BitC Language Specification[†]

Version 0.9+

Jonathan Shapiro, Ph.D. Swaroop Sridhar Scott Doerrie Mark Miller Eric Northup
Systems Research Laboratory
Dept. of Computer Science
Johns Hopkins University

February 17, 2006

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.

While the current language specification uses a Scheme-like concrete syntax, this choice is a matter of convenience only. It is entirely possible to build a C-like concrete syntax for BitC, and at some point it may become compelling to do so.

Contents

1 Overview

The BitC project is part of the successor work to the EROS system [11]. 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, Scheme, and C. Like Standard ML [10], BitC has a formal semantics, static typing, a type inference mechanism, and type variables. Like Scheme [8], BitC uses a surface syntax that is readily represented as BitC data. Like C [1], BitC provides full control over data structure representation, which is necessary for high-performance systems programming. 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.

In contrast to ML, BitC syntax is designed to discourage currying. Currying encourages the formation of closures that capture non-global state. This requires dynamic storage allocation to instantiate these closures at runtime. Since there are applications of BitC in which dynamic allocation is prohibited, currying is an inappropriate idiom for this 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 `define`, 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.

Building on the features of ACL2 [7], BitC incorporates explicit support for stating theorems and invariants about the program as part of the program's text.

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 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 "...". Such a sequence must have at least one element. For example:

```
(begin e ... e)
```

indicates that the `begin` form takes a (non-empty) sequence of expressions. When it is intended that zero elements should be permitted in a sequence, the example will be written:

```
(begin [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.

1.3 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
```

by similarly qualifying a formal parameter, as in:

```
(define (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.

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.

2.1 Comments

A comment is introduced by a semicolon and 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). This implies that the comment syntax cannot be successfully exploited for identifier splicing as in early C preprocessors.

Notwithstanding the preceding, a semicolon appearing within a string or character literal does not begin a comment.

2.2 Identifiers

BitC identifiers are case sensitive. An identifier may start with any "identifier character" (UNICODE 4.1.0 character class XID_Start), followed by any number of optional "identifier continue characters" (UNICODE 4.1.0 character class XID_Continue). The following "extended alphabetic characters" may also appear in *any* position of an identifier.

```
! $ % & * + - / \ < = > ? @ _ ~
```

An identifier may be optionally qualified by an interface binding name followed by a period ("."). Thus, the most general form of an identifier is:

[*Identifier*.]*Identifier*

Identifiers beginning with two leading underscores are reserved for use by the runtime system.

2.3 Reserved Words

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

`#f`	`#t`	`and`
`apply`	`array`	`array-length`
`array-nth`	`begin`	`bitc-version`
`bitfield`	`bool`	`case`
`catch`	`char`	`cond`
`declare`	`defaxiom`	`defexception`
`define`	`definvariant`	`defstruct`
`deftheory`	`defthm`	`defunion`
`deref`	`disable`	`do`
`double`	`dup`	`enable`
`exception`	`external`	`fixint`
`float`	`if`	`import`
`import!`	`int8`	`int16`
`int32`	`int64`	`interface`
`fn`	`lambda`	`let`
`letrec`	`member`	`mutable`
`opaque`	`or`	`otherwise`
`pair`	`proclaim`	`provide`
`provide!`	`quad`	`ref`
`set!`	`string`	`super`
`suspend`	`the`	`throw`
`try`	`tyfn`	`uint8`
`uint16`	`uint32`	`uint64`
`use`	`val`	`vector`
`vector-length`	`vector-nth`	`word`

The following identifiers are reserved for use as future keywords:

`assert`	`case!`	`check`
`coindset`	`constrain`	`defequiv`
`defobject`	`defrefine`	`deftype`
`defvariant`	`do*`	`indset`
`int`	`label`	`length`
`let*`	`list`	`literal`
`namespace`	`nth`	`read-only`
`sensory`	`sizeof`	`tycon`
`typecase`	`using`	

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.

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.

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.

2.4 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.

2.4.1 Integer Literals

The general form of an integer literal is:

```
[-][baser]digits
```

where *base* is radix of the subsequent digits expressed in decimal form. Legal bases are 2, 8, 10, and 16. In the absence of a base prefix, the digits are interpreted as base ten. The *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]

2.4.2 Floating Point Literals

The general form of a floating point literal is:

```
[-][baser]digits.digits[^exponent]
```

where *base* defines the decimal encoded radix of the digits and *exponent* is an integer literal, possibly including an initial minus sign and a radix specification for the exponent part. Digits are selected as for integer literals, above. Note that the decimal point is not optional, and must have digits on both sides. Thus, `0.0` is a valid floating point literal, but `0.` and `.0` are not.

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]

[1] There is an issue here: doesn't the initial set need to be the set of all integer fi eld 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 defi ned over homogeneous argument types. Swaroop points out that expanding the set isn't what creates the problem for type inference.

[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 fbating point literal conversion, but I don't know enough about fbating point conventions to know what that statement should be.

2.4.3 Character Literals

BitC uses the UNICODE character set as defined in version 4.1.0 of the UNICODE standard. 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 previously listed UNICODE categories, the following characters are considered printable characters as well: Notwithstanding the listed UNICODE categories, the characters ’{’ and ’}’ (left and right curly brace) are excluded for use in character literal escaping.

```
! " # $ % & ’ ( ) * + , - . /
: ; < = > ? @ [ \ ] ˆ _ ‘ | ˜
```

A character may also be specified by its unicode code point:

```
#\{U+digits}
```

Where *digits* are hexidecimal.

Deprecated Syntax

The radix syntax for character literals is transitional, and will be dropped in the version 0.10 specification.

A character may also be specified by its unicode code point in binary, decimal, octal, or hexidecimal:

```
#\{[baser]digits}
```

Where *base* is one of 2, 8, 10, or 16 and *digits* are as for integer literals above. In the absence of a specified base, the digits are assumed to be decimal.

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

```
#\space
#\linefeed
#\return
#\tab
#\backspace
#\lbrace
#\rbrace
```

Deprecated Syntax

The following literal forms using curly-braces are transitional, and will be dropped in the version 0.11 specification.

The non-printing character literal forms may optionally be wrapped in curly braces:

```
#\{space}
#\{linefeed}
#\{return}
#\{tab}
#\{backspace}
#\{lbrace}
#\{rbrace}
```

2.4.4 String Literals

BitC strings are written within double quotes, and may contain the previously listed “printable characters” *excluding* backslash (“/”). They may also contain spaces (U+0020), left curly brace (U+007B) and right curly brace (U+007D).

Within a string, the backslash character (“\”) is interpreted as beginning an encoding of a specially embedded character. The character following the “\” is either a single-character embedding or a curly brace character “{” identifying the start of a UNICODE character embedding. The legal forms and their meanings are:

n	Linefeed
r	Carriage Return
t	Horizontal Tab
b	Backspace
s	Space
f	Formfeed
”	Double quote
\\\\	Backslash
\\{**U+*digits***}	Unicode code point, hexidecimal *digits*.

Deprecated Syntax

The following embedded character forms using curly-braces are transitional, and will be dropped in the version 0.11 specification.

The following alternate backslash embeddings are permitted:

\{**linefeed**}	Linefeed
\{**return**}	Carriage Return
\{**tab**}	Horizontal Tab
\{**backspace**}	Backspace
\{**space**}	Space
\{**lbrace**}	Left curly brace
\{**rbrace**}	Right curly brace

2.5 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 begins (ignoring comments) with a `bitc-version` form. The syntax of the `bitc-version` form is:

```
(bitc-version s)
```

where *s* is the version string of the BitC version to which this program conforms. For the version of BitC described in this document, the proper version string is `"0.9+"`.

In the case of an interface compilation unit, the `bitc-version` form is followed by exactly one `interface` form. In source compilation units, the `bitc-version` form is followed by an arbitrary sequence of imports, definitions, declarations, and use forms that are *not* `interface` forms.

2.5.1 Definitions and Declarations

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

```
define       definstance  defstruct
deftypeclass defunion     proclaim
use
```

The `define`, `defunion`, and `defstruct` forms support simple recursion. That is, the identifier(s) being defined may be used in their definition. However, the 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 in interfaces 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.

The `use` form is used to provide an alternative identifier that is equivalent in all respects to some existing top-level identifier.

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

3 Types

BitC provides explicit control over data structure representation while preserving a memory-safe and type-safe language design.

3.1 Categories of Types

BitC has two categories of types: value types and reference types.

A value type is one whose value representation is "embedded" in the representation of its containing composite type or stack frame. The lifetime of an instance of value type is determined by the lifetime of its container, and it is the responsibility of the container to allocate storage for its contained value types.

A reference type is a type whose value representation resides in the heap. Every instance of a reference type has at least one reference value that denotes it. The *reference* is a value type; the value *denoted* by the reference is a reference type.

If *T* is a value type, then `(ref T)` is the type of a reference denoting a heap-allocated instance of *T*. Similarly, if *T* is a reference type, then `(val T)` is the corresponding value type. The `val` type constructor can only be applied to reference types whose target is of statically known size. Storage for a value of value type is allocated from its containing type.

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

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 `()`.[4]

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

bool A boolean value, either `#f` or `#t`. The representation of this type is a single byte, aligned at a arbitrary 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 An implementation-defined unsigned integral type sufficient to hold pointer values. Values of type `word` are aligned at a boundary that is a multiple of their size.

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

```
(bitfield 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. Type equivalence for `bitfield` types is determined structurally: two appearances of a `bitfield` form having the same base type and size denote the same type.

The form

```
(bitfield int32 4)
```

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

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 iwth a 15 bit exponent and a 112 bit mantissa.

The type `bool` and its constructors `#t` and `#f` are defined in the standard prelude.

At present, the BitC runtime system assumes that actual character values will be limited to the UNICODE characters whose encoding falls between 0 and 127. This is a temporary restriction.

3.3 Type Variables

A type variable is an identifier preceded by a single quote, as in `'a`. Type variables may appear in any position where a type can appear. Type variables are most commonly used as type constructor arguments for named constructed types (see below). They can also be used to annotate expressions, for example to require 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 also unspecified (and should be inferred), but the program author is stating that the return type and the last argument type are the same. This type of annotation is sometimes useful to assist the inference engine.

The scope of a type variable is its outermost defining form.

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

```
(ref 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 Value Types

If *`T`* is a reference type, then

```
(val T)
```

is the corresponding value type. The `val` constructor can only be applied to reference type whose target has statically known size.

A complete type is one that has been completely defined. In particular, a named type is not completely defined when used within its defining form, as in a recursive type definition, and no type declared by `def` is considered complete until it's corresponding defining form is complete.

3.4.3 Function Types

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

```
(fn targ tresult)
```

is the type of a function taking a single argument of type t_{arg} and returning a value of type t_{result}.

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 determined by the underlying processor architecture.

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*.

3.6 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

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

```
(defstruct nm nm1:t1 ... nmn:tn)
(defstruct (nm tv1 ... tvn)
  nm1:t1 ... nmn:tn)
```

where nm_i are disjoint identifiers giving the names of the structure fields, and t_i 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.

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:

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

(defstruct (tree-of ’a):ref
  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 `defunion` 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:

```
(defunion nm C1 ... Cn)
(defunion (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 types.

An identifier bound to a union constructor 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 constructor 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:

```
(defunion contrived
  (c char) (i int32))

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

define (respectively) a reference type holding either a `char` or an `int32`, and a value type of optionally null pointer values.

The declaration:

```
(defunion (list 'a):ref
  nil
  (const 'a (list 'a)))
```

Defines the reference type of homogeneous lists.

Storage Layout Each variant of a union declaration effectively defines a *k* element structure, where *k* is the number of arguments to the value constructor. If the union representation requires an explicit tag (see below), storage for this tag appears at the lowest address of the value representation. Assuming that a tag is present, each leg of the union is arranged in memory as though it were a pair of anonymous fields of the form:

```
(pair tag-type
      element-type)
```

without regard to the layout of other legs. In the special 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:

```
(defunion (list 'a):ref
  (declare (tag-type uint8))
  nil
  (cons 'a (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.

A union having exactly one element of reference type and otherwise consisting solely of enumeration values *can* be represented without additional storage for the type tag. However, a type tag declaration will be honored if present.

3.6.3 Value vs. Reference Types

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

```
(defstruct nm:ref
  nm1[:t1] ... nmn[:tn])
(defstruct (nm tv1 ... tvn):ref
    nm_1[:t_1] ... nm_n[:t_n])
  (defunion nm:ref C_1 ... C_n)
  (defunion (nm tv_1 ... tv_n):ref
    C_1 ... C_n)
```

The qualifier ":ref" indicates that the type declared (and consequently the type returned by value constructors) is a reference type. The qualifier ":val" indicates that the type declared is a value type. 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.

3.6.4 Forward Declarations

The declarations

```
(defstruct nm [qual] [external])
(defunion nm [qual] [external])
(defstruct (nm tv_1 ... tv_n)
           [qual] [external])
(defunion (nm tv_1 ... tv_n)
           [qual] [external])
```

state (respectively) that `nm` is a structure (respectively union) reference type of the stated arity whose internal structure is not disclosed. If present, the qualifier *`qual`* optionally declares the type to be one of ":ref", ":val", or ":opaque". In the absence of qualification, the default is ":ref". 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:

```
(defstruct int :val external bitc_int)
```

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.

3.6.5 Named Type Conveniences

The following types are defined in the BitC standard prelude.

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

(defunion (list 'a) :ref
   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.

The `list` type is also supported by a right-associative convenience syntax:

```
[]      =>  nil
[a]     => (cons a nil)
[a,b]   => (cons a (cons b nil))
[a,b,c] =>
  (cons a (cons b (cons c nil)))
```

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

Note

The `list` form may become a syntactic form in future versions of this specification, because we are considering adding pattern matching support for it.

3.7 Mutability

Unless modified by the `mutable` keyword, the preceding types yield immutable instantiations. If *T* is a type, the type

```
(mutable T)
```

is the type of mutable instances of *T*. If the type *T* is a reference type, then the type `(mutable T)` describes a mutable reference to a memory location in the heap.

3.8 Method Types

The type:

```
(method (TC 'a ... 'z)
        (fn T_arg T_out))
```

describes the type of *methods* (see type classes, below) that are members of some type class *TC*. The types of the arguments and return values may refer to the type variable paramaters of the type class. Multiple type class requirements can be expressed using `and`:

```
(method (and (TC1 'a ... 'z)
             (TC2 'a ... 'z))
        (fn T_arg T_out))
```

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 `defexception`. The syntax of an exception declaration is:

```
(defexception nm [T_1 ... T_n])
```

where each T_i 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 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:

```
(define (foo f)
  (pair
    (f (cons 1 (cons 2 nil)))
    (f (cons #t (cons #f nil)))))
```

4 Type Classes

A **type class** defines an n-ary relation on types. Every type class is parameterized over $n \geq 1$ types, and defines a set of methods over those types. Type classes provide support for *ad hoc* polymorphism. 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.

4.1 Definition of Type Classes

A type class is defined by the abstract syntax:

```
(deftypeclass (nm tv ... tv)
  [typeclass-declarations]
  method-definitions)
```

where a typeclass declaration may be either a statement of functional dependency between types [6] or a declaration that some other type relation is a superset of this one:

```
(tyfn (tv ... tv) tv)
(super (typeclass tv ... tv) ...
       (typeclass tv ... tv))
```

and each method definition takes the form:

```
nm : function-type
```

Each method defined by a type class is introduced into the scope containing the type class definition.

Restriction: Superclass relationships must be acyclic.

4.1.1 Example: `Eq`

The canonical first example of type classes is the type class `Eq` of equality types:

```
(deftypeclass (Eq '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 `!=`.

What should this be called? Eql? Equal?

4.1.2 Example: `Ord`

The next example is the type class `Ord` of orderable types:

```
(deftypeclass (Ord 'a)
  (super (Eq 'a))
  < : (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 <, provided these types are also members of the relation defined by `Eq`.

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.

4.1.3 Example: `Convert`

Value promotion and demotion is provided by the Conversions type class, which defines the `convert` method: over a variety of input types, following the conventions of the C programming language:

```
convert : (fn (int8)  int16)
convert : (fn (int8)  int32)
convert : (fn (int8)  int64)

convert : (fn (int16) int8)
convert : (fn (int16) int32)
convert : (fn (int16) int64)
...
```

this is accomplished by defining the method `convert` as a member of the multi parameter type class `Conversions` (elided here):

```
(deftypeclass (Conversions 'a 'b)
  convert: (fn ('a) 'c)
  ...)
```

What this says is that the operator `convert` can be applied to any two types `'a` and `'b`.

4.1.4 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 consulting the instantiations that are currently in scope.

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 declared by the `definstance` form. The declaration:

```
(definstance (Ord int32)
  int32.<)
```

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`.

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:

```
(definstance (Eq int32)
  int32.==
  int32.!=)
(definstance (Ord int32)
  int32.<)
```

5 Binding of Values

5.1 Binding Patterns

Binding patterns are used to bind names to values, and to decompose values into their constituent parts.

The pattern

```
()
```

matches any element of unit type. If functions remain multi-argument, it is not clear that this is still a useful binding pattern.

If *id* is an identifier, the pattern:

```
id
```

matches the result of any expression.

If *id* is an identifier and *T* is a type, then the pattern

```
(the T id)
```

matches any expression whose result type is *T*. As a convenience shorthand, this binding pattern may also be written:

```
id : T
```

The pattern

```
(pair bp1 bp2)
```

matches (via positional correspondence) any instance of type `(pair 'a 'b)`, subject to the requirement that the respective pair elements in turn match the respective binding patterns bp_1 and bp_2.

The right-associative pattern $(bp_1, bp_2, \ldots, bp_{n-1}, bp_n)$ is a convenience syntax for:

```
(pair bp1 (pair bp2 ...
             (pair bpn-1, bpn)))
```

A given identifier may appear free exactly once within a binding pattern.

5.2 define

Value bindings are introduced by `define`:

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

where each *bp* is a binding pattern. The second form is a convenience shorthand for:

```
(define id
  (lambda ([bp1 ... bpn])
    e ... e))
```

where each bp_i is a binding pattern.

The right hand form of a `define` is evaluated to obtain a value, which is then pattern matched against the pattern being bound. For each identifier `x` that appears in the binding pattern, then identifier is bound in the currently active environment to the positionally corresponding element within the value resulting from the evaluation of the expression.

Identifiers defined within a single `define` are deemed "incomplete" until the end of the enclosing `define` form.

Mutually recursive procedure definitions can be achieved by defining a pair of lambdas:

```
(define (pair odd even)
  (pair
    (lambda (x) ; odd
      (cond ((= x 0) #f)
            ((< x 0) (odd (- x)))
            (otherwise
              (not (even (- x 1))))))
    (lambda (x) ; even
      (cond ((= x 0) #t)
            ((< x 0) (even (- x)))
            (otherwise
              (not (odd (- x 1))))))))
```

Restrictions In the definition

```
(define bp e)
```

if *x* is an identifier that appears in the binding pattern (and is therefore incomplete), then either

- the identifier *x* does not appear free in the expression *e*, *or*
- The expression *e* is either a `lambda` form or it is a tree of pattern matchable type instance constructors whose leaf arguments are lambda forms.

This restriction intentionally prevents infinitely recursive data constant definitions.

6 Declarations

The `proclaim` form is used in interfaces 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.

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.

The `proclaim` form may only be used within an interface unit of compilation.

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 Type-Qualified Expressions

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

```
(the T e)
```

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.

7.4 Value Constructors

7.4.1 unit

The expression:

```
()
```

denotes the singleton unit value.

7.4.2 make-vector

The expression:

(`make-vector` e_{len} e_{value})

creates a new vector whose length is determined by the value of the expression e_{len}, which must evaluate to an integer-valued expression greater than zero. Each element of the vector is initialized to *a copy of* the value of e_{value}.

Note that `make-vector` is not widely useful unless the expression e_2 is mutable. The vector created above will contain immutable copies of the value `42:uint32` in all positions. The expression

```
(make-vector 4 42:(mutable uint32))
```

in contrast, creates a vector of length 4, each cell of which contains a mutable pointer. These mutable locations initially contain references to the value `42:uint32`, but the value stored at each location can be modified.

7.4.3 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.

7.4.4 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))

[]    => nil
[a] => (cons a nil)
[a,b] => (cons a (cons b nil))
```

7.5 Expression Sequences

Derived form

The expression:

```
(begin e_1 ... e_n)
```

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.

Derivation The canonical rewriting of the `begin` form using core language constructs is:

```
(begin e1 ... e2) =>
((lambda () e1 ... e2))
```

7.6 Iteration

Derived form

BitC provides the looping construct `do` as a derived form.

```
(do ((bp_1 e_init-1 e_step-1)
     ...
     (bp_n e_init-n e_step-n))
    (e_test e_result)
   e_body-1
   ...
   e_body-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 `do` 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 `do`-bound variables. After all of the initialization values are computed in order, the `do`-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 **#t**, 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 `do`-bound variables are bound to the newly computed results in parallel and a new pass is initiated over the loop body as previously described.

Derivation The canonical rewriting of the `do` form using core language constructs is:

```
(do ((bp_1 e_init-1 e_step-1)
     ...
     (bp_n e_init-n e_step-n))
    (e_test e_result)
   e_body-1
   ...
   e_body-n) =>
(letrec
  ((__loop
    (lambda (bp_1 ... bp_n)
      (if e_test e_result
          (begin
            e_body-1
            ...
            e_body-n
            (__loop e_step-1
                    ...
                    e_step-n))))))
  (__loop e_init-1 .. e_init-n))
```

Note the because `do` is defined in terms of `lambda` application, value restrictions on argument bindings apply.

Pragmatics: The `do` form is guaranteed to be processed tail recursively, as implied by the canonical rewriting.

7.7 Structure Field Reference

If *e* is an expression of structure type, and *field* is an identifier naming some field in that structure type then:

```
(member e field)
```

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

The expression:

```
e.field
```

is a convenience shorthand for

```
(member e field)
```

7.8 Array and Vector Expressions

7.8.1 array-length, vector-length

If *e* is an expression of array (respectively: vector) type, then

```
(array-length e)
(vector-length e)
```

returns a `word` whose value is the number of elements in the array.

7.8.2 array-nth, vector-nth

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

```
(array-nth e ei)
(vector-nth e ei)
```

are (respectively) expressions that return the e_i'th element of the array (respectively: vector). If the value e_i is greater than or equal to the length of the array (respectively: vector), then a `IndexBoundsError` exception is thrown.

`array-nth` and `vector-nth` are syntactic forms. The returned value is an encapsulated reference that can be used as an argument to `set!`.

7.9 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 ([bp1 ... bpn]
  e1 ... en)
```

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.

7.10 Function Application

The expression:

```
(efn [e1 ... en])
```

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.

7.11 Conditional Execution

7.11.1 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 values returned by the e_{then} and e_{else} forms must be compatible with the type expected by the outer expression within which the `if` expression is nested.[5]

Derivation The canonical rewriting of `if` is:

```
(if e1 e2 e3) =>
(case e1
  (#t e2)
  (#f e3))
```

7.11.2 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 `#f`, 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 #f)
```

7.11.3 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 `#t`, 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 #t e2)
```

[5] If the value of an `if` form is not captured, then there is no real need to require that its e_{then} and e_{else} parts return values of the same type. In this case, the following could be permitted:

```
(begin (if e_test 32 false) e1)
```

because the return value is not captured. Only in situations where the return value of the `if` *is* captured is it really required for the e_{then} and e_{else} expressions must return values that are compatible with the receiver.

I'm inclined not to try to take any advantage of this for now. One can always solve the problem of compatibility by writing:

```
(begin (if e_test (begin 32 ()) (begin false
())) e1)
```

7.11.4 cond

Derived form

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

```
(cond (e_test1 e_1)
      (e_test2 e_2)
      ; ...
      (otherwise e_n))
```

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.[6]

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

```
(cond (e_test1 e_1)
      (e_test2 e_2)
      ; ...
      (otherwise e_n)) =>
(if e_test1
    e_1
    (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)
```

7.12 Mutability

The expression:

```
(set! e_1 e_2)
```

is used to set the value of a mutable entity. The expression e_1 should evaluate to an entity v_1 of mutable type. The above expression sets v_1 to the value obtained by evaluating the expression e_2. e_2 should evaluate to a value of compatible type.

7.13 References

7.13.1 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.

7.13.2 deref

If e is an expression of reference type, then:

```
(deref e)
```

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

The expression:

```
e^
```

is a convenience shorthand for

```
(deref e)
```

7.14 Value Matching

7.14.1 Value Patterns

Binding patterns may be used in any place where binding is performed, most notably in formal parameters. In value patterns, the following additional patterns are permitted:

- Every literal is a value matching pattern that matches an expression whose result value is equal to the literal value.

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

- If *uc* is a union value constructor taking *n* arguments, and vp_1 .. vp_n are value patterns, then

  ```
  (uc vp1 ... vpn)
  ```

  is a value pattern that matches any expression whose type is the type constructed by *uc* and whose elements recursively match the value patterns vp_1 through vp_n.

- The following are right-associative convenience syntax supporting the standard prelude `list` type:

  ```
  []    => nil
  [a] => (cons a nil)
  [a,b] => (cons a (cons b nil))
  ```

The distinction between binding patterns and value patterns arises because the arguments to procedures must not be partial types. Thus, given the preceding list example, it is illegal to specify a function parameter as in

```
(lambda ( (cons x) a )
  body)
```

because the pattern `(cons x)` does not exhaustively match the elements of any type.

Restriction: A value pattern may not be used to rebind an identifier that is bound to a value constructor in the current environment. This restriction exists to allow value patterns to match unary constructors. Consider the expression:

```
(define (f x)
  (case (x)
        (nil ...) ...))
```

It is ambiguous whether `nil` is a union constructor pattern matching one kind of list or an identifier to be rebound. The resolution is that it will be treated as a unary constructor pattern.

7.14.2 case

The `case` form provides sequential testing by value pattern matching. The expression:

```
(case e (vp1 e1)
        (vp2 e2)
        ; ...
        [(otherwise en)])
```

value matches *a copy of* the return value of *e* in sequence against each of the value patterns vp_1, ... vp_n. For the first pattern vp_i that matches, the corresponding expression e_i is evaluated in an environment where each free identifier appearing in the pattern p_i has been *unified* with the corresponding element of the value returned by *e*. The result of this evaluation of e_i is returned as the result of the `case` expression. All of the expressions e_i must be of compatible result types.

The pattern match performed by the patterns of a `case` expression are required to be exhaustive. The `otherwise` clause can be used to accomplish this, but may not appear if the pattern match is exhaustive.

Note that patterns are bound to a *copy* of the argument expression. This is significant when the argument is a mutable expression. In particular, the return value of:

```
(defstruct s  a:(mutable int32))
(let ((v (s 1)))
  (case v
    ((s a) (set! a 2)))
  v.a)
```

is 1, not 2.

Note that this cannot be expressed as lambda application, because value patterns are not permitted as lambda parameter patterns. Note that *e* is evaluated exactly once.

7.15 Exception Handling

7.15.1 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 (vp1 e1)
         ...
         (vp2 e2)
         [(otherwise en)]))
```

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

```
(tmp:exception (throw tmp))
```

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)
  (case e
    (vp1 e1)
    (vp2 e2)
    ...
    (otherwise en)))
```

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.

7.15.2 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`.

7.16 Local Binding Forms

7.16.1 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.

7.16.2 letrec

The `letrec` form provides a mechanism for locally binding identifiers to an expression value. 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 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) #f)
             ((< x 0) (odd (- x)))
             (otherwise
               (not
                 (even (- x 1)))))))
   (even
     (lambda (x) ; even
       (cond ((= x 0) #t)
                ((< x 0) (even (- x)))
                (otherwise
                  (not
                    (odd (- x 1))))))))
     body)
```

The value of a `let` 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 `define`. This ensures that cyclical constant data cannot be introduced.[7]

Any binding pattern appearing in the bp_i position in a `letrec` must be statically decomposable at compile time. It is not sufficient that the corresponding e_i be of compatible type. This restriction allows the binding patterns to be flattened away by the compiler without internally violating the completeness restriction.[8]

8 Interfaces

An interface describes a public set of definitions and declarations. From the client perspective, it describes the identifiers that are published by some providing body of code. From the provider perspective, it describes a collection of identifiers that are to be exported from the providing implementation.

An interface is the *only* means by which an identifier defined in one source compilation unit can be used in another. Unless declared as part of an interface, identifiers declared or defined in a source unit of compilation are *local* to that source unit of compilation.

8.1 Defining 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. For example, the interface:

```
(interface sample
  (define x 1) ; constant definition
  (defunion (list 'a):ref
    nil
    (cons 'a (list 'a)))
  (defstruct (tree-of 'a):ref)
  (proclaim y : int32))
  (defstruct S :opaque (int32 i))
```

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

[8] The alternative was to disallow binding patterns in `letrec` forms. The static decomposition constraint preserves greater syntactic consistency with `let`.

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.

The name of an interface is an identifier that is restricted to the upper and lower alphabetic characters, numbers, underscore, and hyphen.[9]

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

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

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

8.2 Importing an Interface, Aliasing

In order to use the identifiers supplied by this interface, the client unit of compilation must import the interface using a top-level `import` form:

```
(import local-name interface-name)
```

where *`interface-name`* is an interface name from the namespace of interface names and *`local-name`* is an identifier to be bound in the current scope that should be used to designate the elements of this interface. A *`use`* form may only appear at tope level. Following this import and `use`, if the imported interface defines a type `my-tree`, this type can be referenced as:

```
(define (f local-name:my-tree)
  ...)
```

[9] This restriction exists to ensure that there can be a straightforward mapping from identifi er names to fi le names in current fi le systems.

The syntax of `use` is:

```
(use use-form ... use-form)
```

where *`use-form`* is either a pair consisting of a name to be bound and a qualified name or a singleton qualified name. For example, given the interface:

```
(interface sample2
  (proclaim i : int32))
  (proclaim j : int32))
```

and an import statement

```
(import ifnm sample2)
```

an importing compilation can avoid the need to write "`ifnm.`" pervasively by writing:

```
(import ifnm sample2)
(use (x ifnm.i) ifnm.j)
```

The effect of this is that `ifnm.i` is aliased by the local name `x` and `ifnm.j` is aliased by the local name `j`. A *`use-form`* of the form *`prefix.ident`* is interpreted exactly as if `(`*`ident prefix.ident`*`)` had been written.

Like the defining and declaring forms, the `use` form introduces a name that may not subsequently be rebound in the outermost scope. In *contrast* to the defining and declaring forms, the appearance of an `use` declaration within an interface does *not* introduce an exported name.

8.2.1 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.

8.2.2 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.

8.3 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 local-name interface-name)
```

For example:

```
(bitc-version "0.9+")
(provide ln sample)

(defstruct (ln.tree-of ’a):ref
  left : (optional
           (ln.tree-of ’a))
  right : (optional
            (ln.tree-of ’a))
  height
  value : ’a)
```

Note that the `provide` form imports the corresponding interface. Note further that because the interface introduces the defining name for `tree-of`, the following alternative definition is equivalent in all respects to the one above:

```
(bitc-version "0.9+")
(provide ln sample)
(use (tree-of ln.tree-of))

(defstruct (tree-of ’a):ref
  left : (optional (tree-of ’a))
  right : (optional (ree-of ’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.

To resolve this, we introdue an inductive rule: if `id` is an *value* declared in an interface and defined in a source unit of compilation, then *all values whose declaration lexically precedes* `id` *in the interface* are deemed complete following the completing definition of `id`. Because complete identifiers cannot be redefined in the same scope, this implies that within any given source unit of compilation, the order of definition for values in a given interface `I` must match the order of appearance of their declarations in the interface. *This rule is probably not quite right. The essential point is that we need a rule that induces a strict lattice ordering on value definitions.*

8.4 Stateful and Stateless Interfaces

In an effort to encourage the construction of more robust and better structured programs, BitC encourages a discipline of separation of concerns in which shared state must be established by introduction and instantiation.

8.4.1 Stateless Interfaces

In the absence of other declaration, the values declared and defined by an interface are required to be "deeply immutable." A value is deeply immutable if:

- It is an immutable primary type,
- It is an immutable reference to an deeply immutable value,
- It is a lambda form whose closure references only deeply immutable values and identifiers.

The effect of this restriction is to ensure that the implementations of two stateless interfaces A and B cannot have a hidden communication channel merely by importing in common a third interface `I` that publishes mutable storage. This restriction corresponds to the notion of "memoryless procedures" introduced by Anita Jones in her dissertation [5].

8.4.2 Stateful Interfaces

Since BitC is a systems programming language, there exists a small number of programs and usages for which stateful interfaces are required. An interface can be declared as stateful by the inclusion within the `interface` form of the declaration:

```
(declare stateful)
```

A stateful interface must be imported using the `import!` top-level form, making the incorporation of state in the imported interface explicit. Similarly, the providing implementation must use the `provide!` form in order to provide definitions of elements declared in interfaces.

It is a compile-time error to reference a stateful interface using `import` or `provide`. Similarly, it is a compile-time error to reference a stateless interface using `import!` or `provide!`.

It is a compile-time error to use the `import!` or `provide!` forms in a source unit of compilation, or the `(declare stateful)` declaration in an interface unit of compilation, if the compiler command line option `--stateful` has not been supplied.

The use of stateful interface is *strongly discouraged*. In particular, we note that the inclusion of implicitly stateful interfaces in Java is the primary source of security failure in that language. We have incorporated stateful interfaces in BitC only because we have not yet been able to identify any effective way to write kernel code without them. Should a solution to this problem become apparent, support for stateful interfaces may well be retired from the language.

8.5 The Reserved Interface `bitc`

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

9 Storage Model

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

10 Pragmatics

10.1 Tail Recursion

BitC provides no syntactic form providing for non-recursive iteration. Instead, the language 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 return value of `g` is the final computation (and therefore the return value) computed by `f`. A function call is said to be **tail recursive** if it is implemented in such a way as to re-use its containing stack frame.

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

- Within any function `f`, calls to `f` that appear in tail position w.r.t. the body of `f` must be tail recursive.
- The `do` construct is properly tail recursive.
- 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.

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

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.

11 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:
;;(defunion string:val (vector char))

; Pairs:
(defstruct (pair 'a 'b):val
  fst:'a snd:'b)

; Nullable pointers:
(defunion (optional 'a):val
  none (some 'a))

; Homogeneous lists:
(defunion (list 'a):val
  nil (cons 'a (list 'a)))

; Bignums
(defunion int:val
  (fix (bitfield int32 31))
  (big (ref (bool, (vector word)))))
```

12 Foundational Type Classes

The standard prelude provides a number of standard type classes:

```
; Equality comparison by identity:
(deftypeclass (EqComparison 'a)
  eq : (fn ('a 'a) bool))

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

; Generalized equality:
(deftypeclass (EqualityComparison 'a)
  == : (fn ('a 'a) bool)
  != : (fn ('a 'a) bool))

; Magnitude comparison
(deftypeclass (Ord 'a)
  (super (EqualityComparison 'a))
  <  : (fn ('a 'a) bool)
  <= : (fn ('a 'a) bool))

; Checked arithmetic
(deftypeclass (Arith 'a)
  (super (Ord '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
  (deftypeclass (Ring 'a)
    (super (Ord '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
  (deftypeclass (Ring 'a)
    (super (Ord 'a))
    negate: (fn ('a) 'a)
    abs:    (fn ('a) 'a))
```

III Standard Library

13 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.

13.1 Built-In Operators

BitC defines the operation `length` to return the length of a vector.

The length operator is defined over the signature:

(vector) $\Rightarrow$ uint64

13.2 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 $\times$ int8 $\Rightarrow$ int8
int16 $\times$ int16 $\Rightarrow$ int16
int32 $\times$ int32 $\Rightarrow$ int32
int64 $\times$ int64 $\Rightarrow$ int64
uint8 $\times$ uint8 $\Rightarrow$ uint8
uint16 $\times$ uint16 $\Rightarrow$ uint16
uint32 $\times$ uint32 $\Rightarrow$ uint32
uint64 $\times$ uint64 $\Rightarrow$ uint64

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

13.3 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 $\times$ char $\Rightarrow$ bool
int8 $\times$ int8 $\Rightarrow$ bool
int16 $\times$ int16 $\Rightarrow$ bool
int32 $\times$ int32 $\Rightarrow$ bool
int64 $\times$ int64 $\Rightarrow$ bool
uint8 $\times$ uint8 $\Rightarrow$ bool
uint16 $\times$ uint16 $\Rightarrow$ bool
uint32 $\times$ uint32 $\Rightarrow$ bool
uint64 $\times$ uint64 $\Rightarrow$ bool

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

14 Verifi cation 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.

14.1 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 `#t` for all possible variable instantiations:

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

14.2 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 `#t` for all possible variable instantiations:

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

14.3 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 `#t` 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)
```

14.4 Theories

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

```
(deftheory name thm_1 ... thm_n)
```

where each *`thm`*$_i$ has been previously introduced by `defthm`.

14.5 Suspending and Enabling

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

```
(disable name_1 ... name_n)
(enable name_1 ... name_n)
```

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.

15 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.

A Change History

This section is non-normative.

This section is an attempt to track the changes to this document by hand. It may not always be accurate!

A.1 Changes (coming) in version 0.10

- Introduction and definition of `definvariant`, `defaxiom`, `deftheory`, `suspend`, `disable`, and `enable`.
- Revised UNICODE character literal syntax, dropping the radix syntax in favor of \{`U+`*`digits`*}. Dropped the curly braces around the special forms for non-printing character literals. The old forms will be retired in version 0.11.
- Revised string backslash embedding rules. Backslash within a string no longer shares a common syntax with character literals. Curly braced forms for non-printing characters in strings will be retired in version 0.11.
- The following statement from the 0.9 specification has been dropped, and may lead to incompatibilities in transitioning to language version 0.10.

  In general, if `#\`*`x`* is a valid character constant, then `"\`*`x`*`"` is a single character string consisting of the same character. By far the most common uses of this escaped encoding is to embed the double quote character and the backslash character within a string, but the generalization also permits encoding unicode characters by code point.

A.2 Changes in version 0.9

Cleaning up a few *more* loose ends:

- Removed `read-only`.
- Added keyword `external` to `proclaim`, along with optional external identifier.
- Renamed `vector-ref`, `array-ref`, `member-ref` to (respectively) `vector-nth`, `array-nth`, `member`.
- Restored the primitive types `bool`, `char`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `word`, `float`, `double`, `quad`.
- The `fixint` keyword is retired in favor of `bitfield`.
- Binding patterns in `letrec` forms must now be statically decomposable at compile time to avoid internal violation of the completeness restriction.
- Order of evaluation in `let` and `letrec` initializers is now specified. This is necessary because of side effects in impure expressions.
- Replaced LOOP with DO.
- Added a new type declaration qualifier `:opaque` that renders the type visible in its defining interface and providers but opaque to importers.
- Removed all references to `tuple`. This is replaced by a new keyword `pair`.
- Lambdas now take multiple arguments, and implicit pairing is now gone.
- The `one-of` type constraint had horrible problems, and has been removed. In consequence, `nth-ref` was split into `array-ref` and `vector-ref` and the indexing convenience syntax has been removed.
- Renamed `type-qualify` to `the`, following Common Lisp conventions. Removed the *`e`*`:`*`T`* convienced syntax for expressions. Types in binding patterns are still specified using the colon-qualified form.
- The `deftype` form was unmotivated and complicated. It is gone.
- Removed the reference chasing misfeature of `array-length`, `vector-length`, `nth-ref`, and `struct-ref`. Renamed `struct-ref` to `member-ref`.
- Renamed keyword `forall` to `method`.
- Replace `vector-ref` and `array-ref` with `nth-ref`
- Added `forall` as explicit syntax for writing method types.
- Added `one-of` as a type constraint, subject to the rule that the member types must not unify.
- *Issue:* Should we introduce labeling forms `check`, `assert`, `label`?
- Described (and required) a variant of the Cardelli optimization for list cell representations.
- Added type classes.
- Corrected definitions of `tuple`, `array` types to be value types. Corrected definition of `vector` to be a reference type.
- Removed `mutable` value constructor. Replaced `immutable` value constructor with `read-only`.
- Removed discussion of primary type conversion, since this is now part of the standard prelude.
- Removed discussion of canonical serialization.

- Discussed declaration of value vs. reference named types.
- Moved `case!` and `deftype` to the experimental section. We may restore these, but I want to try things without it first.
- Renamed `alias` to `use`.
- Removed the "literal type resolution" rule, per Mark Jones.
- Reverted the specification of `mutable`. Mutability is once again defined at field level. Temporarily *removed* the `mutable` and `immutable` value constructors..
- Renamed the `deref` type constructor to `val`.

A.3 Changes in version 0.8

The objective in Version 0.8 is to clean up a few loose ends before I dig in to type classes.

One cleanup that did *not* make it into this document was eliminating the references to "complete" and "incomplete" types. These aren't really issues of type at all. There is no problem from a typing perspective with a recursively defined value type. The issue is really an issue of instantiability, and if we treat it that way and discuss it separately from the discussion of types, we can deal with some of the complications that arise from mutable monomorphism in a more sensible way as well.

- Corrected the "literal type resolution" rule in section 3.2 to apply only to literals, noted Mark Jones's objection, and marked this as provisional.
- Re-introduced named let under the keyword `loop`. Corrected specification of `let`, which is not a derived form.
- Non-escaping types have been removed. This includes `restricted` and `sequence`. This required me to expand `sequence-ref` and `sequence-length` into `array-ref`, `array-length`, `vector-ref`, and `vector-length`.

  This change introduces an open issue, which is whether the array types should be redefined as `(array` *`T len`*`)`, where the type constructor is restricted to require a literal at the argument *`len`*. I am deferring the resolution of this issue until type classes have been added.
- Revised the specification of the `mutable` type constructor to indicate that its use appearances are syntactically restricted.
- Introduced `dup`, removing the `ref` value constructor.

A.4 Changes in version 0.7

Version 0.7 is primarily a pass to reconcile the provisional compiler with the specification, and perform some cleanups. The primary substantive change in this pass is to replace modules with interfaces.

Need to dig into the library some more and define equality types.

- Migrated `typecase` into the experimental section. Added a discussion of inference resolution in the primary types section and a set of type coercion operators in the expression section.
- Removed the Swaroop restrictions, since the implementor is no longer confused.
- Added `case!` to the specification.
- As a consequence of the change to interfaces and the deferral of typecase, clarified that `int32` and friends are proper keywords.
- Added new keywords `alias`, `import!` (was: `stateful-import`), `provide`, `provide!`, and `interface` to provide means for interface specification and name aliasing. Updated the description of compilation units to reflect this addition. Dropped the `module` and `export` keywords in favor of interfaces.
- Added new keyword `proclaim` for forward and external declarations. This obviated the need for `rectypes`, which is now removed.
- Added new keyword `immutable`, with the effect that it returns its argument, stripped of mutability. `immutable` is syntax, because it does not copy its argument.

  Clarified definition of `mutable` value constructor to indicate that it is a non-copying syntactic form.

  Relocated the Pragmatics section, which should have been part of the language specification rather than the library specification.
- Removed `defmacro`, which was a really bad idea.

- Re-merged the core and convenience forms into a single section, but annotate which is which. Having two sections to describe expressions made things unreasonably obfuscated.

A.5 Changes in version 0.6

Version 0.6 introduce interfaces, modules, and support for capability-safe programming. We are vaguely hopeful that the only major changes after this point will be fleshing out the standard library.

- Added a definition of the module system. Changed the definition of compilation units to reflect it.
- Removed discussion of storage model, layout, and alignment in the types section. Temporarily removed the major section on storage model, which needs serious rework.
- Moved `letrec` back into the core, since it cannot be lambda-expanded without the Y combinator, which does not type check in a statically typed lambda calculus.
- Added mention of the value restriction rule.
- Repaired the erroneous parenthesization of `cond`, `case`, and `typecase`. Changed definition of `cond` and `case` to have an explicit otherwise case.
- Clarified that the type inference engine must handle `typecase` with care to avoid type unification.
- Revised `typecase` to note that its result expressions need not have the same type. Also noted that failure to match is a static type error.
- Removed `int` and `nat`. Specified default types of unqualified integer literals. Added new type `word` which is the least unsigned integer type sufficient to hold a pointer representation.
- Split existing `vector` type into `array` and `vector`. Array is fixed length, vector is dynamic length. Vectors are unboxable, but internally reference heap-allocated storage. Both are served by the primitive accessor `sequence-ref` in order to continue to support the convenience syntax for array element reference. Added new core form `sequence-length` that returns the length of its argument value of sequence type.
- Replaced `box`, `unbox` with `ref`, `restricted-ref`. Changed `struct-ref`, `sequence-ref` to implicitly dereference their left hand argument as needed. In consequence it became possible to eliminate the convenience syntax for `deref`, which is now rare.
- Introduced `defmacro` and specified the expansion of convenience syntax into core language forms.
- Split the specification into core expression forms and convenience expressions.

A.6 Changes in version 0.5

Version 0.5 is a major rework of the language. The reader is advised to reread the entire specification, as there have been many changes with potentially subtle interactions.

- Added a “reserved words” section, including some reserved words set aside for use in future module and object systems.
- Noted that BitC is polymorphic, and that it imposes the Value restriction similar to ML.
- Added a syntax for describing procedure types.
- Removed field names from unions
- Replaced `typealias` with `deftype`
- Noticed that binding patterns in `define` already handled mutual recursion and consequently removed `mutual-recursion`. Introduced `rectypes` for recursive type declarations.
- Removed the `ref` keyword, replacing it with `mutable`, with the intended meaning that what we used to write as `(unbox ref x)` is now `(mutable x)`.
- Added an explicit type modifier keyword `box` to the language (the dual of `unbox`).
- State that primitive fixed-size types are unboxed by default, and the constructed types are boxed by default (compare Java).
- **Pending** Add explicit operators `box` and `unbox` to the language. Each creates a copy of its argument. The compiler is free to optimize out the copy when it can do so correctly.
- Removed the character `^` from the legal identifier characters. `^` is now a postfix operator indicating pointer dereference.

- Added a new form `(bitc-version "`*`version`*`")` where *version* is a string identifying the BitC language version in which the program was implemented. This form is syntax, and is permitted only at top-level. It *must* appear exactly once as the first form in every unit of compilation. It is a compile-time error if the compiler implementation does not implement the language-version specified in the input compilation unit.

- Resolved to specify the layout of data structure created using the `defunion` declaration, and to provide a means using for specifying the representation size of the "hidden" union tag:

```
(defunion u
  (cons1 arg1)
  (cons2 args)
  (declare (tag-type T)))
```

  where *T* must be compatible with some `fixint` of unsigned type.

- **pending** Need to specify assignment and binding compatibility of boxed and unboxed types. Resolution is that the language does *not* provide implicit boxing or unboxing.

- **pending** Need to specify assignment and binding compatibility of mutable vs non-mutable things. Resolution is that assignment has copy semantics, and so both assignments are legal. Further, consider the procedures:

```
(define (f x : (mutable T))
  body)
(define (g x : T)
  body)
```

  the mutability of *T* in *f* is strictly *internal* to the procedure. Because BitC has by-value semantics of procedure arguments the signature of *f* for purposes of external type checking and pattern matching is the same as the signature of *g*.

- Added radix prefixes and negation for integer literals. Introduced string literal syntax and specified full syntax for floating point literals. Revised character literal syntax for code points to be lexically similar. This may still need further revision, as the current syntax is exceptionally awkward. I am considering using HTML-style character entity syntax in place of the current one. Note that left and right curly brace are no longer supported using the simple character syntax.

- Added means to describe bitfields via `fixint`.

- Specified string literal syntax. Type of such literals is `(vector char)`. String type is needed in the primitive language in order to support diagnostic output. More precisely, it is needed in order to provide for string literals.

- Revised the definition of `tuple` to remove the requirement that `(tuple x)` is treated like `x`. This should never have been true in SML either.

- Removed the comma-style tuple construction syntax in the expression language, eliminating the syntactic ambiguity of how to interpret `(a)` (application or tuplization?).

- Swaroop and Jonathan initially had a difference of opinion concerning whether deftype should be resolved by textual substitution. The problem is that most of our types are resolved by name equivalence, so this appears to have rather limited utility in reducing verbiage.

  Actually, this is rather a big mess, as is illustrated by the definition of `tree-of` in the discussion of structures. There appears to be no *correct* definition of `tree-of` under our present rules.

A.7 Changes in version 0.4

- Description of tuples was augmented to describe syntax of tuple patterns and allow field names in tuple patterns.

- Added `typealias` declaration.

- Added pragmatics section requiring limited tail recursion.

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] 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] 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.

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