Bit Level Types

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

March 25, 2005

Abstract

We present a language (BLT) for specifying binary formats through a collection of types. BLT specifications can describe both files (e.g., JPEG, ELF, MIDI) and network packets (e.g., IP, TCP, X). By writing a BLT specification for a format, we create a formal definition of *what a valid instance of the format is.* Further, a BLT specification provides enough information to generate parsers and encoders to mediate between the raw binary format and its canonical embedding in a user's programming language of choice, thereby eliminating the error-prone process of writing such code by hand.

1 Introduction

To share information, we rely on standard data formats: many are text-based (e.g., XML, HTML, C, RTF). Just as many data formats are based on bit- and byte-level information packing, both for files (e.g., JPEG, ELF, MIDI), and for network packets (e.g., IP, TCP, X). Keeping track of endianness, bit-ordering, alignment, and other issues of data validation is a laborious and error-prone task. Further, getting this job of parsing and encoding low-level data formats wrong can have disastrous consequences: In the fall of 2004, broken JPEG parsers in the Windows and Mozilla codebases were found which allowed an attacker to run arbitrary attacker-supplied code on a user's computer [17, 16]. Since 2002, the Ethereal network packet analyzer—a standard security tool which is mainly low-level parsing code—has had over three dozen reported vulnerabilities [15].

Why are there so many defects in code dealing with low-level formats? There are two main problems: (1) low-level formats are often specified by an informal mix of prose and figures, and (2) the implementations of parsers and encoders for these formats are most often hand-coded in C.

Here we present a language for precisely specifying binary formats through types. To be of direct use to a programmer, BLT needs a *binding* for her programming language of choice (C, SML, Java, etc.). A backend takes a BLT specification and generates a set of types in the target language that best represents the binary format, along with parsers and encoders which mediate between instances of the binary format and their corresponding representation in the target language. These parsers and encoders take care of endianness, bit-packing, re-boxing, and re-tagging under-the-hood so an application programmer is freed from worrying about the low-level details.

In addition to ease-of-use, BLT has been designed with security in mind. BLT-generated parsers and encoders calculate with infinite precision integer arithmetic and have no fixed buffers. As a result, these routines are immune from exploits based on buffer overflow, integer underflow, and integer overflow. Further, data validation and error-handling code which might otherwise have been over-looked in the tedium of writing low level parsing code is automatically generated from the data specification.

2 Binary Formats Are Different

With extensive literature on parsing regular and context free languages, it is natural to ask what makes binary formats different. After all, if we can accomplish our task with Lex and Yacc, then we might do well not to design a new language.

To help guide our discussion, let us pin down a few low level formats we would like to describe with BLT:

the X Protocol [20], the JFIF/JPEG format [13], and the Macromedia Flash File Format [8].

What sort of notions might we find in a low level format? We could encounter a zero-terminated ASCII string, which we'll represent in BLT as:

```
CString =
struct
   str :  (u8 where $! != 0) list,
   null :  u8 where $! = 0
tcurts
```

Here, $! should be read as "the current value being transformed by the active Ψ type (u8 in both instances)" In the first expression, $! stands for an element of the list; in the second expression $! is synonymous with the value of the *null* field. u8 means an 8-bit unsigned integer.

This particular example provides no challenges to the standard language heirarchy: it is a simple regular expression: $([\wedge\backslash 0])^*[\backslash 0]$. Further, constructs like the C string do occur in formats we are considering. Macromedia's Shockwave Format (SWF), has a control tag NamedAnchor, which contains a null-terminated string (and no length field). The JPEG format has a more elaborate variant on a C string: so-called "entropy-coded segments" which are terminated with a marker of the form 0xFF 0xYY where 0xYY $\neq$ 0x00. This we could write in BLT as[1]:

```
ECS_Datum =
union
   NON :  u8 where $! != 0xFF,
   ENT :  u8 list(2) where $!.(1) = 0,
noinu

EntropyCodedSegment
   = ECS_Datum list asa u8 list
```

Alternatively, it's just as common to see a variant on Pascal strings: that is, a data type where you have an integer n specifying the length of a list of things that follows. We view this as a form of dependent typing since part of the type definition depends on a runtime value: n. To exemplify: Every request and reply packet in the X Protocol has a length field indicating the packet's length. JPEG segments which are not entropy-encoded also have length fields. Though in these two formats the length fields are simple 16-bit integers, this is not always the case. In Shockwave, a rectangle is specified as a 5-bit integer n followed by four integers, each of length n. We would express Shockwave's rectangle type RECT in BLT as[2] :

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

These examples give several reasons why classical language descriptions fall apart. First, we are interpreting part of our data stream not as a simple symbol to determine the next machine state, but as an *integer value*. Our specifications can refer to these values and also to the structure of previously parsed data. So for example, we can recognize strings of the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n\mathtt{d}^n\mathtt{e}^n$ with the BLT specification[3]:

```
ABCDE =
struct
   A :  (u8 = 'a') list,
   B :  (u8 = 'b') list(length(A)),
   C :  (u8 = 'c') list(length(A)),
   D :  (u8 = 'd') list(length(A)),
   E :  (u8 = 'e') list(length(A))
tcurts
```

whereas no regular expression, context-free grammar, or tree-adjoining grammar can. Second, the symbols of the languages we are interested in are not all the same size: something RECT puts in bold relief.

Another notion peculiar to low level formats is the representation of integer values. We must specify both the integer length, whether it is an unsigned or signed value, and its endiannesss. Further, each of these properties of any particular field may be dependent upon a previous field. In the X protocol, for instance, endianness of the integer fields for a session is determined by the value of a byte in the client's first packet.

While other specification languages have attempted to capture the needed qualities for describing binary formats, most fall short: (1) none of them handle dependent endianness cleanly, (2) most of them do not handle dependent typing constructs, (3) most depend on the dangerous arithmetic semantics of the

[1]You may be curious about what asa does. It lets us verify some properties of data using one type, and then "forget" that type and re-interpret the parsed data as some other type, a value of which it passes to the programmer

[2]Here we admit that u8 in our first example was shorthand for uint(8).

[3]Here we use some syntactic sugar so that we don't have to write out " where $!" for exceptionally simple where-clauses

target language, (4) many only provide for parsing, but not for round-trip data transformation and (5) several languages give up declarativity in favor of generality, leaving the semantics to the programmer. Languages that we quibble with on this last point result in specifications (if one can even call them that) which have ambiguous data projections and a haphazard, error-prone structure.

3 Guiding Principles

Certain principles have guided our design choices for BLT:

1. BLT should be well-defined.
2. The types specified in BLT should map cleanly to SML and C.
3. BLT specifications should be short and clean.
4. Parsed data should be ready-to-use.
5. Our parsers and encoders should do as much data validation as reasonable.
6. We should use taste in determining the generated parsers' level of complexity: as a reference, we should try to keep the generated parsing routines to linear time. **XXX linear? really?**

Not all of these points are orthogonal. Doing full data validation may be very expensive, as may making data ready-to-use: consider specifications which describe lists of primes or compressed data. In light of the trade-offs involved, designing a good language for this niche requires a certain sense of good taste.

4 The Big Picture

The main expressions of interest in BLT are those which define bijections between valid instances of a specified binary format and valid projections of said instances into a target programming language. Such an expression defines a Ψ type. Let us look at a small example of such an expression:

```
struct
   n :  u16/le  where n < 1024,
   arr :  u16/le list(n)
tcurts
```

This expression defines a Ψ type that stands for relatively short lists of integers: we have an unsigned 16-bit little-endian length field n, whose where-clause tells us that it must be less than 1024. After n follow n more unsigned 16-bit integers comprising the contents of the list. A question arises as to what language the expression "$n < 1024$" is evaluated within. It is tempting to define the expressions used in where-clauses to simply be terms of the target programming language. This is the design choice of Yacc. Yacc relies on the programmer to create and piece together a suitable data structure through a sequence of calls to various semantic actions during parsing. In contrast, our specifications have declarative semantics: the types corresponding to the low-level data are well-defined by the specification itself. In addition to this design goal difference, we argue that relying on the target language is bad for two reasons: (1) It makes a data specification unnecessarily brittle and dependent upon a specific programming language, and (2) Relying on the semantics of most programming languages for integer arithmetic is generally a losing gambit. We go into these motivations in great detail in §8.1.

For BLT, we introduce a small language for expressions which we will use as the language for where-clauses: one whose syntax for arithmetic expressions is reminiscent C and whose syntax for other datatypes is reminiscent of ML.

The core of our language is our collection of introduction forms for Ψ types, which we define in §5.1-5.10. Before we begin, though, we need to decide the relationship between the Ψ types we are defining and what we will call that type's *shadows*.

Let v be a Ψ type. Then v's shadows are the projections of said type into the programming language at hand (the first-projection) and into the expression language used in BLT (the exp- or second-projection). We refer to them as shadows because they are versions of the type v where we have *lost type information*. Our notation for these projections of a Ψ type v will follow that of physicists for first and second derivative: $\dot{v}$, $\ddot{v}$. In the first projection we lose information about endianness and about restrictions on values. Where we might have specified a type v to be a list of non-zero 16 bit little-endian integers, $\dot{v}$ would only tell us that we have a list of 16 bit integers (actually, the implementation may use any integral type large enough to hold all values). The second projection $\ddot{v}$ would tell us only that we have a list of integers.

We purposely hold the first projection somewhat abstract, as it is the means by which we bind BLT specifications to different programming languages. The exact details of the first projection are determined by the binding definition for a specific programming language.

The second projection, however, is part-and-parcel of our expression language, and we define it precisely as we describe the introduction forms for Ψ types.

As we define our Ψ type constructors–especially for sum types–we'll find it useful to refer to Benjamin Pierce's formalism of *lenses* [6]. Pierce deals with a more general problem than ours: one where a concretization may not fully represent the abstract type, but we can still use his ideas to great benefit.

We can interpret a Ψ type v as a set of two lenses $v_{\mathcal{S}} : bs \rightleftharpoons \ddot{v}$ and $v_{\mathcal{T}} : \dot{v} \rightleftharpoons \ddot{v}$ mediating between the abstract type $\ddot{v}$, and two concretizations: bit strings (of type bs) and $\dot{v}$.

A lens l is made of two functions: $l\nearrow c = a$ (the GET function) and $l\searrow a = c$ (the PUTBACK function) which mediate between the concrete and abstract representations. These may be partial functions, and we say that $f\ c = \bot$ if f is undefined at c, and that $f \sqsubseteq g$ if $f\ x = g\ x \vee f\ x = \bot$ for all x.

Pierce defines *well-behaved* lenses by essentially requiring that they form bijections:

$l\searrow(l\nearrow c) \sqsubseteq c$ GETPUT
$l\nearrow(l\searrow a) \sqsubseteq a$ PUTGET

The programming language bindings for BLT are direct embeddings of our abstract data types, so $v_{\mathcal{T}}$ trivially enforces the GETPUT and PUTGET laws. When we come to the section on `unions`, we will discuss the implications of adhering to the GETPUT and PUTGET laws for the lenses connecting $\mathcal{S}$ with $\mathcal{A}$.

For every Ψ type v, we create two functions parse_v and $\mathsf{unparse}_v$, which we can define as follows:

$$\mathsf{parse}_v \equiv v_{\mathcal{T}}\searrow \circ\, v_{\mathcal{S}}\nearrow : bs \to \dot{v}$$

$$\mathsf{unparse}_v \equiv v_{\mathcal{S}}\searrow \circ\, v_{\mathcal{T}}\nearrow : \dot{v} \to bs$$

5 Ψ Type Expressions in BLT

5.1 Base Types

We have two kinds of base types: integer and floating point. The integer types are the only interesting ones, however: no expressions in BLT may refer to floating point values.

Our integral base types are all based on unsigned or two's complement signed binary arithmetic in some number of bits n: `uint`(n), `uintle`(n), `sint`(n), and `sintle`(n).

The ultimate underlying primitive from which we extract raw binary data is a bit stream. If the runtime system instead provides low-level access in the form of a byte stream, we take the high bit of the first byte as bit 0, the high bit of the second byte as bit 8 and so on. This corresponds to reading integers from the stream as big-endian.

`uint`(8) takes the next 8 bits of the bit stream, most significant bit first, and returns them as a single value from `0..255` (decimal). Similarly, `uint`(4) returns a nybble of the next 4 bits.

`uintle`(32) takes the next 32 bits, and does a little-endian 'byte swap', putting the first 8 bits as the 'low byte', the next 8 bits as the next lowest byte, and so on. After the byte swap, the bits are interpretted as a single 32 bit unsigned value.

The argument to `uintle` must be one of (8,16,32,64). `sint`(n) and `sintle`(n) are defined similarly to `uint`(n) and `uintle`(n).

Our floating point types are based on IEEE 754 standard 32 bit and 64 bit, referred to as `float32`, `float32le`, `float64` and `float64le`.

Below is an example of how the base types might project into C:

v	$\dot{v}$ (in C)	$\ddot{v}$
`uint(8)`	`unsigned char`	`int`
`uint(12)`	`unsigned short`	`int`
`uint(32)`	`unsigned int`	`int`
`uintle(32)`	`unsigned int`	`int`
`uint(64)`	`unsigned long long`	`int`

5.2 Record Types

Though we say "record types," these are not plain record types in the classical sense. We may have an integer n as a field in our record and have a subsequent field which is a list of n `uint`(16)'s. In such a manner, the type of any field may refer to previously parsed variables, so there is a bit of implicit dependent-typing here in our notion of record.

The syntax for defining a Ψ record type is:

```
struct
  v_1 : τ_1 where ref–exp_1,
  v_2 : τ_2 where ref–exp_2,
  v_3 : τ asbits [v_{3,1} : τ_{3,1}, v_{3,2} : τ_{3,2}] where ref–exp_3
tcurts
```

Now v_i are variable names, τ_i are expressions defining a Ψ type, and *ref–exp*$_i$ are expressions which evaluate to a boolean. These expressions are evaluated in the *contexts*: Γ mapping variables to (Ψ) types and γ mapping variables to values. Γ and γ must agree in the following manner: If $\Gamma v = y$ then $\gamma v : \ddot{y}$

Let Γ_0 and γ_0 be the contexts at the beginning of the record definition. Then the refinement expression *ref–exp*$_1$ is evaluated in the context where $\gamma = \gamma_0 \bigcup \{\langle v_1, c\rangle, \langle \$!, c\rangle, \langle @!, B\rangle, \langle @.!, b\rangle\}$, where:

- c is the parsed value of type $\ddot{\tau}_1$
- B is the byte-offset into the current record
- b is the bit-offset into the current record

and $\Gamma = \Gamma_0 \bigcup \{\langle v_1, \tau_1\rangle, \langle \$!, \tau_1\rangle, \langle @!, \mathtt{u8}\rangle, \langle @.!, \mathtt{u8}\rangle\}$

τ_2 is evaluated in the same context as *ref–exp*$_1$ (with the values B and b updated and \$! removed). *ref–exp*$_2$ is evaluated in the context of τ_2 with v_2 and \$! bound to a value of type τ_2.

Now let's examine the third declaration, which is slightly different from the other two.

τ `asbits` $[v_{3,1} : \tau_{3,1},\ v_{3,2} : \tau_{3,2}]$

This declaration takes an integer type and reinterprets it as a set of bitfields. This is needed so we can accomplish little-endian re-ordering of the bit stream before slicing off bit fields. The contexts for parsing the variables $v_{3,1} : \tau_{3,1},\ v_{3,2} : \tau_{3,2}$ takes place exactly as it would have had we inlined this list into the containing strucuture, striking "$v_3 : \tau$ `asbits` [" and "] `where` *ref–exp*$_3$" entirely. The only change here is *which* bits become $v_{3,1}$ and $v_{3,2}$. As syntactic sugar, we may omit the type annotation for a bit field, in which case the type `uint(1)` is assumed.

ref–exp$_3$ is evaluated in a context where $v_3 = \$!$ and $v_{3,i}$ are all bound to values of type τ and $\tau_{3,i}$ appropriately.

We find the first shadow of a type pointwise. If we name the record type above R, then
$\dot{R}$= `struct`
$v_1 : \dot{\tau}_1,\ v_2 : \dot{\tau}_2,\ v_3 : \dot{\tau},$
$v_{3,1} : \dot{\tau}_{3,1},\ v_{3,2} : \dot{\tau}_{3,2},\ v_{3,3} : \dot{\tau}_{3,3}$
`tcurts`

There are a few more details concerning record types: recall that one of our main purposes is to keep the gritty details of low-level data layout and tagging out of our programmer's hair.

We may qualify a subset of the fields of a struct as `implicit`. This means that, given the original Ψ type and the non-`implicit` fields of the struct, we can reconstruct the original sequence of bits in full (with the exception of any `dontcare` fields).

If we can make a field `implicit`, we can forgo giving it a name completely by replacing its name with the wildcard (an underscore _). A field with a wildcard name has its type implicitly marked as `implicit`.

Any fields in a record type v marked as `implicit` will not appear in any of v's shadows. Nevertheless, it is legal to reference a wildcard field within its type refinement via the \$! binding, and it will be legal to reference an `implicit` (but not a `dontcare`) variable (1) in the parsing stage since we have just parsed it and (2) in the encoding stage since `implicit` promises that we can reconstruct the value.

XXX: Note: `implicit` *is going to be a lot easier to implement if we require the user to simply write a function to compute it from the other (non-* `implicit` *) fields.*

Finally, we may sometimes want a short-hand form for the field name v_i. In this case, we can replace v_i with v_i `aka` foo and then use foo wherever we would have used v_i.

5.3 Sum Types

`union`
$\quad NAME_1 : \tau_1$
$\quad NAME_2 : \tau_2$
$\quad NAME_3 : \tau_3$
`noinu`

Sum Types are discriminated in the parse stage similarly to ML pattern matching: that is, the order in which the clauses appears does matter. All of the branches of a union must have distinct names. The evaluation of any where clause of τ_i can expect to have values for $NAME_i$ and \$! bound as in the case of struct, but as this is pure alternation, there will be no bindings from the other $NAME_j$s.

5.4 Lists of Unknown Length

τ `list`

Here τ must use \$! if it is a refinement type. For instance the following is a legitimate definition of a C string.

`struct`
$\quad$ *str* : `(u8 where $!` $!=$ 0) `list`
$\quad$ *null* : `u8 where $!` $=$ 0
`tcurts`

Whenever a τ `list` is given in a BLT specification, if it is followed by anything, the type that follows it must be distinguishable from a value of type τ. *See the section on distinguishability*

5.5 Lists of Known Length

```
τ list(n)
```

As opposed to the previous kind of list, lists of this sort take a Psi type τ and a non-negative integer n as arguments, and parse a sequence of n objects of type τ.

5.6 Where-Clauses of Ψ type

XXX WARNING: This section is rough, as is the section on Psi-Dependent Types inso far as parametric polymorphism

We may specify a Ψ type in the context of some set of exp- and Ψ-type bindings through something which has the same syntax as a *where-clause*: that is, a where-clause whose branches all result in Ψ types. Since we are targetting a language with a static type system, each branch of the where-clause must evaluate to Ψ type that projects onto the same data type in the programming language.

A question arises then, as to what sorts of these expressions are *admissible* in our type specifications as describing valid Ψ types.

As a first example, consider the case where we have a session of the X Protocol. The first packet will determine if we send our data as big-endian or little-endian, and we will capture this piece of information as a value of type `XEndianness` as follows:

```
union
      BIG : u8 where BIG = 'B',
   LITTLE : u8 where LITTLE = 'l'
noinu
```

The Ψ type of all other packets will *depend on knowing this runtime value.* That is, all of the other X packet types will be parameterized by a value of `XEndianness`. As a first step to describing these packets, let us describe a 16 bit integer dependent a value e of type `XEndiänness`:

```
case e
   of BIG => u16/be
   | LITTLE => u16/le
```

It might also be useful to have a parametric type α `Option`, which we could use within other structures. If we restrict all type expressions so that their second projections are equal, then a fully parametric type would not be admissible. Can we resolve this conflict?

If we choose branch expressions to be admissible only if their second projections are equal, the only differences that may exist are the actual values parsed (and hence the lengths of lists and bitfields), and the integer endian-ness. Therefore expressions like this should be illegal:

```
if n < 20
then LinkedList
else HashTable
```

since the fundamental datatypes will clearly not be encoded as the same exp type. So even though it might be convenient to have such an object, it is not allowed directly. You would have to represent it instead as a tagged union.

One way to allow parametric polymorphism would be to allow expressions like that for α `Option` only if they resolved at compile time – that is, only if they were syntactic sugar.

So you could have a Psi-Dependent type constructor:

```
Directions(t :: int) = t withvalues
[NORTH = 0, EAST = 1, SOUTH = 2, WEST = 3]
```

However, this could only be used within a specification as a *a specific instantiation*:
e.g. `Directions(u8)` or `Directions(u16/le)`.

5.7 Refinement Types

τ `where` *ref–exp*

The only difference between this and the version that appeared as part of a record is that here we only have $! bound, and we don't have the other fields of the record to reference.

5.8 exp-Dependent Types

$$\texttt{tycon\_name}(\texttt{foo} : \tau) = \texttt{tyexp} \quad \text{(exp-Dependent Intro)}$$
$$\texttt{tycon\_name}(\texttt{fooval}) = \texttt{tyexp} \quad \text{(exp-Dependent Elim)}$$

`tyexp` here is any Psi type expression evaluated in a context where. The typing context for `tyexp` binds `foo` to a value of type $\ddot{\tau}$

5.9 Psi-Dependent Types

$$\texttt{tycon}(\texttt{t}) = \texttt{tyexp} \quad \text{(Psi-Dependent Intro 1)}$$
$$\texttt{tycon}(\texttt{t} :: \texttt{expty}) = \texttt{tyexp} \quad \text{(Psi-Dependent Intro 2)}$$
$$\texttt{tycon}(\texttt{psity}) \quad \text{(Psi-Dependent Elim)}$$

`tyexp` here is any Psi type expression evaluated in a context where `t` is known to be a Psi type.

`expty` demands some explanation. The idea, is, one needs to be able to express *which* Psi Types are allowable. If you are building an `Option(t)`, you can have `t` be any Psi type. However, if you are expecting to compare your object to an integer, then you should be able to say you allow only those Psi types whose exp-shadows are integers. Similarly, you can imagine copying the syntax for describing Psi types only replacing all of the base types with integer. These are the exp type descriptions.

5.10 Reinterpreted Types

```
t_1 asa t_2
```

The idea here is that t_1 does some checking of the data, and then passes off the data to be interpreted as a different type. As an example, if we know that a comment field is a certain size, and that it contains a UTF8 string, we might have the following declaration:

```
JPEGHeader =
struct
   ...
   n : u16/be,
   comment : u8 list(n-2) asa UTF8String
   ...
tcurts
```

5.11 The `dontcare` Qualifier

Any type which has no restrictions on the values parsed from the bit stream may be marked as `dontcare`. This means that any value is acceptable, and that we do not need to save the underlying data when we project it into our programming language. Any types marked as `dontcare` which appear within a record are implicitly marked as `implicit`.

5.12 Derived Forms

1. We have the following shorthand for arithmetic types:

   `u8` ≡ `uint(8)`
   `u16/be` ≡ `uint(16)`
   `u16/le` ≡ `uintle(16)`
   `u32/be` ≡ `uint(32)`
   `u32/le` ≡ `uintle(32)`
   `u64/be` ≡ `uint(64)`
   `u64/le` ≡ `uintle(64)`
   `s8` ≡ `sint(8)`
   `s16/be` ≡ `sint(16)`
   `s16/le` ≡ `sintle(16)`
   `s32/be` ≡ `sint(32)`
   `s32/le` ≡ `sintle(32)`
   `s64/be` ≡ `sint(64)`
   `s64/le` ≡ `sintle(64)`

2. τ `withvalues` $[l_1 = v_1, l_2 = v_2, l_3 = v_3]$

   This is the BLT equivalent of a C-style enum. t indicates how many bits we use to form each integer. (t's shadow must be an integer type). The l_i are labels, and the v_i are integer values. This is equivalent to the construct:

   $fresh\_tycon\_name(f :: int) =$
   `union`
   $\quad l_1 : f$ `where` $\$! \; = \; v_1$
   $\quad l_2 : f$ `where` $\$! \; = \; v_2$
   $\quad l_3 : f$ `where` $\$! \; = \; v_3$
   `noinu`

   $fresh\_tycon\_name(t)$

3. Omission of " `where` $!":

   If there is a context where we may expect a refinement clause, and instead of `where` *ref–exp*, we read an arithmetic comparison operator `op`, we assume that the user meant `where` $! `op` ....

   Hence, we can write the following:

   nz : `u8` != 0

   instead of the much more verbose

   nz : `u8` `where` $! != 0

6 The Round-Trip Problem

7 Ψ Type Names and Type Equality

Most of the time, naming a Ψ type is just short-hand for writing the type out in full. We may want to allow a form of opaque naming, so we could for instance distinguish `u32`s which are *File Handles* and `u32`s which are *Graphics Contexts*.

8 The Expression Language

Here we provide the syntax and semantics for BLT's where-clause expressions.

8.1 Why Bother?

Any one who has used a parser generator may at this point ask an obvious question: Why not simply use snippets of code from the target language whenever something interesting is parsed? One might choose this option for a number of reasons: (1) some data is in a format not usably mapped to our programming language by a Ψ type, (2) we don't need to have a full translation of the data, or (3) using the target language means we need not write a new language for normal things like arithmetic expressions.

The first reason here is sensible; BLT was designed to re-box and re-tag dependent low-level layouts. We cannot use BLT to automatically generate a pixmap from a JPEG file, for instance. What BLT *does* provide, however, is a sensible interpretation of the overall layout of a JPEG file: we present the user with a sensible representation of the sections of a JPEG file as a collection of structures: quantization and huffman code tables, image data buffers, comment buffers, and so on. Actually de-compressing the huffman-coded image data and running an inverse discrete cosine transform to get the original image is up to the developer of the image manipulation software. If the uncompressed data itself has a complicated dependently typed structure, one may well want to use a separate BLT specification to interpret the uncompressed data. Incorporating arbitrary data transforms is beyond the scope of our language, however.

The second reason is also a completely valid design goal and is why packet filters exist. Sometimes you simply need to check a few bits in a byte stream and forward that chunk of data to someone who will do something sensible with it. Full re-tagging and re-packing of the data is inefficient for such tasks.

The final reason we have offered for using the target programming language is bad for a couple of reasons: (1) Using the target language makes our specifications unnecessarily dependent upon that programming language. Suddenly you need five specifications for "The" JPEG format: one each for C++, Java, SML, Python, and Ada. (2) The models of arithmetic offered by target languages are often poorly suited for writing low-level parsers. Let us illustrate this point by means of a few examples.

```
struct
        opcode :  u8 where opcode == 0x28,
        ...
tcurts
```

Arithmetic Example 1

Here, we might venture to say that casting the phrase "*opcode* `== 0x28`" into the target language poses no ambiguity at all. There is a subtle question here, though, which is: *what type is the constant* `0x28`*?* Though it is straight-forward here to say "it's a `u8`, as that's what we're comparing it with," what about our next example?

```
struct
        opcode :  u16/be where
                      opcode < 0x10000,
        ...
tcurts
```

Arithmetic Example 2

If we use the logic from our first example here, we would cast `0x10000` to zero, making the condition always false, where it should be always true. We may want to alert the programmer that this particular example is probably a typo, since the expression is always true, but quietly truncating a large integer constant to zero is very bad behavior. A quick-and-dirty solution to this example would be to do all operations in a 32-bit machine word; but what about our next example?

```
struct
        pagesize :  u8 where
                (1 << pagesize) <= 16 * 1024,
        npages :  u8,
        data :  u8 list(npages *
                        (1 << pagesize))
tcurts
```

Arithmetic Example 3

While we could rephrase this where-clause as *pagesize* < 15, this obfuscates what we are specifying: page sizes are limited to 16 kilobytes. We want BLT specifications to be *easy* to write and *hard* to get wrong.

This example illuminates the underlying issue in our arithmetic examples: overflow and underflow. So how do C and SML handle exceptional arithmetic conditions?

C quietly ignores integer integer overflow and underflow. Further, there is a subtle ambiguity with regards to arithmetic and the shifting expression above—the results are implementation defined: the first where-clause compiled by gcc would shift by *pagesize* `mod` 32 on x86 and by *pagesize* `mod` 64 on PowerPC. If we are equating shifting to the left with multiplying by powers of two, even mod 2^{32}, both interpretations are egregiously wrong. Therefore, letting our where-clause for *pagesize* be simply an expression in C will rarely give us the result we intended.

SML/NJ throws arithmetic exceptions whenever an arithmetic expression overflows or underflows a machine word. Unfortunately, whereas this makes a straight embedding of arithmetic expressions into SML/NJ's native integer type a safer gambit than in C, SML/NJ's integers are only 31 bits wide. This will not do when evaluating expressions with 32 bit values.

Why is arithmetic overflow and underflow such a big deal? Because these issues cause *real bugs* in *real parsers*. The JPEG exploit in Microsoft's GDI+.dll, which allowed an attacker to run arbitrary code on a user's computer. exploited an integer underflow bug. In section 5.10's example of a re-interpreted type, we see exactly how parsing went wrong: to get the length of the comment field, we subtract two from an unsigned 16-bit integer n. If we don't remember to test to check that $n > 2$, we may incorrectly think the comment field very long indeed.

In BLT, we choose to solve these problems by requiring all arithmetic expressions evaluate *as though they were performed with* BIGNUMs, and by checking that all lists are of non-negative length.

8.2 Integral and Boolean Expressions

In the table below we present our arithmetic and boolean operators. The rows of the table are arranged in order of precendence, with the lower rows binding more strongly.

`< > <= >= = !=`	IntCmp
`NOT` `AND` `OR`	BoolOps
`+ -` `* / %`	IntOps
`|` `<< >>` `&`	BitwiseOps

Boolean and Integer Operations

The typing rules for these are as expected:

$$\frac{e_1 : \mathbb{Z} \quad e_2 : \mathbb{Z} \quad \texttt{op} \in \text{IntCmp}}{e_1 \;\texttt{op}\; e_2 : \mathbb{B}} \;\text{S-IntCmp}$$

$$\frac{e_1 : \mathbb{B} \quad e_2 : \mathbb{B} \quad \texttt{op} \in \text{BoolOps}}{e_1 \;\texttt{op}\; e_2 : \mathbb{B}} \;\text{S-BoolOp}$$

$$\frac{e_1 : \mathbb{Z} \quad e_2 : \mathbb{Z} \quad \texttt{op} \in \text{IntOps}}{e_1 \;\texttt{op}\; e_2 : \mathbb{Z}} \;\text{S-IntOps}$$

$$\frac{e_1 : \mathbb{Z} \quad e_2 : \mathbb{Z} \quad \texttt{op} \in \text{BitwiseOps}}{e_1 \;\texttt{op}\; e_2 : \mathbb{Z}} \;\text{S-BitOps}$$

In addition, certain arguments to these operators will cause runtime errors. The problematic values fall into two classes: arithmetic operators which do not accept negative values `n`, and arithmetic operators which do not accept `0`:

x`/0` x`%0` `n%`x x`%n` `n<<`x `n>>`x

Arithmetic Arguments Causing Runtime Errors

8.3 Comments

XXX figure this out later. Syntactic Issue.

8.4 Constants (Value Intro Forms)

8.4.1 Integer and Boolean Constants

As with all arithmetic expressions in BLT, arithmetic constants are interpreted as though they were BIGNUMs. We allow several syntactic forms for integer constants. Binary constants are prefixed with `0b`, hexadecimal with `0x`, and octal with `0`. A `0` by itself represents the number zero. Case does not matter. In addition, an underscore or period may occur at any position between numerals past the base prefix for easier readability. For instance, the following are all legal renderings of the decimal number 42: `42`, `0x2A`, `0B_010.1010`, `052`.

The boolean constants are `true` and `false`.

8.4.2 Strings and Integer Lists

Strings and lists of integers are syntactic sugar so we don't need to write out a long sequence of `u8` variables, each matching some integer.

Strings are simply lists of integers that are each at most 255: suitable for comparing against sequences of `u8`'s. XXX Figure out precise embedding later. Syntactic Issue. We at least want ASCII strings to be easy.

We also provide this form for integer list constants: `[ 5, 25, 24, 122]`.

8.5 Elimination Forms for Datatypes

- Equality of Integer Lists
- `length`(x) where x is a list
- `case` e `of FOO(`v`) =>` e_{foo} `| BAR =>` e_{bar} (Union Elimination Form)
- e_1.lab (Record Elimination Form)
- e_1.(k) (k^{th} item of a list)

XXX Do we want anything else? We don't want the complications of recursive functions, so we don't want to bother with `cons`, `car`, and `cdr`. What about iteration: `map`, `iter`, `forall`, `exists`?

8.6 Variable and Ψ Type Names

Syntax:

`([a-z][A-Z][_])([a-z][A-Z][-_/'"][0-9])+`

XXX Say something about scope. Ψ type names are shorthand. All top level?

9 Ψ Type Distinguishability

Whenever we have an α list of unknown length, it must be followed either by nothing (i.e., end of file), or it must be followed by an item of type β which is distinguishable from α.

Here we define inductively what it means for two types α and β to be distinguishable. XXX

10 What *isn't* in BLT

It is as important to note what we have *omitted* from BLT as what we have included. For instance, there are no introduction forms for datatypes more complicated than lists of integers. The only way values of such types are introduced is by their being parsed. We have also omitted recursive functions, and *XXX? indeed iteration of all sorts.*

We also have very limited support for offsets into a stream. This can be quite a limiting issue. However, we can get around the issue by parsing in stages. **XXX Insert example about reading in an ELF Header (with dictionary) with a _parse function, and then using that data to skip to other parts of the file and reading them with _parse functions**.

11 Bindings for C and SML

12 Related Work

Low-level parsing is an old problem, and there is a wealth of literature on related problems.

Data-Structure Layout Languages: Our problem is that of mediating between a standardized format and a user's programming language. In this realm, a few languages stand out: XDR [19], X.409/ASN.1 [14, ?], CSN.1 [12], Ada's sublanguage for data representation [7], Flavor [4], and DataScript [1].

XDR is general enough to provide an encoding of a reasonable variety of data structures, but is not general enough to encode alternate endian-ness, bit-packing or type dependency which is not of the form $n\mathrm{a}^n$.

Ada has a beautiful sublanguage for specifying the bit-level layout of data structures. Unfortunately, the standard balks at dependent types, stating:

> An implementation need not support representation items containing nonstatic expressions. [7] §13.1.21

Which means "If you can figure out how to support this run-time dependent typing, great. We can't."

DataScript and Flavor are both extensions to a core language shared with BLT: (dependent) records, unions, and lists (both bounded and unbounded).

Flavor takes a painfully dynamic approach, leaving no way for the user to tell if certain fields were parsed except by means of a runtime error. This is equivalent to Java's mistake of having all of its objects implicitly be object options. Flavor's type specifications look more like the code of type parsers, full of **while** and **for** loops. The consequent semantics are bizarre: if a field is declared in a loop, then each time the loop is passed the old value is discarded and the variable is rebound to a new parsed value. These constructs only make sense in the context that Flavor specifications are littered with code of the target language, as in Yacc- specifications.

As we are interested in standalone, reusable specifications which provide round-trip data conversion to statically-typed languages, Flavor simply will not do.

DataScript is much closer to our ideal language, but it has poor support for dependent endianness, requiring duplicate code for any field with variable endianness.

Neither Flavor or DataScript has a formal definition. Further, both languages rely directly upon the target programming language for evaluating their arithmetic expressions, a bad practice we've already discussed.

Our work can be seen as a clean-up of DataScript, providing it a more rigorous formalization **XXX make this true XXX**, bindings for SML and C **XXX make this true XXX**, and several small extensions: re-interpreted types, bit fields (and `asbits` bit fields), and the `implicit` and `dontcare` qualifiers.

XXX X.409/ASN.1 and CSN.1

Packet Filtering: Packet filters are snippets of code that match a network packet against a specific pattern, and route them to an application [10, 3, 11, 5, 2]. Packet filters are typically written in a very primitive language: a mix of boolean predicates over values from the data stream and integer constants and `SHIFT` instructions to accept a portion of the stream. The literature on packet filtering is aimed mainly at performance, safety, and runtime extensibility.

The goals of packet filters are fast demultiplexing, not round-trip transformation of data to a target language. Therefore, packet filters are not concerned with full type description, and none of them can adequately deal with dependently typed binary formats.

The most interesting "packet filter" from our point of view is PacketTypes of McCann and Chandra [9]. PacketTypes provides a type system strikingly similar to that of BLT for describing incoming packets. We lump PacketTypes in with the packet filtering literature simply because the authors stopped short of using their type system for anything more than recognizing packets.

PacketTypes has the interesting feature of allowing types to be arranged in a refinement hierarchy with *overlays*. For example one may specify two refinements of an `IP` packet – `UDPinIP` and `TCPinIP`. In a sense, this is isomorphic to a "lazy sum" in which the compiler keeps track of all of the refinements of an `IP` packet and invokes a handler when it determines the most refined type. McCann and Chandra go on to take advantage of this laziness by allowing the dynamic insertion of type refinements intoa running packet filter. This sort of behavior doesn't translate well to a language with a static type system such as ML, but does provide an interesting approach to the problem of backwards-compatible format versioning.

With the exception of runtime additions to the type system, BLT and PacketTypes specify roughly the same languages. PacketTypes does not account for endianness, though, and has a slightly more restrictive notation for where-clauses. Therefore, BLT can be seen as a logical extension of PacketTypes which takes care of these issues and supports round-trip conversion of data.

Marshalling or Stub Generation: Sun RPC is the canonical example of marshalling [18]. Much as with XDR, marshalling code is useful for mediating between a programming language's data types and a byte stream, but only when you don't need to specify what the byte stream looks like.

Parsing of Context Free Grammars by Bison and Friends: We've mentioned how our language describes languages that are more and less than context free: we only support a very restrictive form of Kleene closure, and we can parse the language of expressions $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n\mathtt{d}^n$. One might argue that we could use Bison's semantic actions to capture the same sorts of constraints we can capture with BLT. **XXX** Insert Example here of why this is horribly misguided, and degenerates with dependent constructs to nasty global variables and other cruft that makes the process roughly equivalent to writing the parser by hand.

Lenses and the view-update problem: Our work can be seen in terms of *lenses* as put forth by Benjamin Pierce et al.[6] in their description of tree transforms. Pierce's *Harmony* project aims to solve the *view-update* problem for data that comes with synchronizing information encoded in an XML-style labelled tree form.

In terms of lenses, our second projection can be thought of as a canonicl Abstract Type, and our Ψ types are the combinations of two lenses: one which mediates between the bit-stream representation and the second projection, and another between the abstract type and the embedding into the user's programming language.

XXX We cannot claim that our lenses are "well-behaved" since we have only a weaker form of the PutGet law known as the law for a couple of reasons. The problem derives from our definition of union:

Pattern Matching: **XXX**

Dependent and Refinement Types: **XXX**

Problems of Arithmetic: Arithmetic overflow is an old problem, and yet most languages do not handle it in a reliable way. Whether to raise an exception is up to the implementation in C, Ada, Scheme, *Ada and Scheme* encourage *but do not* require *more sensible implementationXXX Other Examples*

SML/NJ chooses to raise exceptions for underflow and overflow of native ints, and also offers infinite precision integers. *XXX Other Examples Is this in the language definition?*

SmallTalk does all numeric computation with infinite precision integers, as do Common Lisp, Nickle, *XXX Other Examples*

13 Conclusion

References

[1] Back, G. Datascript – a specification and scripting language for binary data. In *Proc. ACM Conf. on Generative Programming and Component Engineering Proceedings (GPCE 2002)* (October 2002), pp. 66–77.

[2] Bailey, M. L., Gopal, B., Pagels, M. A., and Peterson, L. L. PathFinder: A pattern-based packet classifier. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation* (Monterey, CA, November 1994), pp. 115–123.

[3] Begel, A., McCanne, S., and Graham, S. L. BPF+. In *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (Cambridge, MA, August 1990), pp. 123–133.

[4] Eleftheriadis, A., and Fang, Y. Flavor: A language for media representation. In *Proceedings of the Fifth ACM International Conference on Multimedia* (November 1997), pp. 1–9.

[5] Engler, D. R., and Kaashoek, M. F. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *ACM SIGCOMM Computer Communication Review, Conference Proceedings on Applications, technologies, architectures and protocols for computer communications* (August 1996), vol. 26.

[6] Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Long Beach, California* (2005). Extended version available as University of Pennsylvania technical report MS-CIS-03-08. Earlier version presented at the *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2004.

[7] Intermetrics. *Ada 95 Reference Manual: The Language, The Standard Libraries.* International Standards Organization.

[8] Macromedia, Inc. Macromedia Flash (SWF) file format specification, version 7.

[9] McCann, P. J., and Chandra, S. Packet Types: Abstract specification of network protocol messages. In *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (Stockholm, Sweden, August 2000), pp. 321–333.

[10] McCanne, S., and Jacobson, V. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Technical Conference Proceedings* (San Diego, CA, November 1993), pp. 39–51.

[11] Mogul, J., Rashid, R., and Accetta, M. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM on Operating Systems Principles* (November 1987), pp. 39–51.

[12] Mouly, M. *CSN.1 Specification, Version 2*, 1998.

[13] Pennebaker, W. B., and Mitchell, J. L. *JPEG Still Image Data Compression Standard*. Chapman & Hall, 1993.

[14] Pope, A. R. Encoding CCITT X.409 presentation transfer syntax. In *ACM SIGCOMM Computer Communication Review* (October 1984).

[15] Security Focus. Ethereal vulnerabilities, 2002–2005.

[16] Security Focus, Macfarlane, C., and Debaggis, N. Microsoft GDI+ library JPEG segment length integer underflow vulnerability, 2004.

[17] Security Focus, and Solar Designer. Netscape communicator JPEG-comment heap overwrite vulnerability, 2000.

[18] Srinivasan, R. RFC 1831: RPC: Remote procedure call protocol specification version 2.

[19] Srinivasan, R. RFC 1832: XDR: External data representation standard.

[20] X Consortium, Inc. *X Window System: Volume Zero, X Protocol Reference Manual for X11 Version 4, Release 6*. O'Reilly & Associates, Inc., 1995.