part of the loop body.

The `loop` form is not let-polymorphic. In consequence, the binding patterns bound within the `do` form are not polymorphic bindings.

7.9 Interface Member Reference

If *`if`* is an identifier naming an interface binding established through `import`, and *`id`* is an identifier defined in that interface, then either of:

```
(member if id)
if.id
```

is an expression that returns the value of that identifier. The returned value is a location, and can be used as an argument to `set!`.

7.10 Structure, Repr Field Reference

If e_{loc} is a location expression of structure or repr type, and *field* is an identifier naming some invariant field in that type then either of:

```
(member e_loc field)
e_loc.field
```

is an expression that returns the field value. `member` is a syntactic form. The returned value is a location, and can be used as an argument to `set!`.

7.11 Union, Repr Tag Reference

If e_{loc} is a location expression of union or repr type, and *tagid* is an identifier naming some union discriminator tag in that union or repr type then either of:

```
(member e_loc tagid)
e_loc.tagid
```

is a boolean expression that returns true exactly if the tag value of the corresponding tag is `tagid`.

7.12 Array and Vector Expressions

7.12.1 Array, ArrayRef, and Vector lengths

If *e* is an expression of array, ArrayRef, or vector type, then

```
e.length
```

returns a `word` whose value is the number of elements in the array, ArrayRef, or vector.

7.12.2 Array, ArrayRef, and Vector indexing

If *e* is an expression of array, ArrayRef, or vector) type, and e_i is an expression with result type `word`, then:

```
e[e_i]
```

returns the e_i'th element of the array, ArrayRef, or vector. If the value e_i is greater than or equal to the length of the array, ArrayRef, or vector), then a `IndexBoundsError` exception is thrown.

Note that type inference for these types is currently incomplete. In the absence of declaration, the type `vector` will be inferred for *e*. Since the type `ArrayRef` can only be declared at parameters and is never inferred, a surprising inference result probably means that something needs to be declared as an array type. Future extensions of BitC are expected to provide generalized accessors, after which this inconvenience will be (backwards compatibly) resolved.

7.13 Procedure Values

Procedure values are introduced by the keyword `lambda`. In contrast to Scheme, Haskell, and Standard ML, BitC procedures take zero or more arguments. The syntax of a procedure definition is:

```
(lambda ([bp_1 ... bp_n]
  e_1 ... e_n)
```

where each bp_i is a binding pattern matching the formal parameters of the procedure and $e_1 ... e_n$ is the body of the procedure. The return value of the procedure is the value computed by the last expression executed in the body.

Each formal argument binding pattern defines a set of variable bindings that are in scope in the body of the lambda. Each formal argument binding pattern is unified with its corresponding actual parameter. Any identifier that is free in the binding pattern is unified with the structurally corresponding element of its associated actual parameter.

BitC argument and return value passing are “by value.” Formal argument and return values must be of value type, which means that *references* can be passed, but the values denoted by these references cannot. The “by value” policy also implies that local variables are *copies* of their initializing expressions, which may yield surprising results if the initializer is of mutable type. A `let` binding is not an alias for its initializer. A `let` binding of a (top level) mutable value cannot simply be substituted by β-reduction into the body of the `let` form.

7.13.1 By-Reference Parameters

By-reference parameters provide an optimized argument passing mechanism for parameters. A by-reference formal parameter is an *alias* of the passed argument; the internal implementation passes a pointer to the argument rather than a copy of the argument. A by-reference parameter may be a reference to an component of an aggregate type, such as a field or a vector member.

The BitC specification permits the representation of a by-reference parameter to be either one word or two. This is intended to simplify the handling of inner pointers by the garbage collector.

By-reference parameters can escape only as part of a first-class procedure, but the lifetime of a by-reference parameter cannot exceed the lifetime of its containing scope.

The formal parameters of a function can be declared as by by-reference parameters as in:

```
(lambda (x:(ByRef τ) ...) ...)        // or
(def (f x:(ByRef τ) ...) ...)
```

A `ByRef` declaration can only appear as a qualifier for the type of a parameter. This is a syntactic restriction.

A function with a formal parameter declared as `(ByRef` τ`)` can only be apllied to an actual argument of type τ. That is, unlike normal parameters, an actual argument of type `mutable(`τ`)` where the formal parameter is of type τ or *vice versa* is not permitted [Here, $\tau \neq$ mutable(τ')].

7.14 Explicit Procedure Return

Derived form

The expression:

```
(return e)
```

causes the nearest enclosing `lambda` form to immediately return the value computed by the expression e. This form executes a form of labeled break. Control does not continue past the end of this form.

Derivation The canonical rewriting of `return` requires that the containing `lambda` also be rewritten:

```
(lambda (args) body) =>
(lambda (args)
  (block __return body))

(return e) =>
(from __return return e)
```

7.15 Function Application

The expression:

```
(e_fn [e_1 ... e_n])
```

denotes function application. The evaluation of the expression e_{fn} must yield a procedure value.

Note that the identifier fn may either evaluate to a procedure or may name a value constructor for a named constructed type.

7.16 Conditional Execution

7.16.1 if

Derived form

The `if` form is used to represent conditional control flow:

```
(if e_test e_then e_else)
```

Where e_{test}, e_{then}, and e_{else}, are BitC expressions.

The value of an `if` form is either the value of the e_{then} form or the value of the e_{else} expression. Exactly one of the e_{then} or e_{else} forms is evaluated.

The value returned by the e_{test} expression must be of boolean type.

The types of the e_{then} and e_{else} must be compatible.

Derivation The canonical rewriting of `if` is:

```
(if e_test e_then e_else) =>
(case e_test
  (true e_then)
  (false e_else))
(if e_test e_then) =>
(case e_test
  (true e_then ())
  (false ()))
```

7.16.2 when

Derived form

The `when` form is used to represent conditional control flow when only one condition is of interest:

```
(when e_test e_then ...)
```

Where e_{test} and e_{then} are BitC expressions.

The e_{test} expression must compatible with boolean. There are no restrictions on the types of the e_{then} forms. The type of a `when` form is Unit.

The e_{then} forms are evaluated only if the value of the e_{test} form is `true`.

Derivation The canonical rewriting of `when` is:

```
(when e_test e_then ...) =>
```

```
(case e_test
  (true e_then ... ())
  (false ()))
```

7.16.3 not

Derived form

The `not` form is used to invert a boolean result. The form:

```
(not e)
```

returns true if its argument evaluates to false, and false it its argument evaluates to true.

Derivation The canonical rewriting of `not` is:

```
(not e) =>
(if e false true)
```

7.16.4 and

Derived form

The `and` form is used to perform lazy expression evaluation. The form:

```
(and e1 e2 ... en)
```

returns true if every one of the expressions e_1 ... e_n evaluates as true. Expressions are evaluated left to right. Each expression must return a result of type `bool`. If any expression evaluates as `false`, no further expressions are evaluated. For this reason, the `and` form cannot be implemented as a procedure.

Derivation The canonical rewriting of `and` proceeds by first rewriting multiargument `and` forms into forms of no more than two arguments:

```
(and e1 e2 ... en) =>
(and e1
     (and e2 ... en))
```

and then rewriting each two argument `and` form as:

```
(and e1 e2) =>
(if e1 e2 false)
```

7.16.5 or

Derived form

The `or` form is used to perform lazy expression evaluation. The form:

```
(or e1 e2 ... en)
```

returns true if any of the expressions e_1 ... e_n evaluates as true. Expressions are evaluated left to right. Each expression must return a result of type `bool`. If any expression evaluates as `true`, no further expressions are evaluated. For this reason, the `or` form cannot be implemented as a procedure.

Derivation The canonical rewriting of `or` proceeds by first rewriting multiargument `or` forms into forms of no more than two arguments:

```
(or e1 e2 ... en) =>
(or e1
     (or e2 ... en))
```

and then rewriting each two argument `or` form as:

```
(or e1 e2) =>
(if e1 true e2)
```

7.16.6 cond

Derived form

The `cond` form is used to represent conditional control flow where there are multiple possible outcomes:

```
(cond (e_test1 e1)
      (e_test2 e2)
      ; ...
      (otherwise en))
```

The $e_{test\text{-}i}$ expressions are evaluated in sequence until one of them evaluates as true. The corresponding e_i is then evaluated and its result becomes the value of the `cond` expression. Subsequent $e_{test\text{-}i}$ expressions are not evaluated. Exactly one of the e_i expressions will be evaluated. The `otherwise` clause is *not* optional.

Any `cond` form can be rewritten as a chain of `if` forms without alteration to meaning.

The values returned by the e_{test} expressions must be of type `bool`. All of the expressions e_i must be of compatible result types.[9]

Derivation The canonical rewriting of `cond` proceeds by removing each conditional expression in turn:

```
(cond (e_test1 e1)
      (e_test2 e2)
      ; ...
      (otherwise en)) =>
(if e_test1
    e1
```

[9] If we choose to relax the type compatibility rules for `if`, we should relax them here too.

```
      (cond (e_test2 e_2)
            ; ...
            (otherwise e_n)))
```

until only two cases remain in the `cond` expression, the last of which has a true predicate. This final cond is rewritten as:

```
(cond (e_test1 e_1)
      (otherwise e_n)) =>
(if e_test1
    e_1
    e_n)
```

7.17 Mutability

The expression:

```
(set! e_loc e_val)
```

is used to set the value of a mutable entity. The expression e_{loc} should evaluate to a location of mutable type `mutable(T)`. The expression e_{val} should evaluate to an assignment-compatible type *T*. The return value of `set!` is the unit value.

7.18 References

7.18.1 dup

If *e* is an expression of non-procedure type, the expression

```
(dup e)
```

returns a reference to a heap-allocated *copy* of the value returned by the expression *e*.

7.18.2 deref

If *e* is an expression of reference type `boxed(`τ`)`, then:

```
(deref e)
```

returns the value named by the reference. `deref` is a syntactic form. The returned value is a location, and can be used as an argument to `set!`.

The expression:

```
e^
```

is a convenience shorthand for

```
(deref e)
```

7.19 Value Matching

The `switch` form provides a mechanism for obtaining access to variant fields of a value of union or repr type. The syntax of `switch` is:

```
(switch id e
  (match_1 e_1.1 ... e_1.n1)
  (match_2 e_2.1 ... e_2.n2)
  ; ...
  (otherwise e_other))
```

where each *match* form is either a single union tag identifier (constructor) or a parenthesized sequence of union tag identifiers. Multiple union constructors may be matched by a single clause only if all matched constructors dominate identical fields. Since the type and bit-offsets of identically named fields within repr-constructors are required to be the same, multiple repr-constructors can be matched in a single clause. In this case, only the common fields of all matched repr-constructors will be visible for selection within $e_{i.i} \ldots e_{i.i}$.

A `switch` expression performs a value match on the tag fields of the expression *e* (or if *e* is of repr type, on the tags of its outermost body) in sequence. The first $match_i$ expression containining a matching tag value is selected, and the corresponding expression sequence $e_{i.1} \ldots e_{i.ni}$ is executed in an environment where `x` is a value of anonymous type. For every field of the original expresion type such that all of its containing union or repr tag qualifications are satisfied, the anonymous type contains a field with the same name denoting the same portion of the The value of `x` is a *copy* of the (discriminated) value returned by the expression *e*.

An expression of anonymous type may only appear only as the expression argument of the `member` form, or as the expression *e* of a `switch` form. It may not be passed as an argument, rebound, or returned as a result value.

If an otherwise form is present, then the body of the otherwise clause is executed in an environment where `x` is bound to a *copy* of the (undiscriminated) value returned by the expression *e*.[10]

If the matches performed by a given `switch` are exhaustive, the `otherwise` clause can be omitted.

For purposes of literal case analysis, the `switch` form will also accept expressions *e* of primary scalar type and matching values that are literals of the corresponding type.

[10] Technically, this need not be a copy, and we are reviewing whether the copy should be bypassed in the otherwise form.

7.20 Exception Handling

7.20.1 Try/Catch

The `try` form is used as the control flow resumption point of a `throw` form. When a `throw` occurs, control resumes at the nearest dynamically containing `try` form whose matching patterns match the name of the exception that was thrown.

The try block syntax is:

```
(try expr
  (catch id [(tagid1 e1)
             ...
             (tagid2 e2)
             ((tagidx tagidy) exy)]
             [(otherwise en)]))
```

In the absence of a programmer-specified `otherwise` clause, the `catch` block behaves as though the clause

```
(otherwise (throw id))
```

had been present.

If the evaluation of *expr* does not cause an exception, the value of the `try` block is the value of *expr*.

If the evaluation of *expr* causes an exception to be thrown, execution proceeds as if the catch block were rewritten to the procedure:

```
(lambda (e:exception)
  (switch nm e
    (tagid1 e1)
    (tagid2 e2)
    ...
    (otherwise eotherwise)))
```

and this procedure were applied to the received exception value. The return value from this procedure is returned as the value of the `case` expression.

7.20.2 Throw

The `throw` form is used to raise an exception. It performs a non-local control flow transfer to the most recent (nearest temporally enclosing) `try` block, with the effect that the thrown exception value is received by the corresponding `catch` block as described above. The `throw` expression has no return value type. The form:

```
(throw e)
```

throws the exception computed by the expression *e*, which must be an expression of type `exception` or of some concrete exception type. The latter case permits the locally bound identifier in a discriminated catch block to be passed directly to throw so that a pre-existing exception can be re-thrown without allocating new storage.

8 Locations

This section is a work in progress, but it is as accurate as I (shap) can currently make it. Corrections, comments, identification of omissions, and so forth are welcome.

BitC is a language supporting mutation. Because of this, a specification of the type system and expression evaluation semantics of BitC does not entirely account for how the behavior of `set!` interacts with the behavior of accessor expressions such as `a[i]`, `member`, `deref`, and expressions consisting of a single identifier. In particular, the characterization of `set!` as

```
(set! e1 e2)
```

does not account for how e_1 can be mutated in place, because the language specification (to this point) does not distinguish between expressions that generate new values (in the sense of values that occupy new storage) and expressions that return pre-existing values. To address this, we present here an informal characterization of locations in BitC.

8.1 Expressions Involving Locations

The following expressions accept locations (addresses of cells) in the indicated positions, and return locations as their result:

```
id
loc[ndx]
(member loc ident)
(deref e)
```

in addition, the `set!` form requires a location as its first argument, and returns the unit value.

```
(set! loc e)
```

8.2 Implicit Value Extraction

When a value of location type appears in any context expecting an expression, the location is implicitly dereferenced to give the expected value as a result. The “value extraction rule” applies both to return values and to applications, with the consequence that “bare” locations can never escape their binding frame in either the upward or downward directions. Only those forms identified explicitly above as accepting and returning locations are exceptions to the value extraction rule.

For example, in the expression:

```
(let ((a b)) ...)
```

the expression `b` evaluates (internally) to a location, but it is then discovered to appear in a binding context requiring an expression, so the value at that location is returned instead. Similarly, the expression `a` evaluates (internally) to a location, allowing it to be initialized in place.

8.3 Generalized Accessors

Note

This section describes a possible *future* enhancement to the language. It is considered experimental, and it is possible that it will never be implemented at all.

It is customary for programs that introduce "collection" types to provide operations for both insertion and lookup. It would be exceedingly convenient if the lookup operation could be used to support efficient access as well, for example:

```
(btree-insert bt key some-obj)
(btree-lookup bt key).field
```

That is, it is sometimes appropriate for the lookup function could return a location.

This cannot be supported for local objects, but it is possible for the type system to successfully infer the distinction between local object locations and global object locations. In this case, we could relax the value extraction rule so that it would *not* apply to return values, with the effect that we could write an accessor function such as:

```
(def (4th-elem vec)
  vec[4])
4th-elem: (fn ((vector 'a word))
              (location 'a))
```

Given such an accessor function, it would even be possible to write:

```
(set! (4th-elem vec) 5)
```

If introduced, this feature would need to be handled with care. It would be all too easy for a binary tree's lookup handler to return the internal node structure, with the effect that external code could modify the stored key "in place," violating the integrity of the binary tree. Because of this risk, it is unclear whether the type `(location T)` should ever be inferred automatically.

9 Interfaces

BitC recognizes two kinds of compilation units: interfaces and modules. An interface contains a public set of definitions and declarations. From the perspective of an importer, it describes the identifiers that are published by one or more providing bodies of code. From the implementor perspective, an interface describes a set of declarations that must be exported by some providing module. Interfaces provide the only means by which functions and types may be shared across multiple units of compilation.

A module contains a private set of definitions and delarations. In most cases, these are not visible outside of the scope of the module. The exception is when a module imports some interface and also declares explicitly that it provides definitions for one or more public declarations of that interface.

9.1 Specifying an Interface

An interface unit of compilation consists of a `bitc version` form followed by a single `interface` form. The `interface` form wraps a sequence of imports, aliases, definitions, and declarations that describe the public identifiers associated with that interface. For example, the interface:

```
interface sample {
  (def x 1) ; constant definition
  (union (list 'a):boxed
    nil
    (cons 'a (list 'a)))
  (struct (tree-of 'a):boxed)
  (proclaim y : int32))
  (struct S :opaque (int32 i))
}
```

Defines a constant `x` with value `1`, defines the now-familiar list type, declares that `tree-of` is an opaque reference type defined in some (unspecified) source unit of compilation, and that `y` is a value of type `int32` declared in some (unspecified) source unit of compilation.

Note that the declaration of `tree-of` provided by this interface is incomplete and therefore opaque. Because `tree-of` is a reference type, clients of this interface can declare variables and arguments of type `tree-of`, but cannot instantiate them because no function returning type `tree-of` is exposed by this interface.

Note further that `val-type` is both incomplete and undeclarable, because it is a value type. Clients may declare arguments of type

```
boxed(sample.value-type)
```

but not of type `value-type`, because the size of `value-type` is not revealed.

9.2 Importing an Interface, Aliasing

In order to use the identifiers supplied by an interface, the client unit of compilation must first import those identifiers using a top-level `import` form. There are three such forms. It is a compile-time error if any local identifier bound by an `import` is already bound.

9.2.1 Hygienic Import

The syntax of the hygienic import form is:

```
import if-name as local-name
```

where *`if-name`* is an interface name and *`local-name`* is an identifier to be bound in the current scope. If *`pubName`* is a name published by `TheInterface`, then after executing

```
import TheInterface as myName
```

it is legal to write `myName.`*`pubName`* at any identifier use occurrence. This is referred to as a **hygienic alias**. Hygienic aliases may appear in any use occurrence where an identifier might ordinarily appear. When a hygienic alias names a provided symbol, the hygienic alias may also appear as the defined identifier of a top-level definition. Hygienic aliases may *not* appear in the defined position of a *local* definition.

Hygienic import preserves a strong distinction between the namespace of the imported interface and the local namespace of the importing unit of compilation. This is appropriate when importing interfaces that are not fully mature, or for which the possibility of future name collisions as a result of interface evolution must be defended against.

9.2.2 Qualified Import

The qualified import syntax imports *selected* public identifiers from a specified interface. The selected identifiers are aliased (after optional re-naming) in the top-level namespace of the importing unit of compilation. The syntax of this form is:

```
import if-name ident-or-remap+
```

where *ident-of-remap* is either some identifier published by the imported interface or it is:

```
localName = pubName
```

If a single identifier is given, the local alias is bound using the public name. If the “as” variant is given, the local alias is bound under the specified local name instead.

It is a compile-time error to form more than one top-level alias in a single unit of compilation for the same public name in a given interface.

9.2.3 Promiscuous Import

The promiscuous import form imports all public identifiers from the imported interface that do not already have top-level aliases in the importing unit of compilation. The syntax of this form is:

```
import if-name
```

This form does not support identifier re-naming on import. Name collisions resulting from import can, if necessary, be managed by first performing a qualified import that re-maps the colliding public name, and then performing a promiscuous import to import the remainder of the interface.

9.2.4 Compile-Time Import Resolution

To locate the source representation of an imported interface, the compiler shall attempt to locate a file *`name`*`.bitc`, where *`name`* is the identifier used to name the corresponding interface. The default search path used for this resolution is not defined by this standard, but shall provide a resolution for every interface specified in the language definition. It is permissable for a compiler to implement some or all of the default search path internally, without reference to any external file name space.

Every file-based compilation environment for BitC shall provide a command-line option `-I` that enables the build environment to append directories to the interface search path.

9.2.5 Error Reporting

When reporting errors, a conforming BitC compiler should *always* report the defining name of the type or variable. It may *optionally* report the alias (use) name by which the type or value was referenced. Only defining names should be exposed for resolution by the linker. For identifiers defined or declared within an interface, the defining name is the fully qualified name of the identifier with respect to its interface. For all other identifiers, the defining name is the one that appears in the defining form.

The BitC interface system provides primarily for separate compilation and name hiding. In contrast to the module system of Standard ML [9], BitC interfaces are purely a tool for namespace control.

9.3 Providing an Interface

A source unit of compilation can indicate that it provides definitions for one or more declarations of an interface by means of the `provide` declaration. The syntax of provide is:

```
provide interface-name ident+
```

Where each *`ident`* is an identifier proclaimed by the named interface. That is: the name as specified in the interface rather than any alias of that name that may have been locally bound.

The effect of `provide` is to *authorize* the definition of the named identifiers. The definitions must then be defined by binding an arbitrarily selected local alias of the public identifier. For example:

```
bitc version "0.11+"
import sample as ln
provide sample TreeOf

(struct (ln.TreeOf 'a):boxed
  left : (optional
           (ln.TreeOf 'a))
  right : (optional
            (ln.TreeOf 'a))
  height
  value : 'a)
```

The requirement that an arbitrary alias be defined can result in strange appearances. The following alternative definition is equivalent in all respects to the one above:

```
bitc version "0.11+"
import sample as ln
provide sample TreeOf
(use (ln.TreeOf as mumble))

(struct (mumble 'a):boxed
  left : (optional (mumble 'a))
  right : (optional (mumble 'a))
  height
  value : 'a)
```

It is *not* required that a single source unit of compilation provide the entirety of an interface. For sufficiently large interfaces (e.g. the standard BitC library), this would be impractical. However the flexibility to define an interface with a collection of independently compiled source units of compilation demands some means to prevent circular type and value declarations. Circular value definitions are precluded by the type-level definition observability rule

9.4 The Reserved Interface `bitc`

The interface name "bitc" is reserved for use by the BitC implementation.

10 Source Modules

A source unit of compilation consists of one or more modules. Each module consists of a `module` form containing an arbitrary sequence of imports, definitions, declarations, and use forms that are *not* `interface` forms.

The module syntax is:

```
module module-name? docstring? { mod_form+ }
```

A source module constitutes a scope. Except for those definitions that are explicitly exported using `provide` (Section 9.3), identifiers bound in a module are not visible in other source modules.

11 Storage Model

This entire section had become hopelessly stale, and needs to be rewritten.

12 Pragmatics

12.1 Closure Construction

BitC seeks to enable the crafting of programs that do not make unexpected use of the heap, and which can make use of `lambda` and `letrec` forms to describe rich [mutual] tail recursions. Becuase of this, it is necessary to state the *minimal* degree of closure analysis that every BitC compiler is required to perform when constructing closures, and more generally, the conditions under which closures will be formed at all.

Closure construction proceeds in two phases. During the initial phase, free identifiers are added to the closure and the program is rewritten to heap-allocate closed values if that is necessary. During the second phase, a check is performed to determine whether the resulting closure is not actually necessary.

Phase 1 Given an identifier `id` appearing free in a `lambda` form `L`:

1. **Globals** If `id` resolves to a globally defined identifier, it will not be added to any closure record.
2. **Closed Lambda Forms** If `id:t` is an immutably bound identifier whose initializing form is a `lambda` term (i.e. a literal `lambda`, not merely an expression returning a value of function type), and `id` appears in L *in non-applicative position*, then `id`, then a corresponding field of type T is added to the closure record, and this field is populated at closure construction time by a *copy* of `id`.
3. **Shallow Immutables** If `id:T` is a locally bound identifier of shallow immutable type, then a corresponding field of type T is added to the closure record, and this field is populated at closure construction time by a *copy* of `id`.
4. **Shallow Mutables** If `id:T` is a locally bound identifier of shallow mutable type, then the program must be rewritten in such a way as to heap-allocate `id`, thereby converting it into a deep mutable value that is shallow immutable. The resulting reference `id:boxed(T)` is then closure converted as a shallow immutable identifier.

Phase 2 If a closure record was created in phase 1, but all elements of that closure were added as a consequence of rule 2 (closed lambda forms), then no explicitly allocatd closure record is either required or permitted. All of the closed lambda forms can be represented using labels without any intervening heap-allocated procedure objects.

Whether or not a closure record is fabricated for a given `lambda` form `L`, if an identifier `id` resolves to a closed lambda form, then any use-occurrence appearing in applicative position in `L` must be implemented by a call (or if tail recursive, jump) to the associated `lambda` form's *label* rather than proceeding through any procedure object that may have been allocated for `id`.

12.2 Tail Recursion

BitC requires a limited form of tail recursion. We do not require fully proper tail recursion because this is difficult to accomplish efficiently in C, and we wish to preserve the ability to compile BitC programs into C for the sake of portability.

Definition: Within a BitC form `f`, a form `g` occurs in **tail position** with respect to the form `f` if the evaluation of `g` is the final evaluation (and therefore the return value) computed by the form `f`. This definition is transitive. A structural consequence of this relationship is that the type of `g` is (copy compatible with) the type of `f`.

An application of a function `f` is said to be **tail recursive** if (a) it appears in tail position with respect to the body of its most closely containing `lambda` body, and (b) it is implemented in such a way as to re-use its containing stack frame.

The BitC specification *requires* that certain procedure calls appearing in tail position must be compiled using a tail-recursive implementation:

- Within a `letrec`, calls to any function bound in the letrec that appear in tail position within some function bound by the `letrec` must be tail recursive.
- Within any function `f`, calls to `f` that appear in tail position w.r.t. the body of `f` must be tail recursive. This is actually a special case of the first rule.

These requirements apply only to function calls whose destination can be statically resolved by the compiler at compile time. A BitC compiler is permitted, but is not required, to implement other function calls tail recursively.

II Standard Prelude

A range of types, type classes, and functions supporting operations on primary types are defined in the BitC standard prelude.

This section needs to be defined.

The following types and values are defined in the BitC standard prelude. The compiler is free to implement some or all of these types internally, and is further free to rely on internal knowledge of these types within the implementation.

13 Foundational Types

The prelude provides definitions for commonly used integral types. Under normal circumstances, the reader and pretty printer conspire to hide the fact that these types are union types.

```
// There is an open issue here: should
```

```
// strings be primitive? Issue is unicode
// character size and long strings.
// Strings:
//(union string:unboxed (vector char))

// Pairs:
(struct (pair 'a 'b):unboxed
  fst:'a snd:'b)

// Optional values:
(union (optional 'a):unboxed
  none (some value:'a))

// Nullable pointers:
unboxed union nullable('a)
  where RefTypes('a)
is Null
   non-null is ptr : boxed('a)

// Homogeneous lists:
(union (list 'a)
  nil
  (cons car:'a cdr:(list 'a)))

// Bignums
(union int:unboxed
  (fix f:(bitfield int32 31))
  (big b: boxed((bool, (vector word)))))
```

14 Foundational Type Classes

The standard prelude provides a number of standard type classes:

```
// Equality comparison by identity:
(trait (EqComparison 'a)
  eq : (fn ('a 'a) bool))

// Equality comparison by identity,
// with exceptional handling for
// numerics:
(trait (EqlComparison 'a)
  eql : (fn ('a 'a) bool))

// Generalized equality:
(trait (EqualityComparison 'a)
  == : (fn ('a 'a) bool)
  != : (fn ('a 'a) bool))

// Magnitude comparison
(forall ((EqualityComparison 'a))
  (trait Ord('a)
  <  : (fn ('a 'a) bool)
  <= : (fn ('a 'a) bool)))

// Checked arithmetic
(forall (Ord('a))
  (trait (Arith 'a)
    +: (fn ('a 'a) 'a)
    -: (fn ('a 'a) 'a)
    *: (fn ('a 'a) 'a)
    /: (fn ('a 'a) 'a)
    <<:(fn ('a word) 'a)
    >>:(fn ('a word) 'a)))

// Ring arithmetic
(forall (Ord('a))
  (trait (Ring 'a)
    R+: (fn ('a 'a) 'a)
    R-: (fn ('a 'a) 'a)
    R*: (fn ('a 'a) 'a)
    R/: (fn ('a 'a) 'a)
    R<<:(fn ('a word) 'a)
    R>>:(fn ('a word) 'a)))

// Sign transformations
(forall (Ord('a))
  (trait Signed('a)
    negate: (fn ('a) 'a)
    abs:    (fn ('a) 'a))
```

III Formal Specification

15 Grammar

The section below gives the extended EBNF grammar for the BitC language, including derived forms. Non-terminals are shown in italics. Tokens are shown in regular face. The characters "{", "}", and "|", are quoted when appearing as tokens. When appearing as a superscript, the character "*" indicates "zero or more" occurrences, the character "+" indicates "one or more" occurrences, and the character "?" indicates "zero or one occurrences." These should be read as metasyntactic only when appearing in a superscript. Note that parenthesis are *not* metasyntactic in extended Backus-Nauer form, and should be read as single-character tokens.

Within the EBNF productions below, the left and right parenthesis, period, colon, commma, and single quote characters should always be read as single character tokens. Spaces around these tokens have been omitted for the benefit of typeset readability.

15.1 Categorical Terminals

The following categorical terminals are defined by the regular expressions given in the respective sections:

Id Identifiers (Section 2.2)

IntLit Integer literals (Section 2.5.1)

FloatLit Floating point literals (Section 2.5.2)

CharLit Character literals (Section 2.5.3)

StringLit String literals (Section 2.5.4)

15.2 Interfaces, Units of Compilation

```
start ::= version? interface
  | version? module+
  | version? implicit_module
ifname ::= {Id.}* Id
interface ::=
  interface ifname docstring? { def+) }
module ::=
  module ifname? docstring? { mod_def+ }
mod_def ::= def | provide
implicit_module ::= mod_def+
import ::= import ifname as Id
provide ::= provide Id ifname Id+
usedecl ::=
    (use {Id.Id | (Id.Id as Id)}+)
def ::= import
  | usedecl
  | typedef
  | typedecl
  | tcdef
  | instdef
  | valdef
  | proclaim
  | declare
```

15.3 Type Declaration and Definition

The `union` and `struct` forms are semantically derivable from `repr` (or vice versa),[11] but for purposes of specifying typing it is more convenient to retain them and use the conventional typing definitions for product and union types.

```
constraint ::= typapp | ident
typnm ::= ident
  | (ident tvar+)
  | (forall (constraint+) ident)
  | (forall (constraint+)
                   (ident tvar+))
val ::= :unboxed | :boxed | :opaque
typedef ::=
    (struct typnm val
                docstring?
                declare+ {field|fill}+)
  | (union typnm val
                docstring?
                declare+ {field|fill}+)
  | (repr typnm val
              docstring? (reprbody))
  | (exception ident
                  docstring? field*)
field ::= Id : type
  | (the type Id)
fill ::=
    (fill (bitfield fixpttype IntLit))
reprbody ::= (tag Id+)
  | field
  | fill
  | (case {(tags (reprbody))}+)
tags ::= Id | (Id+)
typedecl ::=
    (struct typnm val
                docstring?
                {external Id?}?)
  | (union typnm val
                docstring?
               {external Id?}?)
  | (repr typnm val
              docstring?
              {external Id?}?)
tcdef ::=
    (trait typnm
                  docstring?
                  {(tyfn (tvar+) tvar)}*
                  {ident:fntype}*)
instdef ::=
    (instance qual_constraint
                 docstring? expr+)
qual_constraint ::= constraint
  | (forall (constraint+) constraint)
```

15.4 Value Declaration and Definition

```
valdef ::=
    (def defpattern docstring? expr)
  | (def (ident bindingpattern?)
       docstring?  expr+)
defpattern ::= ident
  | ident:qualtype
  | (the qualtype ident)
  | ()
  | (pair defpattern defpattern)
  | ({defpattern,}+ defpattern)
bindingpattern ::= ident
  | ident:type
  | (the type ident)
  | ()
  | (pair defpattern defpattern)
  | ({defpattern,}+ defpattern)
proclaim ::=
    (proclaim ident:qualtype
              {external Id?}?)
```

[11] This statement of semantic derivability ignores the Cardelli family of representation optimizations that are not currently expressable for `repr`, but it is intended to fully support control of these optimizations in future enhancements to the language.

```
     ; Note: external Id may include BitC
     ; reserved words.
qualtype ::= type
   | (forall (constraint+) type)
   | constraint
   | (forall (constraint+) constraint)
```

15.5 Types

Note that the pair type is semantically a derived form. It appears in the grammar solely because of the need to support pattern bindings and multiple return values.

```
tvar ::= 'Id
inttype ::= int8 | int16 | int32 | int64
   | uint8 | uint16 | uint32 | uint64
pairtype ::= (pair type type)
   | ({type,}+ type)
type ::= ident
   | tvar
   | () | bool | char | string | exception
   | inttype
   | float | double | quad
     // integer bitfield:
   | inttype(IntLit)
     // boolean bitfield:
   | bool(1)
   | (boxed type)
   | unboxed( type)
   | mutable(type)
   | (fn (type*) type)
   | pairtype
   | (array type IntLit)
   | (vector type)
   | (ident type+)
```

15.6 Expressions

```
ident ::= Id | Id.Id
expr ::= eform
   | (the type eform)
// eform permits ident via expr.id
eform ::= Id
   | ()
   | eform.Id
   | (the type eform).Id
   | (pair expr expr)
   | (member expr Id)
   | expr [ expr ]
   | expr ^
   | (deref expr)
   | (suspend ident expr)
   | ({expr,}+ expr)
   | (array expr+)
   | (vector expr+)
   | (MakeVector expr expr)
   | (begin expr+)
   | (lambda (bindingpattern*) expr+)
   | (expr expr*)
   | (if expr expr expr)
   | (and expr+)
   | (or expr+)
   | (set! expr expr)
   | (dup expr)
   | (cond ( {(expr expr)}*)
           (otherwise expr))
 // MAY NEED CASE
   | (switch Id expr
           ( {(switchtags expr_seq)}*
             (otherwise expr_seq)))
   | (try expr
      (catch Id
           {(switchtags expr)}*
           (otherwise expr)?))
   | (throw expr)
   | (let ({(bindingpattern expr)}+)
       expr)
   | (letrec ({(bindingpattern expr)}+)
       expr)
   | (do ({(bindingpattern expr expr)}+)
       (expr expr)
       expr)
   | () | false | true | CharLit | StringLit
   | IntLit | FloatLit
switchtags ::= ident | (ident+)
```

15.7 Miscellaneous

```
declare ::=
    (declare {(ident type) | ident}+)
docstring := StringLit
```

IV Standard Library

16 BitC Standard Library

This section needs badly to be completely revisited.

The BitC standard library is described as a set of groups. Each group gives a built-in function, a list of signatures supported by that built-in function, and a description of the operation of the function.

16.1 Arithmetic

BitC defines the built-in operators +, -, *, /, and %, with the usual meanings of two's complement addition, subtraction, multiplication, division and remainder for signed types, and one's complement addition, subtraction, multiplication, division, and remainder for unsigned types.

BitC also defines the build-in operators `bit-or`, `bit-xor`, and `bit-and`, with the usual meanings of one's complement bit manipulation.

These operators are defined over the following signatures:

int8 × int8 → int8
int16 × int16 → int16
int32 × int32 → int32
int64 × int64 → int64
uint8 × uint8 → uint8
uint16 × uint16 → uint16
uint32 × uint32 → uint32
uint64 × uint64 → uint64

Unary minus is also supported over all integral types with the usual meaning.

16.2 Comparison

BitC defines the built-in comparison operators <, <= > >= =, and != with the usual meanings of less than, less than or equal, greater than, greater than or equal, equal, and not equal.

These operations are defined over the following signatures:

char × char → bool
int8 × int8 → bool
int16 × int16 → bool
int32 × int32 → bool
int64 × int64 → bool
uint8 × uint8 → bool
uint16 × uint16 → bool
uint32 × uint32 → bool
uint64 × uint64 → bool

The = and != operators are additionally defined over pointers of like type. They perform structural equality (eq) and inequality.

17 Verification Support

In addition to its role as a means of expressing computation, BitC directly supports the expression of constraints on execution, and the expression of proof obligations concerning the results of computations. While the bulk of verification effort is performed in the BitC Prover, theorems and invariants also introduce requirements for compile-time static checking.

Note that the phrase "all possible variable instantiations" is restricted to *legal* instantions as determined by the type checker. BitC is statically typed, and BitC functions and theorems are therefore defined only over their stated domains.

17.1 Axioms

The `defaxiom` form introduces a term rewrite that is accepted as true by the BitC prover. The body of the axiom is a boolean expression that must always return `true` for all possible variable instantiations:

```
(defaxiom name truth-expr)
```

17.2 Proof Obligations: Theorems

The `defthm` form introduces a proof obligation that must be discharged by the BitC Prover. The body of a theorem is a boolean expression that is considered to be discharged if its result is `true` for all possible variable instantiations:

```
(defthm name truth-expr)
```

17.3 Proof Obligations: Invariants and Suspensions

The `definvariant` form introduces a proof obligation that must be discharged by the BitC Prover at all sequence points where it is not explicitly suspended. The body of an invariant is a boolean expression that is considered to be discharged if its result is `true` for all possible variable instantiations:

```
(definvariant name truth-expr)
```

An invariant may be temporarily suspended by the `suspend` form:

```
(suspend name e)
```

The logical effect of `suspend` is to advise the prover that the invariant given by *`name`* is not expected to hold within the scope of the `suspend` form.

For program semantics purposes, `suspend` is a derived form:

```
(suspend name e) =>
(begin e)
```

17.4 Theories

The `deftheory` form gathers a number of theorems into a single group for purposes of suspension:

```
(deftheory name thm1 ... thmn)
```

where each thm_i has been previously introduced by `defthm`.

17.5 Suspending and Enabling

For purposes of proof search management, theorems and theories may be disabled or enabled by the `disable` and `enable` forms:

```
(disable name1 ... namen)
(enable name1 ... namen)
```

where each $name_i$ has been previously introduced by `defthm` or `deftheory`.

The effect of disablement is to render a theorem or group of theorems inactive for purposes of proof search. Disabling or enabling remains in force until altered by a subsequent enable or disable or until the end of the containing lexical scope.

18 Acknowledgments

We owe a significant debt to the help of Scott Smith of Johns Hopkins University. Scott's input has influenced our thinking about the BitC/L subset language. While BitC/L is not yet visible in the specification, some of the design decisions made here reflect constraints derived from BitC/L.

Paritosh Shroff, also at Hopkins, spent a great deal of time helping us explore the implications, strengths, and weaknesses of the `typecase` construct that survived to version 0.8 of the specification and the "match type" notion that was needed to support it. Beginning in version 0.9, we abandoned match types in favor of type classes. This decision was greatly assisted by the input of Mark Jones of the Oregon Graduate Institute.

References

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

[2] —: *IEEE Standard for Binary Floating-Point Arithmetic*, 1985, ANSI/IEEE Standard 754-1985.

[3] —: *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, 1987, ANSI/IEEE Standard 854-1987.

[4] Jacques Garrigue. "Relaxing the Value Restriction." *Proc. International Symposium on Functional and Logic Programming*. 2004.

[5] Anita K. Jones. *Protection in Programmed Systems*, Doctoral Dissertation, Department of Computer Science, Carnegie-Mellon University, June 1973.

[6] 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.

[7] NOT USED M. Kaufmann, J. S. Moore. *Computer Aided Reasoning: An Approach*, Kluwer Academic Publishers, 2000.

[8] 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.

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

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

[11] Iavor S. Diatchki, Mark P. Jones, and Rebekah Leslie. "High- level Views on Low-level Representations." *Proc. 10th ACM Conference on Functional Programming* pp. 168–179. September 2005.

[12] J. S. Shapiro, 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.

[13] Unicode Consortium. The Unicode Standard, version 4.1.0, defined by *The Unicode Standard Version 4.0*, Addison Wesley, 2003, ISBN 0-321-18578-1, as amended by *Unicode 4.0.1* and by *Unicode 4.1.0*. `http://www.unicode.org`.

[14] N. Wirth and K. Jensen. *Pascal: User Manual and Report*, 3rd Edition, Springer-Verlag, 1988