# Bit Level Types: Syntax and Semantics

David T Eger (eger@cs.cmu.edu)
Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213

September 22, 2005

## 0 To Do

Herein is a draft of the semantics for BLT. This draft is missing some key features from full-blown BLT, notably the constructs `implicit`, `dontcare`, `asbits`, wildcards, the locatives @.! and @!, support for opaque types such as IEEE 754 floats, and the specific rules for type embedding backends for (C/SMLNJ/etc.).

## 1 Overview

Suppose we are given a document which describes the Shockwave Flash File Format (SWF). This document will go to great lengths describing how to interpret the sequence of bits in a shockwave file as structured data. Certain sequences of bits will translate to strings of text. Certain sequences of bits will translate to data structures representing rectangles. Certain sequences of bits are raw data that must be run through a decompression algorithm before their true structure can be interpreted.

Suppose we would like to write a program in C which manipulates SWF files. We must take the prose and figures which are the SWF specification, and write down an interpretation of them as structured data in C. Some portions of the specification may not map cleanly to C. For instance, a rectangle is described in Flash as a structure with five fields:

```
RECT =
struct
      n : uint(5),
      xmin : uint(n),
      xmax : uint(n),
      ymin : uint(n),
      ymax : uint(n)
tcurts
```

*BLT spec of an SWF RECT*

```
typedef struct {
        unsigned char n;
        unsigned int xmin;
        unsigned int xmax;
        unsigned int ymin;
        unsigned int ymax;
} RECT;
```

*A Possible Interpretation in C*

The first (5-bit) integer field specifies how many bits each of the following four integer fields will take in the bit stream. From this we can infer that none of the fields in this structure will consume more than 32 bits, and so on a 32 bit platform it is safe to map SWF's notion of a rectangle to the C struct in the figure on the right. However, our interpretation of a SWF RECT in C *is not* a real SWF RECT. The *definition* of what a SWF RECT is is defined exactly by the SWF specification, which defines the *file format*.

Thus, there is often a *loss of information* about the structure of data when we interpret said data in a target language such as C. According to the SWF specification, the field *n* is only allowed to have values between 0 and 31. Our embedding in C, however, allows *n* to have any value between '$\backslash x00$' and '$\backslash 0xFF$'. In our embedding

in C, we have also forgotten the dependency between the field *n* and the fields *xmin*, *xmax*, *ymin*, and *ymax*. In a value of type SWF RECT, the latter four fields must each be an integer in the range 0 and $2^n - 1$. In our embedding in C, we know only that each must be between 0 and $2^{32} - 1$.

Nonetheless, we can create an approximation of what a SWF RECT is within C in terms of C's data types. We must then write marshalling code to convert between the data on disk and our approximation of it in C.

As we parse, we must be careful to interpret the data correctly: accounting for bit field issues, dependency issues, and validation conditions on the data. When we want to write the data back out to disk, we need to ensure that the invariants on a Flash File which are not captured by the C type system have been preserved.

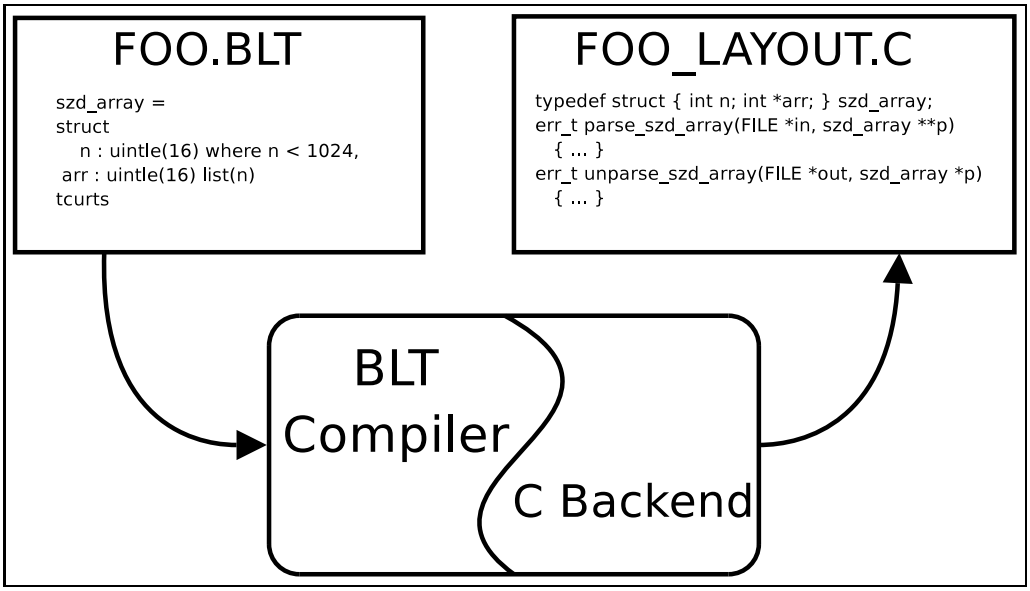
At the end of the day, we will have written two C functions:

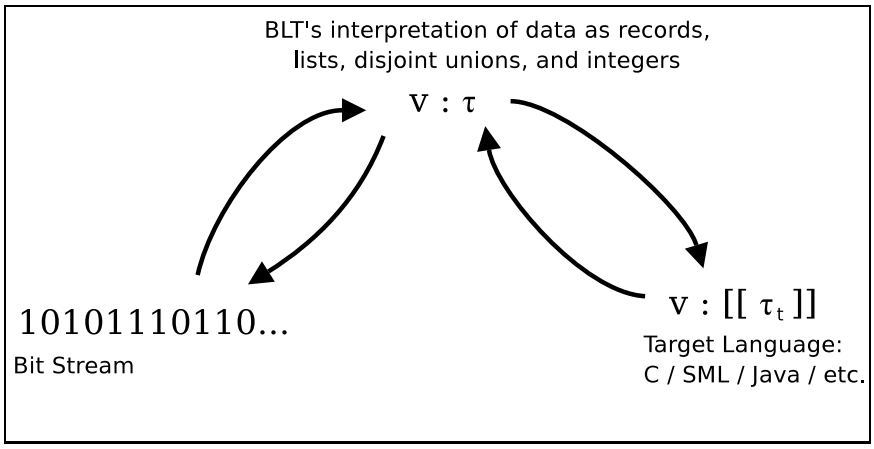
```
err_t parse_RECT(FILE * f, RECT **r);
err_t unparse_RECT(FILE * f, RECT *r);
```

The goal of BLT is to take the burden of interpreting low-level data off the shoulders of an application writer. Instead of laboriously translating a two-hundred page file format specification into parsing code, we provide a *specification language* (BLT). Each BLT specification specifies how a certain type of structured data is laid out on disk. The types of data we can specify are integers, records, disjoint unions, and lists. We provide a small set of terms for integer layout types and combinators for constructing layout terms for structured data out of smaller layout terms. A BLT compiler has a backend for a *target language* (be it C, SML, Java, etc.) which translates an abstract type in BLT to an appropriate concrete interpretation in that target language. For instance, the mapping of BLT types to C looks as follows:

| BLT type | int | $\{v_i : \tau_i\}_{i=1}^n$ | $[v_i : \tau_i]_{i=1}^n$ | $\tau$ list |
|---|---|---|---|---|
| *C type* | *One of* :<br>(un)signed char<br>(un)signed short<br>(un)signed int<br>(un)signed long<br>BIGNUM | struct{$([\![\tau_i]\!] v_i;)_{i=1}^n$}; | struct{<br>int dmx;<br>union{$([\![\tau_i]\!] v_i;)_{i=1}^n$};<br>}; | struct{<br>int len;<br>$[\![\tau]\!]$ *arr;<br>}; |

A BLT compiler will translate a data layout specification into appropriate `parse` and `unparse` functions in the programmer's target language.






An important design decision to mention before we go further is that our language is based on the presumption that the data we wish to parse can be parsed *predictively*. That is, we are working with some variant on the idea of LL(k) grammars: once we've looked at the next few bits of the incoming bit stream and decided on a branch of a union or whether a greedy list terminates, there's no need to backtrack.

We want this behavior so that we can parse streams in pieces. As an example, let's suppose we are writing a specification for the data sent over a socket in an X Protocol Session. We may describe this data as an XHandshake packet followed by a sequence of XRequests followed optionally by an XShutdown packet. This correctly describes a session between an X Client and an X Server. However, if we're using BLT to make a monitoring or fault injection tool, parsing the entire session is no good to us. We want the ability to change individual X Requests as they are made, and the X Protocol is designed so we can do this: each request, handshake, and shutdown packet can be parsed without knowing about any data that follows it.

One of our goals in designing BLT, then, will be to be able to provide warnings to the specification writer when his specification contains an ambiguous parse: one where the predictive parse is not the only parse.

## 2 Well Formed BLT Terms and Types

BLT is a language for concisely describing the layout of a data structure as a string of bits. Those terms which describe a data type and its layout are of type '$\alpha$ layout.' When compiled, such terms will be exposed to the programmer in a form reminiscent of combinator parsers: $\{\mathsf{parse} : \mathsf{bit\ string} \rightarrow [\![\alpha]\!] * \mathsf{bit\ string}, \mathsf{unparse} : [\![\alpha]\!] \rightarrow \mathsf{bit\ string}\}$, where $[\![\alpha]\!]$ is the translation of a BLT type into the target language's native type system.

| | | | |
|---|---|---|---|
| $\tau$ | ::= | int | Integer (Unbounded) |
| | | $\{\mathtt{l_i} : \tau_i\}_{i=1}^{n}$ | Record |
| | | $[\mathtt{l_i} : \tau_i]_{i=1}^{n}$ | Disjoint Sum |
| | | $\tau$ list | List of elements of type $\tau$ |
| | | $\tau$ layout | Layout of a $\tau$ on a bitstream |
| | | $\tau_1 \rightarrow \tau_2$ | Function |

| | | | |
|---|---|---|---|
| e | ::= | $\lambda v : \tau_1.\mathtt{e}_2$ | Lambda Abstraction |
| | | $\lambda v \wr \mathtt{e}_1.\mathtt{e}_2$ | Lambda Abstraction with Type – Directed Sugar |
| | | $v$ | Reference A Bound Variable |
| | | k | Konstant (see below) |
| | | $\mathtt{e}_1(\mathtt{e}_2)$ | Function Application |
| | | $\mathtt{e_i}$ where $\mathtt{e}_2$ | Type Refinement/Validation Condition |
| | | $\mathtt{e}_1$ asa $\mathtt{e}_2$ | Checked Coercion |
| | | $\mathtt{struct}(v_i : \mathtt{e}_i)_{i=1}^{n}\mathtt{tcurts}$ | Dependent Record Layout (no padding) |
| | | $\mathtt{union}(v_i : \mathtt{e}_i)_{i=1}^{n}\mathtt{noinu}$ | Simple Sum ($\mathtt{e_i}$ need ensure representation discriminability) |
| | | $\mathtt{e}_1\mathtt{list}(\mathtt{e}_2)$ | List of $\mathtt{e}_1$'s of Length $\mathtt{e}_2$ (no padding) |
| | | $\mathtt{e}_1\mathtt{list}$ | Greedy $\mathtt{e}_1$ List (no padding) |
| | | case e of $(v_i(v)=\!>\mathtt{e_i})_{i=1}^{n}$ | Elimination Form For Sums |
| | | $\mathtt{e}_1.v$ | Elimination Form For Records |

For convenience of presentation we will define $\mathsf{bool} \stackrel{def}{=} [true : \{\}, false : \{\}]$ and the syntax "if e then $\mathtt{e}_1$ else $\mathtt{e}_2$" as sugar for "case e of $true$=>$\mathtt{e}_1$, $false$=>$\mathtt{e}_2$." We'll also represent tuples in the common way: a tuple is just a record with natural number labelled fields 1...n.

| | | | *Type* | *Description* |
|---|---|---|---|---|
| k | ::= | true \| false | bool | Boolean Constant |
| | | AND \| OR | bool * bool → bool | Boolean Connectives |
| | | NOT | bool → bool | Boolean Negation |
| | | ..., −1, 0, 1, ... | int | Integer Constant |
| | | length | $\forall\alpha.\ \alpha$ list → int | Give the Length of a list |
| | | nth | $\forall\alpha.\ \alpha$ list * int → $\alpha$ | Project the $n^{\text{th}}$ Element of a list |
| | | = \| != \| < \| > \| <= \| >= \| | int * int → bool | Integer Comparison |
| | | + \| * \| / \| % \| - | int * int → int | Integer Operations |
| | | & \| \| \| << \| >> | int * int → int | "Bitwise" Operators (partial! see later) |
| | | uint | int → int layout | $n$ bit unsigned integer |
| | | uintle | int → int layout | $n$ bit unsigned little endian integer |
| | | sint | int → int layout | $n$ bit two's complement signed integer |
| | | sintle | int → int layout | $n$ bit two's complement signed little endian integer |

For syntactic convenience, we shall provide infix notation for the common arithmetic operators.

One of the oddities to note is that we have a number of symbols that are or should arguably be colons (see table below). This is due to the nature of the type $\tau$ layout.

We can view terms of type $\tau$ layout from a couple of viewpoints. First, we can view a term of type $\tau$ layout as specifying a marshaller for objects of type $\tau$ which implements one of several possible layout strategies.

Secondly, we can view terms of type $\tau$ layout as specifying refinements of $\tau$ according to its dependencies and where-clauses – things not accounted for by the BLT type system. (These constraints we will reason about using a form of Abstract Interpretation.) Both of these properties of layout terms are needed in order to formalize the prose that comprises the original specifications for binary formats.

We therefore view the syntactic constructs `struct ... tcurts`, `union ... noinu`, `asa`, `list` and `where` as marshaller combinators, mirroring the structure of the type system.
Having explained the correspondence between marshaller combinators (terms) and types, let us now explain the meaning of each colon wherever it may occur.

| Colon-like Symbol | Example Usage | Meaning |
|---|---|---|
| : | $\{\mathsf{a} : \mathsf{int}, \mathsf{b} : \mathsf{int}\}$ | Type of a field of a Record Type |
| : | $[\mathsf{a} : \mathsf{int}, \mathsf{b} : \mathsf{int}]$ | Type of a labeled variant of a Union Type |
| : | `term:typ` | The BLT term `term` has the type `typ` |
| : | `struct` $a$ : `uint(8) tcurts` | Part of the syntax of the `struct` record layout introduction form: This field should be marshalled by the layout term on the right |
| : | `union` $a$ : `uint(8) noinu` | Part of the syntax of the `union` sum layout introduction form: This branch should be marshalled by the layout term on the right |
| : | $\lambda v : \tau_1.\mathtt{e}_2$ | Lambda Abstraction |
| ≀ | $\lambda v \wr \mathtt{e}_1.\mathtt{e}_2$ | Type-Directed Sugar. $\mathtt{e}_1$ must be of type $\tau$ layout, and this is equivalent to $\lambda v : \tau.\mathtt{e}_2$ |

Let's start with an example:

$$\texttt{struct } a : \texttt{uint(8)}, b : \texttt{uint(16) tcurts}$$

is of type "$\{\mathsf{a} : \mathsf{int}, \mathsf{b} : \mathsf{int}\}$ layout". `struct` constructs its marshallers by sequencing a series of marshallers, passing the partially parsed results forward during the parse phase, and passing the complete parsed structure to each marshaller during the unparse phase. Therefore, the term to the right of "$a$ :" which is "`uint(8)`" needs to be a layout type, which it is.

Most of our table should be familiar, but our syntactic sugar "squiggle" may need some explanation. As illustrated in the extended example in Section 5, BLT specifications consist of a sequence of declarations. It is common for a layout term (in the example, cu16) to depend on something previously parsed (in the example, an XEndianness field). Therefore, it is natural to want to indicate an argument by simply naming the layout term which parsed the data instead of writing the BLT type of the parsed data explicitly.

Nonetheless, we may occasionally want to write a plain lambda, which we illustrate by:

$$min = \lambda x : \mathsf{int}.\ \lambda y : \mathsf{int}.\ \texttt{if } x \texttt{<} y \texttt{ then } x \texttt{ else } y$$

# 3 Type Equality

$$\frac{}{\mathsf{int} =_\mathsf{T} \mathsf{int}}\ \textsc{Eqt-Int} \qquad \frac{m = n \quad (\tau_i =_\mathsf{T} \sigma_\mathtt{i} \text{ and } \mathtt{l_i} = \mathtt{m_i} \text{ (as strings)})_{i=1\ldots n}}{\{\mathtt{l_i} : \tau_\mathtt{i}\}_{i=1}^{m} =_\mathsf{T} \{\mathtt{m_i} : \sigma_\mathtt{i}\}_{i=1}^{n}}\ \textsc{Eqt-Rcd}$$

$$\frac{m = n \quad (\tau_i =_\mathsf{T} \sigma_\mathtt{i} \text{ and } \mathtt{l_i} = \mathtt{m_i} \text{ (as strings)})_{i=1\ldots n}}{[\mathtt{l_i} : \tau_\mathtt{i}]_{i=1}^{m} =_\mathsf{T} [\mathtt{m_i} : \sigma_\mathtt{i}]_{i=1}^{n}}\ \textsc{Eqt-Sum}$$

$$\frac{\tau =_\mathsf{T} \sigma}{\tau\ \mathsf{list} =_\mathsf{T} \sigma\ \mathsf{list}}\ \textsc{Eqt-List} \qquad \frac{\tau =_\mathsf{T} \sigma}{\tau\ \mathsf{layout} =_\mathsf{T} \sigma\ \mathsf{layout}}\ \textsc{Eqt-Layout} \qquad \frac{\tau_1 =_\mathsf{T} \tau_3 \quad \tau_2 =_\mathsf{T} \tau_4}{\tau_1 \to \tau_2 =_\mathsf{T} \tau_3 \to \tau_4}\ \textsc{Eqt-Fn}$$

# 4 Static Semantics

For each of the constants k above, we have a rule saying that in any context, said term is of the type we specified in the table. The constants are not all truly constants per sé, as for instance our BLT type system does not account for universal quantifiers. We account for terms like `length` and `nth`, then, using the same sort of "bolted on" approach that Pierce uses in the early chapters of TAPL. The rules for our constants follow from their types, so we omit them for brevity.

$$\frac{\Gamma, v:\tau_1 \vdash e_2:\tau_2}{\Gamma \vdash (\lambda v:\tau_1.e_2):\tau_1 \to \tau_2}\ \text{S-AbsT} \qquad \frac{\Gamma \vdash e_1:\tau_1\ \text{layout} \quad \Gamma, v:\tau_1 \vdash e_2:\tau_2}{\Gamma \vdash (\lambda v \wr e_1.e_2):\tau_1 \to \tau_2}\ \text{S-AbsLayout}$$

$$\frac{\Gamma \vdash e_1:\tau_1 \to \tau \quad \Gamma \vdash e_2:\tau_1}{\Gamma \vdash e_1(e_2):\tau}\ \text{S-App}$$

$$\frac{\Gamma \vdash e:[l_i:\tau_i]_{i=1}^n \qquad (l_i = v_i \text{ and } \Gamma, v:\tau_i \vdash e_i:\tau)_{i=1}^n}{\Gamma \vdash \texttt{case e of } (v_i(v) => e_i)_{i=1}^n : \tau}\ \text{S-ElimUnion}$$

$$\frac{\Gamma \vdash foo:\{v_i:\tau_i\}_{i=1}^n}{\Gamma \vdash foo.v_k:\tau_k}\ \text{S-ElimRcd}$$

$$\frac{\Gamma \vdash e_1:\alpha\ \text{layout} \quad \Gamma, \$!:\alpha \vdash e_2:\text{bool}}{\Gamma \vdash e_1\ \texttt{where}\ e_2:\alpha\ \text{layout}}\ \text{S-Refine} \qquad \frac{\Gamma \vdash e_1:\alpha\ \text{layout} \quad \Gamma \vdash e_2:\alpha_2\ \text{layout}}{\Gamma \vdash e_1\ \texttt{asa}\ e_2:\alpha_2\ \text{layout}}\ \text{S-ReInterpret}$$

$$\frac{\Gamma \vdash e_1:\alpha\ \text{layout}}{\Gamma \vdash e_1\ \texttt{list}:\alpha\ \text{list layout}}\ \text{S-List} \qquad \frac{\Gamma \vdash e_1:\alpha\ \text{layout} \quad \Gamma \vdash e_2:\text{int}}{\Gamma \vdash e_1\texttt{list}(e_2):\alpha\ \text{list layout}}\ \text{S-NList}$$

$$\frac{\Gamma \vdash e_i:\tau_i\ \text{layout} \quad \text{for } i \in [1 \ldots n] \quad v_i\ \text{all distinct}}{\Gamma \vdash \texttt{union}(v_i:e_i)_{i=1}^n\texttt{noinu}:[v_i:\tau_i]_{i=1}^n\ \text{layout}}\ \text{S-Union}$$

$$\frac{}{\texttt{struct tcurts}:\{\}_{i=1}^0\ \text{layout}}\ \text{S-StructBase}$$

$$\frac{\Gamma \vdash \texttt{struct}\ (v_i:e_i)_{i=1}^{n-1}\ \texttt{tcurts}:\{v_i:\tau_i\}_{i=1}^{n-1} \qquad \Gamma, (v_i:\tau_i)_{i=1}^{n-1} \vdash e_n:\tau_n\ \text{layout}}{\Gamma \vdash \texttt{struct}\ (v_i:e_i)_{i=1}^n\ \texttt{tcurts}:\{v_i:\tau_i\}_{i=1}^n\ \text{layout}}\ \text{S-StructStep}$$

*Static Semantics for BLT (sans rules for Konstants)*

The first two rules of our static semantics, S-AbsT and S-AbsLayout re-iterate the distinction between our two colons at the term level. S-App is straight-forward, but let us walk through the next four rules one at a time, hinting at our operational semantics. S-Refine would be used to typecheck a term such as:

$$\texttt{uint}(8)\ \texttt{where}\ \$! > 5$$

This term is of type int layout, and is different from `uint(8)` in that the integer represented (that is, the integer resulting from parsing or provided to unparse) must be greater than 5. Thus where-clauses represent refinements or validation conditions on data (i.e. boolean predicates that must be satisfied). The symbol "$!" should be read as a "this" symbol, bound in the body of $e_2$. A derivation for the above example can be worked as follows (rules for Konstants elided):

$$\dfrac{\dfrac{\cdot \vdash \texttt{uint} : \mathsf{int} \to \mathsf{int\ layout} \quad \cdot \vdash 8 : \mathsf{int}}{\cdot \vdash \texttt{uint}(8) : \mathsf{int\ layout}}\ \textsc{S-App} \quad \dfrac{\$! : \mathsf{int} \vdash (\$!, 5) : \mathsf{int} * \mathsf{int} \quad \$! : \mathsf{int} \vdash > : \mathsf{int} * \mathsf{int} \to \mathsf{bool}}{\$! : \mathsf{int} \vdash \$! > 5 : \mathsf{bool}}\ \textsc{S-App}}{\cdot \vdash \texttt{uint}(8)\ \texttt{where}\ \$! > 5 : \mathsf{int\ layout}}\ \textsc{S-Refine}$$

A Re-Interpreted type is a form of type-cast. The use of this construct is two-fold: first to guarantee that the low-level bits may be interpreted as a value of the type specified by $\texttt{e}_1$, and secondly, $\texttt{e}_1$ determines the size of the prefix of the bitstream being parsed that must be interpreted as an $\texttt{e}_2$.

We have two layout constructors for lists. The first simply takes an $\alpha$ layout and uses it greedily in the parse stage to read in a list of $\alpha$'s. The second reads a list of $\alpha$'s of some integer length $\texttt{e}_2$.

`union`s represent the layout of disjoint union types. This is one point where the fact that our system is based on *predictive* parsing shows especially. Parsers for `union` types try each $\tau_i$ layout in turn until one successfully parses the prefix to the bit stream, and *does not ever try a different branch later on the same prefix*. To ensure round-trip preservation of data types, the specification writer needs to determine that the branches of a `union` encode to different bit strings.

The rules for constructing dependent records are fairly straight-forward: later fields' marshallers can be determined using the values parsed as earlier fields.

# 5 An Extended Example

Let's finish off this section by illustrating the first fragment of the X Protocol.

```
u8 = uint(8)

XEndianness = union
                BIG    : u8 where $! =' B',
                LITTLE : u8 where $! =' l'
              noinu

cu16 = λc⟅XEndianness.
         case c of   BIG(v)    => uint(16)
                   | LITTLE(v) => uintle(16)

padto = λn : int.λsz_align : int.
          if n%sz_align = 0 then 0
          else n+(sz_align - n%sz_align)

XClientHandshake = struct
                     byte_order aka c : XEndianness,
                     unused           : u8,
                     protocol_major_version : cu16(c) where $! = 11,
                     protocol_minor_version : cu16(c),
                     n                : cu16(c),
                     d                : cu16(c),
                     protocol_name    : u8 list(n),
                     p0               : u8 list(padto(n)(4)),
                     protocol_data    : u8 list(d),
                     p1               : u8 list(padto(d)(4))
                   tcurts
```

The above specification[1] is elaborated to have the following types:

$$\texttt{u8} : \mathsf{int\ layout}$$

[1] BLT Specifications consist of a list of named expressions which are to be compiled and provided to the application programmer as a library. The two features we use here which we have not explained heretofore are parentheses within arithmetic expressions for grouping, and the '`aka`' construct, which provides an alias for an otherwise long field name within a structure declaration.

$$\text{XEndianness}: \ [\text{BIG} : \mathsf{int}, \text{LITTLE} : \mathsf{int}]\ \mathsf{layout}$$

$$\text{cu16}: \ [\text{BIG} : \mathsf{int}, \text{LITTLE} : \mathsf{int}] \to \mathsf{int}\ \mathsf{layout}$$

$$\text{padto}: \ \mathsf{int} \to \mathsf{int} \to \mathsf{int}$$

$$\begin{array}{rl} \text{XClientHandshake}: & \{ \ \text{byte\_order} : [\text{BIG} : \mathsf{int}, \text{LITTLE} : \mathsf{int}], \ \text{unused} : \mathsf{int}, \\ & \ \ \text{protocol\_major\_version} : \mathsf{int}, \\ & \ \ \text{protocol\_minor\_version} : \mathsf{int}, \\ & \ \ \text{n} : \mathsf{int}, \ \text{d} : \mathsf{int}, \\ & \ \ \text{protocol\_name} : \mathsf{int\ list}, \ \text{p0} : \mathsf{int\ list}, \\ & \ \ \text{protocol\_data} : \mathsf{int\ list}, \ \text{p1} : \mathsf{int\ list} \ \}\ \mathsf{layout} \end{array}$$

Note that in the definition of XClientHandshake, even though "`uint(16)`" and "`uintle(16)`" are not the same, because they are both of type "int layout," they can be used as alternate branches of a `case`. One could similarly have "`uint(8)`" and "`uintle(32) where $! ≠ 56`" as branches of a `case` statement, but the sort of dependencies BLT can express are only of this sort (those for which the parsed data is the same BLT type). The following BLT specification would not type-check since the branches of the `if` statement are of different types: [BIG : int, LITTLE : int] layout versus int layout.

$$\lambda n : \mathsf{int}.\ \texttt{if}\ n > 0\ \texttt{then XEndianness else uint}(32)$$

# 6 Operational Semantics

One of the things that may seem strange about our Static Semantics is that there are basically no elimination forms for layout terms. Sure, we can use layout terms to construct bigger layout terms, but there's no direct way to use these terms within BLT. This is by design. Layout terms are only actually used in their compiled form from within the target language to marshall structured data.

Our presentation of BLT's Operational Semantics is for definitional purposes, and as such have left optimizations to the implementor. Nonetheless, there are some fundamental issues that we will address later which we gloss over here: those of choosing appropriate representations for data. For instance, we would like to interpret a `uint(8) list` in C as a `struct{int len; char *arr}`; instead of as a `struct {int len; BIGNUM *arr};`. For now, we ignore this detail and operate in an idealized setting where all integers in our computations have the semantics of BIGNUMS. We provide our operational semantics by means of a canonical compilation of a BLT expression into an SML library. These operational semantics mirror what an actual implementation would look like. We avoid the detail of reifying an abstract data structure of a BLT type into the target language by operating on native SML data explicitly.

**XXX See attached unhygenic macro-ish SML. Can I use Aleks's formalism here?**

# 7 Operational Semantics (take 2)

Here, we provide interpretation of a BLT layout terms `t` by parse and unparse relations they define on strings of bits $s$:

$$\begin{array}{cc} s \cdot s', \mathsf{e} \overset{p}{\mapsto} v, s' & \mathsf{e}\ \text{parses the prefix}\ s\ \text{to the value}\ v\ \text{leaving}\ s' \\ v, \mathsf{e} \overset{u}{\mapsto} s & \mathsf{e}\ \text{unparses the value}\ v\ \text{to the string}\ s \\ \mathsf{e}_1 \mapsto \mathsf{e}_2 & \mathsf{e}_1\ \text{steps to}\ \mathsf{e}_2\ \text{without affecting the state} \\ \mathsf{e}_1 \to \mathsf{e}_2 & \text{Transitive closure of}\ \mapsto \end{array}$$

$$\frac{\mathsf{e}_1 \mapsto \mathsf{e}_1'}{\mathsf{e}_1(\mathsf{e}_2) \mapsto \mathsf{e}_1'(\mathsf{e}_2)}\ \text{D-AppProg1} \qquad \frac{\mathsf{e}_2 \mapsto \mathsf{e}_2'}{\mathsf{e}_1(\mathsf{e}_2) \mapsto \mathsf{e}_1(\mathsf{e}_2')}\ \text{D-AppProg2}$$

$$\frac{s\cdot s', \mathtt{e}_1 \overset{p}{\mapsto} v_0, s' \quad s\cdot \varepsilon, \mathtt{e}_2 \overset{p}{\mapsto} v, \varepsilon}{s\cdot s', \mathtt{e}_1\ \mathtt{asa}\ \mathtt{e}_2 \overset{p}{\mapsto} v, s'}\ \text{DP-ASA}$$

$$\frac{s\cdot s', \mathtt{e}_1 \overset{p}{\mapsto} v, s' \quad [v/\$!]\mathtt{e}_2 \to \mathtt{true}}{s\cdot s', \mathtt{e}_1\ \mathtt{where}\ \mathtt{e}_2 \overset{p}{\mapsto} v, s'}\ \text{DP-WHERETRUE} \qquad \frac{s\cdot s', \mathtt{e}_1 \overset{p}{\mapsto} v, s' \quad [v/\$!]\mathtt{e}_2 \to \mathtt{false}}{s\cdot s', \mathtt{e}_1\ \mathtt{where}\ \mathtt{e}_2 \overset{p}{\mapsto} \mathtt{error}, s\cdot s'}\ \text{DP-WHEREFALSE}$$

$$\frac{}{\varepsilon\cdot s', \mathtt{structtcurts} \overset{p}{\mapsto} \langle\rangle, s'}\ \text{DP-STRUCTBASE} \qquad \frac{}{s\cdot s', \mathtt{struct}ts, \mathrm{field}^o, \mathit{field}^i : \mathtt{e}}\ \text{DP-STRUCTSTEP}$$

# 8 Choosing An Appropriate Representation

We need to account for a fundamental optimization–that which allows us to choose an appropriate interpretation in the target language for a specific layout. For instance, we want to interpret a $\mathtt{uint}(8)\ \mathtt{list}$ in C as a $\mathtt{struct}\{\mathtt{int\ len; char} * \mathtt{arr}\}$; instead of as a $\mathtt{struct}\ \{\mathtt{int\ len; BIGNUM} * \mathtt{arr}\}$;.

Let us consider a term $\mathtt{t}:\tau$ layout. The parser which corresponds to $\mathtt{t}$ will parse only a subset of values of type $\tau$. For instance, a value parsed by a parser for the term $\mathtt{uint}(8)$ must be an integer between 0 and 255 inclusively. If $\mathtt{t}:\tau$ layout, let $\tau_\mathtt{t}$ denote the subset of the values of type $\tau$ that $\mathtt{t}$ might parse.

Our requirement for the embedding type $\mathtt{e}$ in the target language for the layout term $\mathtt{t}$ is that $\tau_\mathtt{t} \subseteq e \subseteq \tau$ or equivalently $\tau_\mathtt{t} \leq e \leq \tau$. The relation between $e$ and $\tau$ is witnessed by coercions: $c_{\mathtt{e}\to\tau}$ and $c_{\tau\to\mathtt{e}}$ We do not require that $e = \tau_\mathtt{t}$ since $\tau_\mathtt{t}$ may not be representable in the target language, or even if encodable, $\tau_\mathtt{t}$ may not be an efficient representation. Furthermore, calculating a "smallest" $e$ in a particular language's type system is often NP-Hard (see Appendix I).

A first pass approximation for finding reasonably appropriate representations is to use the range information that is easily provided by the type constructors themselves. The use is straight forward when our integer type constructors are applied to constants as in the case of $\mathtt{uint}(8)$. We know immediately that any integral type that can represent the range $[0\ldots255]$ will do. A slightly more difficult example is that of the SWF RECT, but this still seems within reason: we calculate the required range of the field *xmin* by $\bigvee_{i=1}^{31}\mathtt{uint}(i)$ to find that any integral type that can represent $[0\ldots2,147,483,647]$ will do.

Unfortunately, we might not always want to perform that calculation. For instance, we would not want to try the same trick with the field $x$ in the specification:

```
WOAH =
struct
      n : uint(16),
      m : uint(16),
      p : uint(16),
      x : uint(5*n*n*m - p*n + p*p -n*n*p),
tcurts
```

**XXX Insert intutions here that we will do simple range analysis on BLT Specifications to choose appropriate types. Abstract Interpretation á la Cousot and Cousot Anyone?**

# 9 Distinguishability

An important goal not covered heretofore is that of warning the specification writer that he may have written an ambiguous grammar. We prove that determining whether a BLT specification is ambiguous is NP Hard in Appendix II.

Despite the fact that this problem is NP Hard, there are good strategies for making unambiguous grammars. The most straight-forward approach is *tagging*: guaranteeing that a bit pattern specified by one branch of a `union` will not appear in another branch.

Deriving this information requires the interpretation of where-clauses by our primitive layout constructors. With the base constructors we have given, we are guaranteed that singleton values will correspond to unique bit-patterns. This need not be the case in general, though, as the following example illustrates:

$$(\texttt{if } x > 0 \texttt{ then uint}(8) \texttt{ else uint}(32)) \texttt{ where \$!} = 5$$

**XXX Display an algorithm we will use to attack this problem, explain the subtleties of `lists` and follow sets, as well as `unions`.**

# Appendix I : NP Hardness of BLT Embedding Optimization Problem

[BLT-OPT] **BLT Optimal Embedding Problem**
INSTANCE: BLT Specification $S$, Lattice of Integer Types (i.e. ranges) in a target language such as C
QUESTION: What is the optimal embedding for each named layout term $t \in S$ as a type constructed using the given integral types.

We will show that MAX2SAT is reducible to BLT-OPT.

Consider the lattice of integral types one might consider for an embedding in C: unsigned char (8 bit), unsigned short (16 bit), unsigned int (32 bit), unsigned long (64 bit), and BIGNUM. Suppose *Opt* is a function which calculates the optimal embedding $e \geq \mathtt{t}_\tau$ given this lattice for a layout term `t` in a BLT Specification.

Let $v_1, v_2 \ldots v_n$, $\{x_i, y_i\}_{i=1}^m$, $k$ be an instance of the MAX2SAT over $n$ variables and $m$ clauses. Here each $x$ and $y$ is either $v_j$ or $\neg v_j$. Let $\hat{x} \equiv 1 - v_j$ if $x = \neg v_j$ and $\hat{x} = v_j$ if $x = v_j$.

Examine the following BLT Specification:

| ZERO_OR_ONE = `uint`(8) `where` \$! = 0 `OR` \$! = 1 |
|---|
| GT_K_SAT = |
| `struct` |
| $v_1$ : ZERO_OR_ONE, |
| $v_2$ : ZERO_OR_ONE, |
| $\vdots$ |
| $v_n$ : ZERO_OR_ONE, |
| $z$ : `uint`(16) `where if` $\sum_{i=1}^m \hat{x}_i * \hat{y}_i > k$ `then` \$! = 255 `else` \$! = 65535 |
| `tcurts` |

Then $Opt(\text{GT\_K\_SAT}) = \texttt{struct}\{\texttt{char x}_1\texttt{; char x}_2\texttt{;} \cdots \texttt{char x}_n\texttt{; unsigned short z;}\}$; if and only if the MAX2SAT instance is satisfiable.

# Appendix II : NP Hardness of Ambiguity Detection

[BLT-AMBIGUITY] **BLT Ambiguity Problem**
INSTANCE: BLT Specification $S$
QUESTION: Can a sequence of bits be interpreted differently if we were to construct a non-predictive parser. (We could create a non-predictive parser by choosing non-deterministically the order of branches tried in unions and the stopping points for lists of unknown length.)

We will show that MAX2SAT is reducible to BLT-AMBIGUITY. Suppose *Amb* can decide whether a BLT specification is ambiguous. Examine the following extension of the BLT Specification in Appendix I:

| |
|---|
| ALWAYS_YES = <br> `struct` <br> $v_1$ : ZERO_OR_ONE, <br> $v_2$ : ZERO_OR_ONE, <br> $\vdots$ <br> $v_n$ : ZERO_OR_ONE, <br> $z$ : `uint`$(16)$ `where` $\$! = 65535$ <br> `tcurts` |
| AMBIGUOUS = <br> `union` <br> K_CLAUSES_SATISFIABLE : GT_K_SAT <br> K_CLAUSES_UNSATISFIABLE : ALWAYS_YES <br> `noinu` |

Then $Amb(AMBIGUOUS)$ is true if and only if the MAX2SAT instance is satisfiable.