On the Generality and Convenience of Etypes

W. Michael Petullo and Joseph Suh
Department of Electrical Engineering and Computer Science
United States Military Academy
West Point, New York USA
mike@flyn.org, joseph.suh@usma.edu

***Abstract*—The Ethos operating system provides a number of features which aid programmers as they craft robust computer programs. One such feature of Ethos is its distributed, mandatory type system—Etypes. Etypes provides three key properties: (1) every Ethos object (e.g., a file or network connection) has a declared type, (2) Ethos forbids programs from writing ill-formed data to an object, and (3) Ethos forbids programs from reading ill-formed data from an object. In any case, programmers declare ahead of time the permitted data types, and Ethos' application of operating-system-level recognition simplifies their programs.**

This paper first investigates the generality of Etypes. Toward this end, we describe how to convert a grammar in Chomsky normal form into an Ethos type capable of expressing exactly the set of syntax trees which are valid vis-à-vis the grammar. Next, the paper addresses the convenience of Etypes. If Etypes does not make it easier to craft programs, then programmers will avoid the facilities it provides, for example by declaring string types which in fact serve to encode other types (here Etypes would check the string but not the encoded type). Finally, we present a sample distributed program for Ethos which makes use of the techniques we describe.

Keywords*—*operating systems; type systems; formal languages; recognizers

I. Introduction

The Ethos operating system provides a set of novel system calls which aims to aid programmers as they construct robust applications. One feature of Ethos is that its read and write system calls are *typed*—they operate on typed objects or typed object streams instead of bytes or byte streams. We call this subsystem Etypes. Because Etypes is implemented within the operating-system software layer, programs cannot avoid its protections, no matter which programming language specifies them. This is in contrast to input-language verifiers whose present layering leaves them discretionary (e.g., Thrift [1] and Protocol Buffers [2]). programming-language/document type systems (e.g., XDuce [3]), and type-safe operating systems which require that all programs run within a particular runtime (e.g., Singularity [4] and JX [5]).

Etypes allows programmers to define a wide range of type definitions using Etypes Notation (eNotation), and it also defines a corresponding Etypes enCoding (eCoding) which serves as the wire representation for these types. Including Etypes within the Ethos operating system—and the benefits which arise from this layering—seemed to require a unique co-design of Ethos and Etypes' declaration and encoding formats. For example, loosely-coupled development efforts require universally unique type identifiers without a central naming authority, and Etypes requires that types be bound to a description of their semantics. We depict the types and encodings supported by Etypes in Table I.

After defining a type using eNotation, an Ethos programmer uses a program called eg2source to generate programming-language-specific routines to transform programming-language types into their corresponding eCoding and vice versa. (Another tool, et2g, first transforms eNotation declarations into a machine-readable form we call a type graph.) We describe in Table II the eg2source-generated routines which we will reference in subsequent examples. Programmers make use of these routines in order to perform I/O. Etypes presently supports the C and Go programming languages.

Ethos objects—files, interprocess-communication channels, and network connections—are present as nodes in an Ethos filesystem. Each such filesystem node bears some type's Universally Unique Identifier (UUID) in addition to traditional access control information, and thus such a UUID specifies its corresponding object's type. Ethos consults an object's UUID during the course of servicing an I/O system call. If the data to be read or written is not well-formed vis-à-vis the UUID/type, then Ethos rejects the system call. Thus programs running on Ethos cannot receive ill-formed input nor produce ill-formed output. Previous work describes the details of Etypes [6], including type hashes, which serve as type UUIDs; annotations, which bind each type to its meaning; and the design decisions which forbid partial reads and writes. The same work provides a number of example programs and compares Etypes to JavaScript Object Notation (JSON) [7] and eXternal Data Representation (XDR) [8].

Two remaining questions have been: *is Etypes sufficiently general*? And, *is Etypes sufficiently convenient*? Insufficient generality would leave it impossible to write certain useful programs on Ethos. Inconvenience would encourage programmers to dilute the benefits provided by Etypes. For example, if a programmer decides to declare string types which in fact contain encodings of other types, then Ethos would recognize only the strings and do nothing to ensure the encoded values themselves were well-formed. Clearly, Ethos' mandatory enforcement—as well as the security benefits that follow from this enforcement—depend on both generality and convenience.

To demonstrate generality, we show that eNotation can represent context-free grammars. We demonstrate convenience by showing (1) that a program (bnf2etn) can automatically

Permanent ID of this document: b427d668b629bd45c2dadf228fedf952
Date: March 6, 2014.

Type	eNotation	eCoding
Integers	e byte i int$X$ u uint$X$	little-endian signed or unsigned X-bit integers, where X is 8, 16, 32, or 64
Boolean	b bool	unsigned 8-bit integer
Floats	f float32 f float64	little-endian IEEE-754
Pointer	p *T	encoding method ‖ value
Array	a [n] T	values
Tuple	t []T	length ‖ values
String	s string	length ‖ Unicode values
Dictionary	d [T]S	length ‖ key/value pairs
Structure	n struct {...}	field values
Tagged union	m union {...}	uint64 union tag* ‖ value
Any	y Any	type's UUID ‖ value
RPC	F(T$_0$,T$_1$,...,T$_n$)	uint64 func. ID ‖ arguments

*Although eCoding encodes a union's tag as an unsigned integer, the decoding process throws an error if the integer is greater than or equal to the number of fields in the union. Correspondingly, the encoding process will not generate an invalid tag.

TABLE I: Primitive, vector, composite, and RPC type eNotation and eCoding; UUIDs are encoded as arrays of bytes, lengths (n) are encoded as 32-bit integers, T represents an arbitrary type, and ‖ represents concatenation

Code fragment	Description
TypeX x	Declare variable x as type TypeX.
x = mkTypeX(...)	Construct an object of type TypeX.
enc, dec = en.Ipc(host, RpcY)	Connect to host using a channel of type RpcY.
enc.RpcYFunc$_1$(...)	Invoke RPC RpcYFunc$_1$.
enc, dec = en.Import(RpcY)	Accept a request to create a channel of type RpcY.
dec.RpcYHandle(enc)	Receive, dispatch, and respond to an RPC within the set RpcY.

TABLE II: Sample use of routines and programming-language types which eg2source would generate from two eNotation declarations: TypeX (a non-RPC type) and RpcY (a set of RPCs containing Func$_1$, Func$_2$, ..., Func$_n$); routines and data structures generated by eg2source are underlined

derive eNotation-defined representations from Backus–Naur Form (BNF), (2) that other programs (et2g and eg2source) can generate routines and data structures which generate and process these representations, and (3) that similarities exist between existing programming environments and the use of these routines and data structures.

We next describe related work in §II. Following this, we describe the generality and convenience of Etypes in §III and §IV, respectively. Finally, we draw some conclusions and propose future work in §V.

II. Related work

The language-theoretic security approach espouses two principals: (1) input languages should grant minimal computational power and (2) the components that make up a computer system must transform between data structures and streams of bytes in an equivalent way [9]. A consequence of the context-free equivalence problem is that the latter must be addressed by formally specifying input languages and machine-deriving transformation routines (Etypes facilitates this across all programs). We use the former as a foothold as we describe why Etypes is sufficiently general.

The Chomsky hierarchy of languages organizes language classes by the level of computational power necessary to recognize them [10]. From the lowest to highest computational power, the hierarchy consists of the regular languages, the context-free languages, the context-sensitive languages, and the unrestricted languages. It is worth noting that context-free languages themselves can be divided into those that are unambiguous (deterministic) and those that are ambiguous (nondeterministic). Recognizers for the four levels of the hierarchy require finite-state machines, push-down automata, linear-bounded automata, and Turing machines, respectively.

The computational power of context-free grammars appears powerful enough for most computer input languages. For example, the grammar for HTML4 is context-free, as are the grammars for arithmetic expressions, many database languages, and many general-purpose programming languages. Accordingly, this paper focuses on the relationship between Etypes and context-free grammars.

Network protocol grammars often have components which are context-sensitive [11]. This appears accidental in some cases; we later show that the use of Etypes sometimes reduces the need for such computational power (e.g., as with replacing HTTP chunked encoding). Etypes also provides features similar to context sensitivity for use when such power really is necessary. In other cases—in particular, with network- and transport-layer protocols—it is likely that context sensitivity will remain for some time. Yet these protocols are processed primarily by system software, and more specialized techniques such as PacketTypes [12] can aid in writing these parsers, whether in system software or in specialized tools such as packet capture and analysis software.

Context-free grammars are generally specified as a set of productions, where each production consists of a single variable along with the string of variables and terminal characters it can produce. BNF defines one way to describe such productions. A particularly restricted means of specifying context-free grammars is Chomsky Normal Form (CNF) [13, Definition 8]. In CNF, every production takes one of three forms:

$$\langle S \rangle ::= \epsilon,$$

$$\langle A \rangle ::= \langle B \rangle \langle C \rangle, \text{ or}$$

$$\langle A \rangle ::= \text{a},$$

where $\langle S \rangle$ represents the start variable; $\langle A \rangle$, $\langle B \rangle$, and $\langle C \rangle$ represent distinct variables; and 'a' represents a terminal. CNF further forbids $\langle B \rangle$ and $\langle C \rangle$ from serving as the start variable.

There exist a number of algorithms which transform any context-free grammar into to an equivalent grammar in CNF. Thus we are free to compare Etypes to CNF as a means of more generally comparing Etypes to all context-free grammars; this is beneficial because of the simple nature of CNF. One disadvantage of CNF is that it increases the size of a grammar.

If a grammar's size is defined as:

$$|G| := \sum_{A \in N} \sum_{A \to \alpha} |A\alpha|,$$

where N is the set of the grammar's variables, then the most space-efficient of the known to-CNF algorithms produces grammars of size $O(|G|^2)$ [14, Table 3].

There are a number of programming interfaces which aid in producing statements that satisfy a particular grammar. Existing interfaces for HTML include Python's html module [15] and Java's renderSnake package [16]. Such interfaces make available a number of expressions which produce objects representing various language constructs. An alternative is to build statements using templates, as with Go's template package [17], PHP [18], and Java's JSP [19]. Our system bears some resemblance to the former technique.

⟨*Expr*⟩ ::= uint64
  | ⟨*Term*⟩, ⟨*MulOp_Factor*⟩
  | ⟨*Expr*⟩, ⟨*AddOp_Term*⟩

⟨*Term*⟩ ::= uint64
  | ⟨*Term*⟩, ⟨*MulOp_Factor*⟩

⟨*Factor*⟩ ::= '`uint64`'

⟨*AddOp_Term*⟩ ::= ⟨*AddOp*⟩, ⟨*Term*⟩

⟨*MulOp_Factor*⟩ ::= ⟨*MulOp*⟩, ⟨*Factor*⟩

⟨*MulOp*⟩ ::= '`bool`'

⟨*AddOp*⟩ ::= '`bool`'

Fig. 1: Grammar G in BNF

III. The generality of Etypes

Before we describe what Etypes can express, we must admit what it cannot. We did not design Etypes to directly represent string-based to-be-parsed languages such as HTML. Lexical analysis and parsing code has historically been the source of many bugs, including general bugs such as buffer overflows [20, 21, 22], injection vulnerabilities [23, 24], Dalí/chameleon vulnerabilities [25, 26], and others [9, 27].

To avoid these bugs, Etypes forgoes string-based language representations and instead makes use of an encoding format called eCoding, along with the data description language eNotation. Programmers who use Etypes also benefit from a more straightforward mapping between programming-language types and encoded data than what is found with string-based language representations. Furthermore, the conciseness of eNotation/eCoding permits us to machine-derive recognizers for programmer-defined types. A previous publication describes this in more detail [6].

Here we focus on the use of Etypes to define a set of types sufficient to represent the syntax trees which enumerate a context-free language. Instead of using HTML and similar string-based languages, Etypes-based programs communicate using eCoding messages which contain these syntax trees (or simpler types, when context-free input is not necessary). This means that a program which currently uses HTML would have to be rewritten to instead make use of eCoding syntax trees. The benefit of this work is that Ethos can then categorically forbid ill-formed input and output, even though programmers can continuously introduce new types to a running system.

```
Expr union {
  uint64_0 uint64
  term_MulOp_Factor_1 *Term_MulOp_Factor
  expr_AddOp_Term_2 *Expr_AddOp_Term
}

Term_MulOp_Factor struct {
  term_0 *Term
  mulOp_Factor_1 *MulOp_Factor
}

Expr_AddOp_Term struct {
  expr_0 *Expr
  addOp_Term_1 *AddOp_Term
}

Term union {
  uint64_0 uint64
  term_MulOp_Factor_1 *Term_MulOp_Factor
}

AddOp_Term struct {
  addOp_0 *AddOp
  term_1 *Term
}

MulOp_Factor struct {
  mulOp_0 *MulOp
  factor_1 *Factor
}

Factor uint64

MulOp bool

AddOp bool
```

Fig. 2: Grammar G in eNotation

A BNF-to-eNotation compiler: To machine-derive eNotation from BNF, we wrote a compiler which we call bnf2etn. bnf2etn takes as input a grammar written using BNF in a relaxed variant of CNF and produces as output a series of eNotation type definitions. The eNotation produced describes a syntax tree sufficient to represent the statements which are valid vis-à-vis the input BNF.

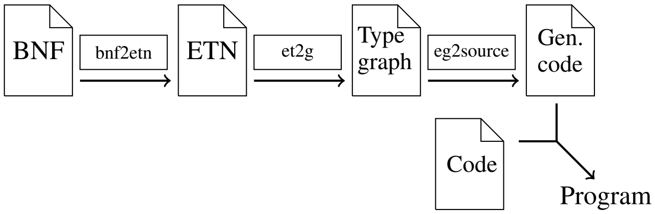

Fig. 3: Programming with bnf2etn; a programmer combines hand-written code with encoding and decoding routines which bnf2etn generates

After lexically analyzing and parsing a BNF file, bnf2etn applies the following rules to produce the eNotation type definitions:

Rule 1 If an identifier i appears on the left side of two or more productions, then generate a tagged union which provides one field for the right side of each production. Set the type of each field to represent its corresponding right side. Name the tagged union i.

Rule 2 If an identifier i appears on the left side of only one production, then define i as a type which is equivalent to the type of the right side.

Rule 3 If two or more variables exist on the right side of a single production, then generate a struct s which provides one field for each right-side identifier. Set the type of each field to represent its corresponding identifier. Name the struct by combining the names of each of s's fields.

Note that although CNF has the disadvantages we described in §II, CNF is no less expressive than BNF. Furthermore, one could write a compiler that accepted other forms, albeit with more complex transformation rules. It is also worth mentioning that while our automatic tool demonstrates generality, it does not preclude human-written types. In some cases, human-written eNotation will likely remain more concise and convenient to reason about than machine-generated eNotation for some time, much as was the case with programming-language compilers until the science of optimizing programming-language compilers was better understood.

Once bnf2etn produces eNotation, a programmer can use et2g and eg2source to generate routines which process the types described by the eNotation [6]. Assuming that bnf2etn places its output in a file called *types.t*, then executing et2g *types.t* will produce a type graph which describes the types in a machine-readable form. Executing eg2source *types* generates a programming-language type corresponding to each eNotation type defined in *types.t*, along with the routines which transform these programming-language types to eCoding and vice versa. We summarize the process in Figure 3. §IV provides an example which better describes this generated code.

A grammar example: Consider the grammar G for arithmetic expressions which we depict in Figures 1–2. Figure 1 describes G using BNF in CNF, and Figure 2 depicts the corresponding eNotation as produced by bnf2etn. In this case, bnf2etn lexically analyzes and parses the description of G which we depict in Figure 1, produces a syntax tree which represents G itself, and applies its rules to the syntax tree to produce the output in Figure 2 (which itself can represent yet other syntax trees, these corresponding to the strings in G). Of course, any context-free grammar described using this notation could be processed in the same way.

Lines 1–5 Application of Rule 1 to the variable ⟨*Expr*⟩.

Lines 7–10 Application of Rule 3 to the variables ⟨*Term*⟩ and ⟨*MulOp_Factor*⟩.

Lines 12–15 Application of Rule 3 to the variables ⟨*Expr*⟩ and ⟨*AddOp_Term*⟩.

Lines 17–20 Application of Rule 1 to the variable ⟨*Term*⟩.

Lines 22–25 Application of Rule 3 to the variables ⟨*AddOp*⟩ and ⟨*Term*⟩.

Lines 27–30 Application of Rule 3 to the variables ⟨*MulOp*⟩ and ⟨*Factor*⟩.

Line 32 Application of Rule 2 to the variable ⟨*Factor*⟩.

Line 34 Application of Rule 2 to the variable ⟨*MulOp*⟩.

Line 36 Application of Rule 2 to the variable ⟨*AddOp*⟩.

There are three characteristics of G and the corresponding eNotation which merit further explanation. First, syntax tree validity is equivalent to type system validity. Conversely, Etypes rejects invalid syntax trees and thus no process can receive one from or produce one for another process.

Second, what at first appear as terminals (i.e., '`uint64`' and '`bool`' in G), are, in fact, a reference to some type already known to Etypes. True terminals are of less use here, since they often merely serve to allow the encoding of syntax trees as strings, as with C's '`while`', '`;`', or '`{`'. Since Etypes instead makes direct use of syntax trees, the information which C conveys with keywords such as '`while`' is in Etypes represented by the tag present in a union's encoding. While a grammar for C might define the variable ⟨*Integer*⟩ using a regular expression made up of the terminals '`0`', '`1`', . . . , '`9`', Etypes is free to simply use the existing type '`uint64`'.

Finally, the use of terminals to reference known Etypes adds some degree of expressiveness to our use of context-free grammars. Many network protocols contain portions that are not context-free, such as fields which are preceded by their length [11]. (As with HTML, we are not concerned with encoding existing network protocols using Etypes, but rather with showing Etypes is expressive enough to specify replacement protocols.) Since Etypes provides arrays (vectors of a fixed length) and tuples (vectors of arbitrary lengths), terminals defined as these types provide features resembling some of those found in context-sensitive languages.

In some cases, context-sensitivity is accidental and becomes unnecessary due to the way Etypes integrates with Ethos. For example, Etypes objects bear a type UUID. This more robustly provides file descriptions than HTTP, and it replaces the use of MIME-type deliminator fields, a context-sensitive feature. Likewise, Etypes replaces HTTP's chunked encoding with encodings where each "chunk" is a valid type with a self-evident length, albeit only one piece of a larger valid type such as a tuple, array, or dictionary. Properly handling chunked encoding as found in HTTP has been the source of security vulnerabilities [28, 29].

Blinded data: Through its use of Etypes, Ethos applies grammar recognition before any program receives data as input and before any generated output can be consumed by another program. For this to be efficacious, programs ought to avoid blinding their outputs. For example, an encrypted or compressed output would not appear in its expected form, and thus might be rejected by Etypes' recognizer. Encryption follows from the semantics of Ethos' system calls. Since the operating system performs encryption and decryption, the operating system remains able to recognize data while it is still in its cleartext form. Other benefits arise from this design [30], and a similar technique could provide compression services.

Specialized type systems: There exist many specialized type systems which provide properties absent from Etypes. These include linear types [31], which ensure at most one reference to each object; dependent types [32], where values

```
void main() {
  Encoder enc, Decoder dec = en.Ipc(hostname, CalcServer)

  // Build expression tree.
  MulOp_Factor opFactor = mkMulOp_Factor(true, 7)
  Term_MulOp_Factor termOpFactor = mkTerm_MulOp_Factor(9, opFactor)
  Term term = mkTermTerm_MulOp_Factor(termOpFactor)
  AddOp_Term opTerm = mkAddOp_Term(true, term.term_MulOp_Factor_1)
  Expr_AddOp_Term exprOpTerm = mkExpr_AddOp_Term(5, opTerm)
  Expr expr = mkExprExprOpTerm(exprOpTerm)

  enc.CalcExpr(expr.expr_AddOp_Term_2)
  dec.CalcHandle(enc)  // Receive response.
}
```

Fig. 4: Calculator client in pseudo code with the routines and data structures generated by eg2source underlined

```
void main() {
  Encoder enc, Decoder dec = en.Import(CalcServer)
  forever {
    dec.CalcHandle(enc) // Handle receiving RPC.
  }
}

void rpcCalcExpr(Encoder enc, uint64 callID, Expr expression) {
  uint64 result = calcExpr(expression)
  enc.CalcExprReply(result)
}

uint64 calcExpr(Expr expr)
  uint64 result
  switch typeof(expr) {
    case uint64:
      result = expr.data
    case Term_MulOp_Factor:
      Term_MulOp_Factor tf = (expr.data.term_MulOp_Factor_1)
      if (tf.mulOp_Factor_1.mulOp_0) {
        result = calcTerm(tf.term_0) × calcMulOpFactor(tf.mulOp_Factor_1)
      } else {
        result = calcTerm(tf.term_0) ÷ calcMulOpFactor(tf.mulOp_Factor_1)
      }
    case Expr_AddOp_Term:
      Expr_AddOp_Term et = (expr.data.expr_AddOp_Term_2)
      if (et.addOp_Term_1.addOp_0) {
        result = calcExpr(et.expr_0) + calcAddOpTerm(et.addOp_Term_1)
      } else {
        result = calcExpr(et.expr_0) − calcAddOpTerm(et.addOp_Term_1)
      }
    default:
      panic("Ill−formed") // Ethos prevents.
  }

  return result
}
```

Fig. 5: Calculator server in pseudo code with the routines and data structures generated by eg2source underlined; some routines, such as calcMulOpFactor, are not shown

influence types; and union types [33], which permit only operations valid for two or more types. We designed Etypes to be general and to tightly integrate with many programming languages. Thus Etypes is less exotic; for example, an Etypes graph built using pointers might contain cycles or aliases, and neither Etypes nor many programming languages' type systems provide a way to prevent this. Etypes leaves such checks to program logic and more exotic languages at the upper layers of the software stack.

IV. The Convenience of Etypes

Of course, programmers could merely declare that their programs produce and consume strings, which in turn could represent arbitrary data. This would cause Ethos programs to devolve back into resembling many of today's existing programs: Ethos could verify the validity of these strings, but would be unable to check the encoded data they bear. Thus it is important that Etypes provide mechanisms which are easier to use than encoding to and parsing strings.

Indeed, Etypes does reduce the number of lines of code required by input handling routines. Previous work described the tens-of-thousands of lines of parsing code present in many existing applications [6]. Generating output under Etypes requires no more effort than existing routines for producing outputs such as HTML.

A client example: Figure 4 lists pseudo code which defines a program capable of evaluating an arithmetic expression by making use of an RPC service. The types and routines that were generated by bnf2etn/et2g/eg2source are underlined. Here Etypes would ensure that the encoding that represents an RPC call—which takes the form of a method ID, call ID, and ⟨*Expr*⟩—is well-formed before sending the encoded RPC call across a network connection.

Line 38 establishes a connection to the RPC server at hostname which implements the CalcServer interface. Lines 41–46 build the arithmetic expression "$5 + 9 \times 7$". We note that the manner in which the program builds its encoded expression using the mk routines resembles Python's html module, etc. The difference is that eg2source generates the mk routines from eNotation descriptions. Line 48 invokes the calculate expression RPC on the server, and Line 49 receives the result from the server.

A server example: Figure 5 lists pseudo code which defines an RPC server capable of evaluating arithmetic expressions. Etypes ensures the server receives only well-formed data.

Line 52 receives a connection to an RPC client, and Line 54 receives and dispatches RPC requests. Lines 58–61 implement the skeleton function rpcCalcExpr, which the RPC dispatching function calcHandle calls upon receiving a CalcExpr request from the client. rpcCalcExpr calls calcExpr, listed on Line 48.

calcExpr contains two nested conditionals (the second conditional is not necessary if expr.type is uint64). The first identifies the type of expression received (recall that the Expr type is a tagged union) and acts accordingly. The second conditional inspects the operator boolean to determine whether the expression makes use of $+/\times$ or $-/\div$.

Not shown is the code for calcTerm, calcMulOpFactor, and calcAddOpFactor, but these functions resemble calcExpr. calcExpr and the other functions calcExpr invokes to evaluate an expression resemble the code that would otherwise be necessary in an expression compiler, albeit without the lexical analysis and parsing steps.

A relaxation of CNF and an HTML-like grammar: While the grammar in Figure 1 is in CNF, bnf2etn's normal form actually permits more natural productions. The rule which in CNF allows productions of the form:

$$\langle A \rangle ::= \langle B \rangle \langle C \rangle,$$

in bnf2etn's normal form is less strict, namely:

$$\langle X_0 \rangle ::= \langle X_1 \rangle, \ldots, \langle X_n \rangle,$$

where each $\langle X_i \rangle$ is a variable and there is no requirement that each i be unique (i.e., here a variable can appear more than once on the right side, and the left-side variable can also appear on the right side.)

The grammar depicted in Figure 6 makes use of bnf2etn's normal form to define a markup language named HYPER in a straightforward way. This makes it easier to reason about the grammar than if it had been specified using strict CNF. We present the corresponding eNotation in Figure 7, and Figures 8–9 depict pseudo code which generates and prints a HYPER ordered list.

⟨*Document*⟩ ::= ⟨*Head*⟩, ⟨*Body*⟩

⟨*Head*⟩ ::= ⟨*Title*⟩

⟨*Title*⟩ ::= '`string`'

⟨*Body*⟩ ::= ⟨*Markup*⟩, ⟨*Markup*⟩
  | ⟨*Markup*⟩

⟨*Markup*⟩ ::= '`string`'
  | ⟨*Emph*⟩
  | ⟨*Hyper*⟩
  | ⟨*OrdList*⟩
  | ⟨*List*⟩
  | ⟨*Markup*⟩, ⟨*Markup*⟩

⟨*Emph*⟩ ::= '`string`'

⟨*Hyper*⟩ ::= ⟨*Reference*⟩, ⟨*Label*⟩

⟨*Reference*⟩ ::= '`string`'

⟨*Label*⟩ ::= '`string`'

⟨*OrdList*⟩ ::= ⟨*OrdListPair*⟩

⟨*OrdListPair*⟩ ::= ⟨*OrdListItem*⟩, ⟨*OrdListItem*⟩

⟨*OrdListItem*⟩ ::= '`string`'

⟨*OrdListItem*⟩ ::= ⟨*OrdListPair*⟩

⟨*List*⟩ ::= ⟨*ListPair*⟩

⟨*ListPair*⟩ ::= ⟨*ListItem*⟩, ⟨*ListItem*⟩

⟨*ListItem*⟩ ::= '`string`'

⟨*ListItem*⟩ ::= ⟨*ListPair*⟩

Fig. 6: Grammar HYPER in BNF

We depict in Figure 10 a code fragment in Java/renderSnake which generates an ordered list. There is little difference in programming complexity between this code and the Etypes code in Figure 8. This should come as no surprise, because we already observed that eg2source-generated routines resemble existing encoding routines in form.

At first glance, the code in Figure 9 might appear more complicated, especially when one considers that such code must be implemented for each type (i.e., ⟨*Head*⟩, ⟨*Title*⟩, ⟨*Body*⟩, etc.). However, this is always the case when processing syntax trees. Indeed, eNotation helps because it removes the requirement for lexical-analysis and parsing.

A final consideration is the requirement for defining types in eNotation, separate from work in a traditional programming language. This might appear inconvenient at first, yet such specifications are unavoidable in loosely-coupled distributed systems. For example, RFCs are examples of these types of specifications, albeit in less formal form. Thus eNotation helps programmers specify communication formats in a formal, but programming-language independent manner. The authors of

```
Document Head_Body

Head_Body struct {
  Head_0 *Head
  Body_1 Body
}

Head Title
Title string

Body union {
  Markup_Markup_0 *Markup_Markup
  Markup_1 Markup
}

Markup union {
  str_0 string
  Emph_1 *Emph
  Hyper_2 *Hyper
  OrdList_3 *OrdList
  List_4 *List
  Markup_Markup_5 *Markup_Markup
}

Markup_Markup struct {
  Markup_0 Markup
  Markup_1 Markup
}

Emph string
Hyper Reference_Label

Reference_Label struct {
  Reference_0 *Reference
  Label_1 *Label
}

Reference string
Label string
OrdList OrdListPair
OrdListPair OrdListItem_OrdListItem

OrdListItem_OrdListItem struct {
  OrdListItem_0 OrdListItem
  OrdListItem_1 OrdListItem
}

OrdListItem union {
  str_0 string
  OrdListPair_1 *OrdListPair
}

List ListPair
ListPair ListItem_ListItem

ListItem_ListItem struct {
  ListItem_0 ListItem
  ListItem_1 ListItem
}

ListItem union {
  str_0 string
  ListPair_1 *ListPair
}
```

Fig. 7: Grammar HYPER in eNotation

```
OrdList ol = mkOrdList("One",
  mkOrdListPair("Two",
    mkOrdListPair("Three",
      mkOrdListPair("Four", "Five"))))
```

Fig. 8: Pseudo code to generate a HYPER ordered list

```
void printOL(OrdList ol, int *i) {
  switch typeof(ol.OrdListItem_0) {
    case string:
      print(*i + ". " + ol.OrdListItem_0.str_0)
    case OrdListPair:
      printOL(ol.OrdListItem_0.OrdListPair_1, i)
    default:
      panic("Ill-formed") // Ethos prevents.
  }
  *i += 1
  switch typeof(ol.OrdListItem_1) {
    case string:
      print(*i + ". " + ol.OrdListItem_1.str_0)
    case OrdListPair:
      printOL(ol.OrdListItem_1.OrdListPair_1, i)
    default:
      panic("Ill-formed") // Ethos prevents.
  }
}
```

Fig. 9: Pseudo code to print a HYPER-encoded ordered list

```
HtmlCanvas html = new HtmlCanvas();
html
  .ol()
    .li().content("One")
    .li().content("Two")
    .li().content("Three")
    .li().content("Four")
    .li().content("Five")
  ._ol();
```

Fig. 10: Java/renderSnake code to generate an HTML ordered list

PACKETTYPES [12] made a similar observation.

V. CONCLUSION

bnf2etn shows that eNotation can specify types with an expressiveness that is equivalent to context-free grammars. Context-free grammars, in turn, are sufficient for expressing constraints on most computer input. eNotation also provides some context-sensitive features, and Etypes addresses without context-sensitivity certain requirements that existing network grammars address using context-sensitive grammars (e.g., HTTP's MIME-type deliminator fields and chunked encoding). Thus eNotation is sufficiently general to describe all permitted user-space inputs and outputs, and Etypes is sufficiently general that it can perform type checking on all user-space input and output. Ethos applications cannot receive ill-formed input nor produce ill-formed output.

bnf2etn also shows that eNotation remains convenient to use, even when handling sophisticated types. This is necessary because otherwise programmers might declare their programs to consume and produce strings, even though the strings themselves contain the encodings of other types. First, the use of eNotation allows programmers to avoid writing the lexical analyzers and parsers which would convert strings to their encoded type and vice versa. Second, bnf2etn produces functions which generate the various types necessary to express the constructs described by its BNF input. Finally, eNotation serves to formally document communication formats for loosely-coupled distributed systems much like PACKETTYPES and similar systems. Thus it is less work to use Etypes properly than it is to abuse it.

Future work on bnf2etn includes further relaxing the requirement that its input conform to CNF. Yet more permissive forms of BNF might reduce the number of productions in input grammars, simplify the types that bnf2etn generates, and produce types which more closely resemble the form expressed by the original grammar.

Perhaps a larger task involves investigating the design of a new programming language which would unify the definition of programming-language types with the production of Etypes type graphs. It appears that removing the need for eNotation definitions outside of a program's programming-language specification would further improve convenience, even while continuing to generate the type graphs necessary for integration with Ethos and other languages.

Acknowledgments

The authors would like to thank all of the researchers who have contributed to Jon Solworth's Ethos project, in particular Wenyuan Fei and Pat Gavlin for their work on Etypes. Kyle Moses also provided a great deal of support, mainly in the form of patience as we discussed various pieces of this paper. The comments from our anonymous reviewers also greatly helped us revise our initial drafts.

References

[1] "Apache thrift," https://thrift.apache.org/ [Accessed Feb 19, 2015].

[2] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with Sawzall," *Scientific Programming*, vol. 13, no. 4, pp. 277–298, Oct. 2005.

[3] H. Hosoya and B. C. Pierce, "XDuce: A statically typed XML processing language," *ACM Trans. Internet Technol.*, vol. 3, no. 2, pp. 117–148, May 2003.

[4] G. C. Hunt and J. R. Larus, "Singularity: rethinking the software stack," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 37–49, 2007.

[5] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder, "The JX operating system," in *Proc. of the USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2002, pp. 45–58.

[6] W. M. Petullo, J. A. Solworth, W. Fei, and P. Gavlin, "Ethos' deeply integrated distributed types," in *Proceedings of the 2014 IEEE Security and Privacy Workshops*. New York, NY, USA: IEEE, May 2014.

[7] D. Crockford, "RFC 4627: The application/json media type for JavaScript Object Notation (JSON)," Jul. 2006, status: INFORMATIONAL.

[8] M. Eisler, "RFC 4506: XDR: External data representation standard," May 2006, status: INFORMATIONAL.

[9] L. Sassaman, M. L. Patterson, S. Bratus, and A. Shubina, "The halting problems of network stack insecurity," *;login: the USENIX Association newsletter*, vol. 36, no. 6, pp. 22–32, Dec. 2011.

[10] N. Chomsky, "Three models for the description of language," *IRE Transactions on Information Theory*, vol. 2, pp. 113–124, 1956.

[11] D. Davidson, R. Smith, N. Doyle, and S. Jha, "Protocol normalization using attribute grammars," in *Computer Security–ESORICS 2009*. Springer, 2009, pp. 216–231.

[12] P. J. McCann and S. Chandra, "Packet types: Abstract specification of network protocol messages," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. New York, NY, USA: ACM, 2000, pp. 321–333.

[13] N. Chomsky, "On certain formal properties of grammars," *Information and Control*, vol. 2, no. 2, pp. 137–167, June 1959.

[14] M. Lange and H. Leiß, "To CNF or not to CNF? An efficient yet presentable version of the CYK algorithm." *Informatica Didactica*, vol. 8, 2009.

[15] "Python html package," https://pypi.python.org/pypi/html/ [Accessed Dec 23, 2014].

[16] "renderSnake," http://rendersnake.org/ [Accessed Dec 23, 2014].

[17] "Go html/template package," http://golang.org/pkg/html/template/ [Accessed Dec 23, 2014].

[18] "PHP: Hypertext Preprocessor," http://www.php.net/ [Accessed Dec 23, 2014].

[19] "JavaServer pages technology," http://www.oracle.com/technetwork/java/javaee/jsp/index.html/ [Accessed Dec 23, 2014].

[20] "CVE-2011-3908," http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3908 [Accessed Feb 24, 2015], December 2011, US National Vulnerability Database.

[21] "CVE-2011-3906," http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3906 [Accessed Feb 24, 2015], December 2011, US National Vulnerability Database.

[22] "CVE-2011-3025," http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3025 [Accessed Feb 24, 2015], February 2012, US National Vulnerability Database.

[23] R. J. Hansen and M. L. Patterson, "Guns and butter: Towards formal axioms of input validation," 2005.

[24] Z. Su and G. Wassermann, "The essence of command injection attacks in Web applications," *ACM SIGPLAN Notices*, vol. 41, no. 1, pp. 372–382, Jan. 2006.

[25] F. Buccafurri, G. Caminiti, and G. Lax, "Fortifying the Dalí attack on digital signature," in *Proceedings of the 2nd International Conference on Security of Information and Networks*. New York, NY, USA: ACM, 2009, pp. 278–287.

[26] S. Jana and V. Shmatikov, "Abusing file processing in malware detectors for fun and profit," in *Proc. IEEE Symp. Security and Privacy*, San Francisco, CA, May 2012.

[27] L. Sassaman, M. L. Patterson, S. Bratus, and M. E. Locasto, "Security applications of formal language theory," *IEEE Systems Journal*, vol. 7, no. 3, pp. 489–500, 2013.

[28] "CVE-2002-0392," http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2002-0392 [Accessed Feb 24, 2015], April 2002, US National Vulnerability Database.

[29] "CVE-2013-2028," http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-2028 [Accessed Feb 24, 2015], February 2013, US National Vulnerability Database.

[30] W. M. Petullo and J. A. Solworth, "Simple-to-use, secure-by-design networking in Ethos," in *Proceedings of the Sixth European Workshop on System Security*. New York, NY, USA: ACM, Apr. 2013, https://www.ethos-os.org/papers/.

[31] P. Wadler, "Linear types can change the world!" in *IFIP TC 2 Working Conference on Programming Concepts and Methods*, M. Broy and C. Jones, Eds. Sea of Galilee, Israel: North Holland, 1990, pp. 347–359.

[32] P. Martin-Löf, "Intuitionistic type theory," 1984.

[33] A. Igarashi and H. Nagira, "Union types for object-oriented programming," in *Proceedings of the 2006 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2006, pp. 1435–1441.