Bidirectional Grammars for Machine-Code Decoding and Encoding

Gang Tan · Greg Morrisett

Abstract Binary analysis, which analyzes machine code, requires a decoder for converting bits into abstract syntax of machine instructions. Binary rewriting requires an encoder for converting instructions to bits. We propose a domain-specific language that enables the specification of both decoding and encoding in a single bidirectional grammar. With dependent types, a bigrammar enables the extraction of an executable decoder and encoder as well as a correctness proof showing their consistency. The bigrammar DSL is embedded in Coq with machine-checked proofs. We have used the bigrammar DSL to specify the decoding and encoding of subsets of the x86-32 and MIPS instruction sets. We have also extracted an executable decoder and encoder from the x86 bigrammar with competitive performance.

1 Introduction

Much recent research has been devoted to binary analysis, which performs static or dynamic analysis on machine code for purposes such as malware detection [3,8], vulnerability identification [6,25,27], and safety verification [31, 32]. As a prominent example, Google's Native Client (NaCl [32]) statically

Gang Tan
Dept. of Computer Science & Engineering, The Pennsylvania State University, University Park, PA 16802, USA
Tel.: +1-814-8657364
E-mail: gtan@cse.psu.edu

Greg Morrisett
Computing and Information Science, Cornell University, Ithaca, NY 14853
Tel.: +1-607-2559188
E-mail: jgm19@cornell.edu

checks whether a piece of machine code respects a browser sandbox security policy, which prevents buggy or malicious machine code from corrupting the Chrome browser's state, leaking information[1], or directly accessing system resources.

When analyzing machine code, a binary analysis has to start with a disassembly step, which requires the decoding of bits into abstract syntax of machine instructions. For some architectures, decoding is relatively trivial. But for an architecture as rich as the x86, building a decoder is incredibly difficult, as it has thousands of unique instructions, with variable lengths, variable numbers of operands, and a large selection of addressing modes, all of which can be prefixed with a number of different byte sequences that change the semantics. Flipping just one bit leads to a totally different instruction, and can invalidate the rest of binary analysis. For example, Kroll and Dean [18] found a number of bugs in the decoder of PittSFIeld [19], a tool for sandboxing x86 machine code; an attacker could potentially leverage these decoder bugs to bypass the sandboxing defense.

In our previous work [21], we developed a Domain-Specific Language (DSL) for constructing high-fidelity machine-code decoders. The DSL allows the specification of a machine-code decoder in a declarative grammar. The specification process in the DSL is user friendly in that a user can take the decoding tables from an architecture manual and use them to directly construct patterns in the decoder DSL. Furthermore, the decoder DSL comes with a denotational and operational semantics, and a proof of adequacy for the two semantics. The semantics allows us to reason about grammars written in the decoder DSL. Finally, we can automatically generate efficient recognizers and parsers from grammars in the DSL, with a proof of correctness about the generation process based on the semantics. The generated recognizers and parsers are executable, allowing testing against other implementations.

The inverse of machine-code decoding is encoding: going from the abstract syntax of instructions to bits. Machine-code encoding is also important for some applications. For instance, binary rewriting has often been used to enforce security properties on untrusted code by inserting security checks before dangerous instructions [1,29,30], or for binary-level automatic parallelization [17]. After binary rewriting, the new code needs to be encoded into bits. Following the spirit of the previous decoder DSL, the machine-code encoding process should also be specified in some grammar with formal semantics. More importantly, we should be able to show the consistency between the decoder and the encoder: ideally, if we encode an instruction into bits and then decode those bits, we should get the instruction back (and also the other way around).

In this paper, we propose a DSL that allows the specification of both machine-code encoding and decoding in the same bidirectional grammar. The DSL is equipped with formal semantics. From a bidirectional grammar, we can generate a decoder and an encoder, as well as a machine-checked consistency

---

[1] NaCl on x86-32 restricts both memory reads and writes through hardware segmentation; NaCl on x86-64 and ARM, on the other hand, instruments only memory writes and does not protect confidentiality.

proof that relates the decoder and encoder. Major contributions of the paper are as follows:

- We propose a bidirectional grammar (abbreviated as bigrammar) DSL that allows simultaneous specification of decoding and encoding. Using dependent types, it enables correctness by construction: if a bidirectional grammar in the DSL can be type checked, then the generated decoder and encoder must be consistent. Our consistency definition takes into consideration that practical parsers may lose information during parsing and may produce values in loose semantic domains.
- We have used the bigrammar DSL to specify the decoding and encoding of an x86-32 model and a MIPS model, which demonstrates the practicality of our proposed DSL. These models include substantial subsets of the x86-32 and MIPS instruction sets, with around 300 instructions in the x86-32 model and 50 in the MIPS model.[2] In this process we identified a dozen bugs in our previous x86 encoder and decoder, which were written separately and without a correctness proof.
- We have extensively tested the generated x86 decoder and encoder and evaluated their performance. Experiments show that the encoder generated from the x86 grammar runs faster than a manually developed x86 encoder.
- The bigrammar DSL and its semantics are formally encoded in Coq [9] and all proofs are machine-checked. The coq development is open sourced and available at the following URL: `https://github.com/gangtan/CPUmodels/releases/tag/v1.1`.[3]

Machine decoding is an instance of parsing and encoding is an instance of pretty-printing. There has been previous work in the Haskell community on unifying parsing and pretty printing using invertible syntax [2, 13, 24]. In comparison, since our DSL is embedded in Coq, consistency proofs between decoding and encoding are explicitly represented as part of bigrammars and machine checked. Previous work in Haskell relies on paper and pencil consistency proofs. Another difference is on the consistency definition. Early work [2, 13] required that parsers and pretty-printers are complete inverses (i.e., they form bijections). Rendel and Ostermann [24] argued that the bijection requirement is too strong in practice and proposed a consistency definition based on partial isomorphisms. We further simplify the requirement by eliminating equivalence relations in partial isomorphisms; details will be in Section 3.

The rest of the paper is structured as follows. In Section 2, we first introduce the decoder DSL designed in our previous work. In Section 3, we discuss how consistency should be defined between a parser and its corresponding pretty printer. Based on the consistency definition, we present in Section 4 a dependently typed bigrammar language through which one can specify a

---

[2] Henceforth, we just write x86 for its 32-bit version.

[3] The x86 bigrammar is under `x86model/Model` and the MIPS bigrammar under `MIPSmodel`. For the x86model, the bigrammar DSL is in `BiGrammar.v` and an x86 bigrammar is in `X86BG.v` and `X86BG_ins.v`.

parser and a pretty printer simultaneously. Section 5 generalizes the definitions to ambiguous bigrammars. In Section 6, we discuss our experience of migrating grammars for x86 and MIPS to bigrammars. Finally, we discuss related work in Section 7 and conclude in Section 8.

A conference version of this article was published in the Proceedings of the Eighth International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2016) [28]. We describe the differences between the conference version and this article as follows. First, we have implemented a more efficient mechanism (described in Sec. 6.1.3) for combining a list of bigrammars into a single bigrammar via a disjoint-sum operator; it speeds up bigrammar proof checking substantially when combining a large list of bigrammars. Second, a bigrammar for the instruction decoding and encoding of the MIPS architecture has been implemented. Third, we compared the performance between the encoder generated from the x86 bigrammar and a manually developed x86 encoder (see Sec. 6.3); it shows the generated encoder runs even faster than the manual encoder. Finally, we have expanded the discussion of related work and steamlined the writing.

2 Background: the Decoder DSL

We next briefly describe the decoder DSL, upon which the bidirectional DSL is based. The decoder DSL was developed as part of RockSalt [21], a machine-code security verifier with a formal correctness proof mechanized in Coq. The decoder language is embedded into Coq and lets users specify bit-level patterns and associated semantic actions for transforming input strings of bits to outputs such as abstract syntax. The pattern language is limited to regular expressions, but the semantic actions are arbitrary Coq functions. The decoder language is defined in terms of a small set of constructors given by the following type-indexed datatype:

```
Inductive grammar : Type → Type :=
| Char : ch → grammar ch
| Eps : grammar unit
| Zero : ∀t, grammar t
| Cat : ∀t₁ t₂, grammar t₁ → grammar t₂ → grammar (t₁ ∗ t₂)
| Alt : ∀t₁ t₂, grammar t₁ → grammar t₂ → grammar (t₁ + t₂)
| Map : ∀t₁ t₂, (t₁ → t₂) → grammar t₁ → grammar t₂
| Star : ∀t, grammar t → grammar (list t)
```

The above inductive definition in Coq lists all constructors that can build "grammar t", which stands for grammars that produce values of type t. A grammar is parameterized by ch, the type for input characters. For machine decoders, the ch type contains bits 0 and 1 (represented by boolean false and true in our development). For each constructor, the definition of grammar presents its type. For instance, constructor Char takes a character and builds a grammar of type "grammar ch".

Semantically, a value of type "grammar t" represents a relation between input strings, which are lists of ch's, and semantic values of type t. Alternatively, we can think of the grammar as matching an input string and returning a set of associated semantic values. Formally, the denotation of a grammar is the least relation over strings and values satisfying the following equations:

$$\begin{aligned}
[\![\texttt{Char}\,c]\!] &= \{\,(c :: \texttt{nil}, c)\,\} \\
[\![\texttt{Eps}]\!] &= \{\,(\texttt{nil}, \texttt{tt})\,\} \\
[\![\texttt{Zero}]\!] &= \emptyset \\
[\![\texttt{Cat}\,g_1\,g_2]\!] &= \{\,(s_1 s_2, (v_1, v_2)) \mid (s_i, v_i) \in [\![g_i]\!]\,\} \\
[\![\texttt{Alt}\,g_1\,g_2]\!] &= \{\,(s, \texttt{inl}\ v_1) \mid (s, v_1) \in [\![g_1]\!]\,\} \cup \{\,(s, \texttt{inr}\ v_2) \mid (s, v_2) \in [\![g_2]\!]\,\} \\
[\![\texttt{Map}\,f\,g]\!] &= \{\,(s, f(v)) \mid (s, v) \in [\![g]\!]\,\} \\
[\![\texttt{Star}\,g]\!] &= [\![\texttt{Map}\,(\lambda\,\_.\,\texttt{nil})\,\texttt{Eps}]\!] \cup [\![\texttt{Map}\,(::)\,(\texttt{Cat}\,g\,(\texttt{Star}\,g))]\!]
\end{aligned}$$

Grammar "$\texttt{Char}\ c$" matches strings containing only the character c, and returns that character as the semantic value. $\texttt{Eps}$ matches only the empty string and returns $\texttt{tt}$ (Coq's unit value). Grammar $\texttt{Zero}$ matches no strings and thus returns no values. When g_1 is a grammar that returns values of type t_1 and g_2 is a grammar that returns values of type t_2, then "$\texttt{Alt}\,g_1\,g_2$" matches a string s if either g_1 or g_2 matches s. It returns values of the sum type $t_1 + t_2$; we use $\texttt{inl}$ for the left injection function of type "$t_1 \to (t_1 + t_2)$", and $\texttt{inr}$ for the right injection function of type "$t_2 \to (t_1 + t_2)$". "$\texttt{Cat}\,g_1\,g_2$" matches a string if it can be broken into two pieces that match the grammars. It returns a pair of the values computed by the grammars. $\texttt{Star}$ matches zero or more occurrences of a pattern, returning the result as a list. Note that "$[\![\texttt{Star}\,g]\!]$" is not defined as $[\![\texttt{Alt}\,\texttt{Eps}\,(\texttt{Cat}\,g\,(\texttt{Star}\,g))]\!]$ because we want it return lists.

$\texttt{Map}$ is the constructor for semantic actions. When g is a grammar that returns t_1 values, and f is a function of type $t_1 \to t_2$, then "$\texttt{Map}\,f\,g$" is the grammar that matches the same set of strings as g, but transforms the outputs from t_1 values to t_2 values using f.

The denotational semantics allows formal reasoning about grammars, but it cannot be directly executed. The operational semantics of grammars is defined using the notion of *derivatives* [7]. Informally, the derivative of a grammar g for an input character c is a residual grammar that returns the same semantic values as g and takes the same input strings except c. Using the notion of derivatives, we can build a parsing function that takes input strings and builds appropriate semantic values according to the grammar. We have also built a tool that constructs an efficient, table-driven recognizer from a grammar. Details about derivatives and table-driven recognizers can be found in our previous paper [21].

x86 ISA and its decoder grammar. The x86 architecture is notoriously complicated, with thousands of instructions. Using the decoder DSL, our previous work [21] developed a decoder grammar for a substantial fragment of x86-32. The fragment includes about 300 different instructions, including all integer, floating-point, SSE, and MMX instructions. It includes support for operands that include byte and word immediates, registers, and complicated addressing

```
Definition INC_p : grammar instr_t :=
   ("1111" $$ "111" $$ anybit $ "11000" $$ reg) @
      (fun p => let (w,r) := p in INC w (Reg_op r))
|| ("0100" $$ "0" $$ reg) @
      (fun r => INC true (Reg_op r))
|| ("1111" $$ "111" $$ anybit $ ext_op_modrm_noreg "000") @
      (fun p => let (w,addr) := p in INC w (Address_op addr)
```

Fig. 1 Parsing specification for the INC instruction.

modes (*e.g.*, scaled index plus offset). Furthermore, the fragment allows prefix bytes, such as operand size override, locking, and string repeat, that can be combined in many different ways and added to the front of instructions.

BNF grammars of the abstract syntax of the x86 instructions in our fragment are sketched as follows:

Register $\quad reg \quad ::= \texttt{EAX} \mid \texttt{ECX} \mid \texttt{EDX} \mid \cdots$
Segment Reg. $sreg ::= \texttt{ES} \mid \texttt{CS} \mid \texttt{SS} \mid \cdots$
Scale $\quad scale ::= 1 \mid 2 \mid 4 \mid 8$
Operand $\quad op \quad ::= int_{32} \mid reg \mid$
$\qquad\qquad\qquad int_{32} \times \texttt{option}(reg) \times \texttt{option}(scale \times reg)$
Instruction $\quad i \quad ::= \texttt{AAA} \mid \texttt{AAD} \mid \texttt{AAM} \mid \texttt{AAS} \mid \texttt{ADC}\ b\ op_1\ op_2 \mid$
$\qquad\qquad\qquad \cdots \mid \texttt{INC}\ b\ op \mid \cdots$

For instance, the INC instruction takes two arguments: a boolean b that tells whether it would operate on 32 bits or 8 bits and an operand op.

The job of the x86 decoder grammar is to specify the correspondence between bytes and abstract syntax of instructions, by patterns at the bit level. Fig. 1 gives an example grammar for the x86 INC instruction. We use Coq's notation mechanism to make the grammar more readable. Next we list the definitions for the notation used.

$g \mathbin{@} f := \texttt{Map}\ f\ g$
$g_1 \mathbin{\$} g_2 := \texttt{Cat}\ g_1\ g_2$
$\texttt{literal}\ [c_1, \ldots, c_n] := (\texttt{Char}\ c_1) \mathbin{\$} \ldots \mathbin{\$} (\texttt{Char}\ c_n)$
$s_1 \mathbin{\$\$} g_2 := ((\texttt{literal}\ s_1) \mathbin{\$} g_2) \mathbin{@} \texttt{snd}$
$g_1 \mathbin{||} g_2 := (\texttt{Alt}\ g_1\ g_2) @ (\lambda v.\ \texttt{match}\ v\ \texttt{with}\ \texttt{inl}\ v_1 \Rightarrow v_1 \mid \texttt{inr}\ v_2 \Rightarrow v_2\ \texttt{end})$

Note that "$g_1 \mathbin{||} g_2$" uses the union operation and assumes both g_1 and g_2 are of type "grammar t" for some t. It throws away information about which branch is taken.

At a high-level, the grammar in Fig. 1 specifies three alternatives that can build an INC instruction. Each case includes a pattern specifying literal sequences of bits (*e.g.*, "1111"), followed by other components like anybit or reg that are themselves grammars that compute values of an appropriate type. For example, in the first case, we take the bit returned by anybit and the register returned by reg and use them to build the abstract syntax for a version of the INC instruction with a register operand.

3 Relating Parsing and Pretty-Printing

A machine decoder is a special parser and a machine encoder is a special pretty printer. In general, a parser accepts an input string s and constructs a semantic value v according to a grammar. A pretty printer goes in the reverse direction, taking a semantic value v and printing a string s according to some grammar. In this section, we discuss how parsers and pretty printers should be formally related. We will first assume an unambiguous grammar g: that is, for a string s, there is at most one v so that $(s, v) \in [\![g]\!]$. In Section 5, we will present how to generalize it to ambiguous grammars.

Ideally, a parser and its corresponding pretty printer should form a bijection [2,13]: (i) if we parse some string s to get some semantic value v and then run the pretty printer on v, we should get the same string s back, and (ii) if we run the pretty printer on some v to get string s and then run the parser on s, we should get value v back. While some simple parsers and pretty printers do form bijections, most of them do not, because of the following two reasons.

Information loss during parsing. Parsing is often forgetful, losing information in the input. A simplest example is that a source-code parser often forgets the amount of white space in the AST produced by the parser. Another example happens when the union operator (i.e., $||$) is used during parsing. The problem for "$g_1 || g_2$" is that, if it parses the input to a semantic value v, it is possibly the result of either g_1 or g_2; consequently, there are two possibilities when reversing v to an input. As a simple example, in "$(\texttt{Eps}@(\lambda\_.c)) || (\texttt{Char}\,c)$", both branches can produce semantic value c. It loses the information about which branch was taken during parsing. In contrast, "$\texttt{Alt}\,(\texttt{Eps}@(\lambda\_.c))\,(\texttt{Char}\,c)$" produces tagged semantic values, which do not lose information. Our x86 decoder grammar has many uses of unions. An example is the `INC` grammar in Fig. 1.

Because information is lost in a typical parser, multiple input strings may be parsed to the same semantic value. Therefore, for such a semantic value, the pretty printer has to either list all possible input strings, or choose a particular one, which may not be the same as the original input string. In this work, we take the second option since listing all possible input strings can be challenging for certain parsers (e.g., x86 has many bit-string encodings for the same instructions and operands; enumerating all of them during encoding is troublesome at least).
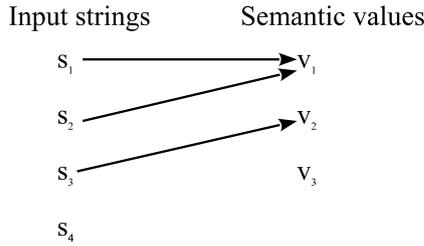
Loose semantic domains. A parser produces semantic values in some domain. For uniformity the semantic domain may include values that cannot be possible parsing results. Here is a contrived example: a parser takes strings that represent even numbers and converts them to values in the natural-number domain; the result domain is loose as the parser cannot produce odd numbers. Our x86 decoder has many examples, especially with respect to instruction operands. In the x86 syntax, operands can be immediates, registers, memory addresses, etc.; an instruction can take zero or several operands. A two-operand

instruction cannot use memory addresses for both operands, but for uniformity our decoder just uses the operand domain for both operands. Similarly, some instructions cannot take all registers but only specific registers, but our decoder also uses the operand domain for these instructions.

Some of these issues can be fixed by tightening semantic domains so that they match exactly the set of possible parse results. While this is beneficial in some cases, it would in general require the introduction of many refined semantic domains, which would make the abstract syntax and the processing following parsing messy. For the example of x86 operands, we would need to define extra syntax for different groups of operands and, when we defined the semantics of instructions, we would need to introduce many more interpretation functions for those extra groups of operands.

The implication of loose semantic domains is that the pretty printer has to be partial: it cannot convert all possible semantic values back to input strings.

Formalizing consistency between parsing and pretty printing. The following diagram depicts the relationship between the domain of input strings and the output semantic domain for a parser: because of information loss during parsing, multiple input strings can be parsed to the same semantic value; because of loose semantic domains, some values may not be possible parsing results; finally, a typical parser is partial and may reject some input strings during parsing.

With the above diagram in mind, we next formalize the properties we desire from a parser and its corresponding pretty printer. Both the parser and the pretty printer are parameterized by a grammar of type "`bigrammar` t", which stands for bidirectional grammars that produce semantic values of type t. We will present the details of our bidirectional grammars in the next section; for now we just discuss the desired properties about the parser and the pretty printer that are derived from a bigrammar. These properties will be used to motivate the design of bigrammars.

Formally, a parser turns an input string (as a list of chars) into a possible value[4] of type t, according to a grammar indexed by t. A pretty printer encodes

---

[4] Our parser implementation actually returns a list of values during parsing, for simplicity of presentation we ignore that aspect in this paper.

a semantic value in a possible string, according to a grammar.

$$\texttt{parse} : \forall t, (\texttt{bigrammar } t) \rightarrow \texttt{list ch} \rightarrow \texttt{option } t$$
$$\texttt{pretty-print} : \forall t, (\texttt{bigrammar } t) \rightarrow t \rightarrow \texttt{option (list ch)}$$

Two consistency properties that relate a parser and a pretty printer for the same grammar g of type "$\texttt{bigrammar } t$" are as follows:

Definition 1 (Consistency between parsers and pretty printers)

Prop1: If $\texttt{parse } g \ s = \texttt{Some } v$, then exists s' so that $\texttt{pretty-print } g \ v = \texttt{Some } s'$.

Prop2: If $\texttt{pretty-print } g \ v = \texttt{Some } s$, then $\texttt{parse } g \ s = \texttt{Some } v$.

Property 1 says that if a parser turns an input string s into a semantic value v, then the pretty printer should encode that value into some input string s'; however, s and s' may be different—this is to accommodate the situation when multiple input strings may correspond to the same semantic value. The property allows the pretty printer to choose one of them (the pretty printer cannot just pick an arbitrary string that is unrelated to v because of Property 2).

Property 2 says that if the pretty printer encodes value v in string s, then the parser should parse s into the same value. Note that it places no restriction when the pretty printer cannot invert v—this is to accommodate a loose semantic domain in which some semantic values are not possible parsing results.

Rendel and Ostermann [24] proposed to use partial isomorphisms to relate parsers and pretty-printers. A partial isomorphism requires the same Property 2, but also requires s' and s are in some equivalence relation in Property 1. While it is mathematically appealing, requiring an extra equivalence relation would require adding unintuitive equivalence relations for many of our examples. For instance, x86 often has multiple bit-string encodings for the same instruction; for each case, we would have to define a special equivalence relation that just relates those bit strings. In our approach, the information about equivalence is actually contained in the bigrammar; the implicit equivalence relation relates all bit strings that are parsed to the same semantic value. We could build an additional layer on top of our design and provide additional checking to ensure an explicitly specified equivalence relation is consistent with the implicit equivalence relation, but at the price of a programmer specifying the equivalence relations explicitly.

4 A Bidirectional Grammar

It would certainly be possible to write a parser and a pretty printer separately and then develop a correctness proof based on the consistency definition we presented. We actually developed an encoder separately from the x86-32 decoder (extracted from the decoder grammar). However, we realized that developing a correctness proof this way was rather difficult. Since the decoder

```
Inductive bigrammar : Type → Type :=
| char : ch → bigrammar ch
| eps : bigrammar unit
| zero : ∀t, bigrammar t
| cat : ∀t₁ t₂, bigrammar t₁ → bigrammar t₂ → bigrammar (t₁ * t₂)
| alt : ∀t₁ t₂, bigrammar t₁ → bigrammar t₂ → bigrammar (t₁ + t₂)
| map : ∀t₁ t₂ (f₁ : t₁ → t₂)(f₂ : t₂ → option t₁)(g : bigrammar t₁)
          (pf : invertible(f₁, f₂, g)), bigrammar t₂
| star : ∀t, bigrammar t → bigrammar (list t)

where invertible(f₁, f₂, g) ≜
    (∀v : t₁, v ∈ rng(g) ⇒ ∃v', f₂ (f₁ v) = Some v' ∧ v' ∈ rng(g))∧
    (∀v : t₁ ∀w : t₂, v ∈ rng(g) ⇒ f₂ w = Some v ⇒ f₁ v = w)
and rng(g) ≜ { v | ∃s, (s, v) ∈ [g] }
```

Fig. 2 Bigrammar DSL syntax.

and the encoder were developed separately, their internal structures were not designed to match closely and thus not amenable to a proof that relates them.

More importantly, the reverse pretty-printing functions for most constructors in our decoder DSL can be automatically calculated, but the separate encoder does not take advantage of that. For instance, the parser for "Cat g_1 g_2" parses the input string to construct a pair of values (v_1, v_2), the reverse pretty-printing function is then to encode v_1 to get s_1 according to g_1, encode v_2 to get s_2 according to g_2, and return s_1 followed by s_2. In fact, if a grammar forgoes the use of Map, then the semantic value returned by the grammar represents the input as a parse tree, which loses no information; the pretty printer can easily take the parse tree and the grammar to reconstruct the input string.

Our bigrammar DSL takes advantage of the above observation and requires an inverse function only for the map case. Both a parser and a pretty printer are extracted from a bigrammar. Furthermore, the extracted parser and the pretty printer meet the consistency requirement. Therefore, it enables correctness by construction of parsers and pretty printers.

Fig. 2 presents the bigrammar DSL syntax. We use lower-case constructors for bigrammars to distinguish them from grammar constructors. They are almost the same as those in the decoder DSL, except that the map case requires an additional partial inverse function that goes from values of type t_2 to values of type "option t_1" as well as a proof showing that the map function f_1 and the inverse function f_2 are invertible (we will discuss the invertible definition later). The inverse function is partial to accommodate loose semantic domains; i.e, some values cannot be possible results of the map function.

The denotational semantics of bigrammars is the same as the denotational semantics of the decoder DSL in Section 2, except that the map case ignores the inverse function and the proof; therefore, we do not repeat it. We still use g for a bigrammar. As before, notation $[\![g]\!]$ is the denotation of g and contains all pairs of (s, v) according to the denotational semantics.

Fig. 3 presents the pretty printer for bigrammars. It takes a bigrammar and a semantic value, and returns an optional string s for the semantic value ac-

```
pretty-print (char c)        = λc'. if c = c' then Some (c :: nil) else None
pretty-print eps             = λ_. Some nil
pretty-print (zero t)        = λ_. None
pretty-print (cat g₁ g₂)     = λv. s₁ ← pretty-print g₁ (fst v);
                                    s₂ ← pretty-print g₂ (snd v);
                                    ret (s₁++s₂)
pretty-print (alt g₁ g₂)     = λv. match v with
                                    | inl v₁ ⇒ pretty-print g₁ v₁
                                    | inr v₂ ⇒ pretty-print g₂ v₂ end
pretty-print (map f₁ f₂ g pf) = λv. v' ← f₂ v; pretty-print g v'
pretty-print (star g)        = fix λpp v. match v with
                                    | nil ⇒ Some nil
                                    | hd :: tl ⇒
                                        s₁ ← pretty-print g hd; s₂ ← pp tl;
                                        ret (s₁++s₂) end
```

Fig. 3 The `pretty-print` function for bigrammars.

cording to the bigrammar. We comment only on a few cases next. For "`char c`",
since it is impossible for "`char c`" to produce a char that is different from c,
the pretty printer tests if the semantic value c' is the same as c; if so, the input
string must be a single-char string that contains c; otherwise, it returns `None`.
Since `pretty-print` returns an optional value, we use an option monad and
use the Haskell-style monadic notation ($\leftarrow$ and `ret`) to simplify the syntax
of propagating `None` values.[5] Take the case of "`cat g₁ g₂`" as an example: if
running `pretty-print` on g_1 produces `None`, then it returns `None` for the `cat`
grammar; otherwise, the returned string is bound to s_1 and `pretty-print`
is run on g_2; it either returns `None` or some s_2; in the first case, `None` is re-
turned; in the second case, return s_1++s_2, which is s_1 concatenated with s_2.
For "`map f₁ f₂ g pf`", the inverse function f_2 is first used to convert v to a
possible v'; if it succeeds, `pretty-print` is run on v' according to g. Only
the `map` case uses an explicit inverse function; other cases' inverse functions
are completely determined by the shapes of the bigrammar and the semantic
value.

In "`map f₁ f₂ g pf`", a proof that the map function and the inverse function
are invertible is required. The definition of invertibility is formulated so that
the parser and the pretty printer for a bigrammar should meet the correct-
ness properties in Def. 1. Therefore, the two conditions closely follow the two
properties in Def. 1. The definition also takes g as a parameter and quantifies
over all values in the range of g; this is to accommodate the situation when
the range of g is a strict subset of the values in t_1. Requiring that the property
holds for all values in t_1 would be too strong and unnecessary (and make the
invertibility conditions unprovable for certain useful bigrammars).

Now we show how to prove that the parser and the pretty printer for a
bigrammar meet the consistency requirement. With the help of denotational
semantics, we can decouple the correctness proof of the parser from the cor-

---

[5] That is, "`ret s`" is defined as `Some s`; and "$s_1 \leftarrow v; f$" is defined as
`match v with | None ⇒ None | Some s₁ ⇒ f s₁ end`.

rectness proof of the pretty printer. The correctness of the parser extracted from a grammar has been shown in the RockSalt paper [21] for a derivative-based parser; in the same way, a derivative-based parser can be extracted from a bigrammar with a similar correctness proof:

Theorem 1 *(Parser correctness for for unambiguous bigrammars.)*

$$(s, v) \in [\![g]\!] \text{ if and only if } \texttt{parse } g \ s = \texttt{Some } v.$$

The second theorem is about pretty-printer correctness.

Theorem 2 *(Pretty-printer correctness for unambiguous bigrammars.)*

(1) If $(s, v) \in [\![g]\!]$, then exists s' so that $\texttt{pretty-print } g \ v = \texttt{Some } s'$.
(2) If $\texttt{pretty-print } g \ v = \texttt{Some } s$, then $(s, v) \in [\![g]\!]$.

The proof of the above theorem is straightforward, based on induction over the syntax of g. With the theorems about parser and pretty-printer correctness, it can be checked easily that the consistency requirement in Def. 1 is a corollary.

The following notation is also introduced to simplify bigrammar construction:

$$g @ f_1 \& f_2 \& pf := \texttt{map } f_1 \ f_2 \ g \ pf \qquad g_1 + g_2 := \texttt{alt } g_1 \ g_2$$

5 Generalization to Ambiguous Grammars

An ambiguous bigrammar g can relate the same input string s to multiple semantic values. A simple example is "$\texttt{alt } (\texttt{char } c) \ (\texttt{char } c)$", where c is some character; it relates the single-character string c to "$\texttt{inl } c$" and "$\texttt{inr } c$". Because of ambiguity, the type of the parser is changed to the following:

$$\texttt{parse} : \forall t, (\texttt{bigrammar } t) \to \texttt{list ch} \to \texttt{list } t$$

It takes a possibly ambiguous bigrammar and an input string, and returns a list of values of type t. Correspondingly, the correctness theorem for the parser has to change:

Theorem 3 *(Parser correctness for ambiguous bigrammars.)*

$$(s, v) \in [\![g]\!] \text{ if and only if } v \in \texttt{parse } g \ s.$$

We write "$v \in \texttt{parse } g \ s$" to mean that v is in the list produced by $\texttt{parse } g \ s$.

The type signature of the pretty printer and its correctness formulation are as before. With parser and pretty-printer correctness, we can show the following consistency theorem:

Theorem 4 *(Consistency between parsers and pretty printers for ambiguous bigrammars.)*

Prop1: If $v \in \texttt{parse } g \ s$, then exists s' so that $\texttt{pretty-print } g \ v = \texttt{Some } s'$.
Prop2: If $\texttt{pretty-print } g \ v = \texttt{Some } s$, then $v \in \texttt{parse } g \ s$.

6 Engineering Bigrammars for ISA Decoding and Encoding

In previous work [21], we defined grammars in the decoder DSL for two Instruction Set Architectures (ISAs): 32-bit x86 and MIPS. Decoders are then generated from those grammars. Using the new bigrammar DSL, we retrofitted the grammars into bigrammars. In this section, we discuss the retrofitting process and also compare between grammars and their retrofitted bigrammars.

6.1 From grammars to bigrammars

Since our bigrammar language differs from the decoder language only for the `Map` constructor, retrofitting a grammar to a bigrammar in general involves adding inverse functions and invertibility proofs to places where `Map` is used in the grammar. A direct port from the original grammar to a bigrammar, without changing the grammar's structure, would be possible. However, to make it easier to develop invertibility proofs and to make the encoder more efficient, we often needed to change grammar rules (sometimes substantially) and introduce new constructors. Using representative examples from the x86 grammar, we next discuss what changes we made to engineer decoder/encoder bigrammars for ISAs.

6.1.1 Tighten semantic domains.

Most of the changes were because the map functions used in the original x86 decoder grammar are not surjective, causing loose semantic domains. Some of those instances can be fixed by having a tightened semantic domain. Here is a typical example. The grammar for parsing immediate values takes either 32 bits or 16 bits, depending on an operand-size override flag, and returns an immediate operand:

```
Definition imm_p (opsize_override:bool): grammar operand_t :=
  match opsize_override with
    | false => word @ (fun w => Imm_op w)
    | true => halfword @ (fun w => Imm_op (sign_extend16_32 w))
  end.
```

In the above, `word` and `halfword` are grammars that parse 32-bit and 16-bit immediate values, respectively. Recall that "g @ f" is the notation for "`Map` f g", which converts the value returned by g using semantic action f.

To convert it to a bigrammar, we need to add one inverse function for each of the two cases. However, the operand domain contains not just immediate operands, but also other kinds of operands such as register operands. So the inverse function, which takes an operand value, has to first check if the value is an immediate operand. For instance, the inverse function for the first case is:

```
fun op => match op with | Imm_op w => Some w | _ => None end
```

13

Furthermore, we need a lemma that says operands produced by `imm_p` must be immediate operands and use the lemma in other bigrammars that use `imm_p`.

This approach can produce a bigrammar, but the inverse function has runtime tests on operands. The reason for those runtime tests is that the map functions in `imm_p` are not surjective and the resulting `operand` domain is loose. The fix is to change `imm_p` to return 32-bit immediates and any client of it applies `Imm_op` in its map function when necessary. This makes the inverse function more efficient by avoiding runtime tests. In particular, the `imm_b` bigrammar is as follows (in this and following examples, invertibility proofs are omitted and are represented as _).

```
Program Definition imm_b (opsize_override:bool):
  bigrammar word_t :=
  match opsize_override with
    | false => word
    | true => halfword @ (fun w => sign_extend16_32 w)
                        & (fun w =>
                             if repr_in_signed_halfword_dec w then
                               Some (sign_shrink32_16 w)
                             else None)
                        & _
  end.
```

6.1.2 Eliminating the uses of the union operator.

Not all instances of loose semantic domains can be fixed easily, because of the extensive use of the union operator in our decoder grammars. Many instructions' grammars have multiple cases. We have seen a typical example about the `INC` instruction in Fig. 1, which has three cases. Each case uses a map function and the three cases are combined through union, which throws away information about which case is taken during parsing.

To turn the `INC` grammar to a bigrammar, one possible way is to add an inverse function for each of the three cases. For instance, the inverse function for the first case (copied below) would pattern match the two arguments; if they are of the form "`w, (Reg_op r)`", return `Some (w,r)`; otherwise, return `None`.

```
"1111" $$ "111" $$ anybit $ "11000" $$ reg @
   (fun p => let (w,r) := p in INC w (Reg_op r))
```

The three cases can then be combined using a special union operator that constructs bigrammars:

```
Program Definition union t (g1 g2:bigrammar t): bigrammar t :=
  (g1 + g2)
    @ (fun w => match w with inl v1 => v1 | inr v2 => v2 end)
    & (fun v: t =>
```

14

```
      match pretty_print g1 v with
        | Some _ => Some (inl v)
        | None =>
          match pretty_print g2 v with
            | Some _ => Some (inr v) | None => None end
      end)
  & _.
```

The inverse function for the union operator implements a backtracking seman-
tics: it uses the pretty printer to first try the left branch; if it succeeds, inject
v to the left branch; if it fails, it tries the right branch. The union constructor
is biased toward the left branch if v can be produced by both branches.

Although the use of union is convenient for converting grammars to bi-
grammars, it is terribly inefficient. If the union is used to convert the INC
grammar to a bigrammar, each case needs an inverse function that performs
strict pattern matches and the inverse function in the union sequentially tries
each case and checks which one succeeds. It comes with many runtime tests.
Things get worse for instructions that have more cases; for instance, the gram-
mar for MOV has a dozen cases. In general, suppose g is the result of combining
n bigrammars via union, each bigrammar is of size m, and pretty printing
each bigrammar takes time $O(m)$. Then running the pretty printer on g takes
$O(n * m)$ time in the worst case since it may try all n possibilities. Therefore,
the use of the union operator should be avoided as much as possible and it
would be much more efficient to have some test at the top of the inverse func-
tion and use the test result to dispatch to *one of* the m cases, which leads to
a complexity of $O(n + m)$. Our first version of the x86 bigrammar used the
union operator for convenience, but in a subsequent version all uses of union
were eliminated. We next discuss how this was achieved via the INC example.

In the following INC bigrammar, we combine cases using the disjoint-sum
operator alt (abbreviated as +). Essentially, the three cases are combined to
produce a parse tree and a single map function is used to convert the parse
tree into arguments for INC. A single inverse function then converts arguments
for INC to a parse tree. We will discuss later why the map function constructs
arguments for INC instead of INC instructions.

```
Program Definition INC_b: bigrammar (pair_t bool_t operand_t) :=
(   "1111" $$ "111" $$ anybit $ "11000" $$ reg
  + "0100" $$ "0" $$ reg
  + "1111" $$ "111" $$ anybit $ ext_op_modrm_noreg "000")
  @ (fun v => match v with
                | inl (w,r) => (w, Reg_op r)
                | inr (inl r) => (true, Reg_op r)
                | inr (inr (w,addr)) => (w, Address_op addr)
              end)
  & (fun u => let (w,op):=u in
        match op with
          | Reg_op r =>
```

```
            if w then Some (inr (inl r)) else Some (inl (w,r))
        | Address_op addr => Some (inr (inr (w,addr)))
        | _ => None
      end)
  & _.
```

Since the above pattern is used over and over again, we have constructed specialized Coq tactics for facilitating the process. Suppose we have a list of n bigrammars and bigrammar g_i is of type "bigrammar t_i" and map function f_i is of type $t_i \rightarrow t$. For instance, the INC example has three cases:

```
Grammar 1: "1111" $$ "111" $$ anybit $ "11000" $$ reg
Map 1: fun v => let (w,r):=v in (w, Reg_op r)
Grammar 2: "0100" $$ "0" $$ reg
Map 2: fun r => (true, Reg_op r)
Grammar 3: "1111" $$ "111" $$ anybit $ ext_op_modrm_noreg "000"
Map 3: fun v => let (w,addr):=v in (w, Address_op addr)
```

Bigrammars g_1 to g_n can then be combined through repeated uses of alt and a single map function can be produced based on f_1 to f_n. We have automated the process by introducing tactics that combine bigrammars and for producing the map function based on f_1 to f_n. The inverse function needs to perform case analysis over values of type t and construct appropriate parse trees. A special tactic is introduced to facilitate the writing of the inverse function; it takes a case number and a value and produces a parse tree by inserting appropriate inl and inr constructors.

6.1.3 Speeding up tactics for combining bigrammars

When combining n bigrammars via the disjoint-sum operator for a large n, how the tactics are implemented makes a substantial difference in Coq's proof-checking time. For our initial versions of automation tactics, it literally took Coq hours to check a bigrammar proof when n is greater than 30. To speed up proof checking, we have implemented two optimization techniques.

First, our tactics construct balanced parse trees when combining bigrammars via the disjoint-sum operator; this significantly speeds up the proof checking of the combined bigrammar. To see why, one naive way would be to combine bigrammars in a right-associative fashion (or similarly in a left-associative fashion): $g = g_1 + (g_2 + (\ldots + (g_{n-1} + g_n)))$. However, to inject a value v_i produced by g_i into a parse tree, an average of $O(n)$ number of inl and inr constructors would be used. In contrast, our tactics balance the sizes of branches when combining bigrammars. For instance, g_1 to g_4 are combined to be $(g_1 + g_2) + (g_3 + g_4)$. This way, an average of $O(\log n)$ number of inls and inrs are used to inject a value to a parse tree. When developing proofs about g, this makes a substantial difference in the speed of proof checking. For instance, bigrammar reg_no_esp accepts the encoding of all registers except for esp and has therefore seven cases; it took five seconds in Coq 8.4 to finish

16

checking the bigrammar definition for the right-associative combination and took only two seconds for the balanced combination.

The second technique is to use dependently typed terms to get compact representations of terms during intermediate proof states, which saves both space and time. This technique was implemented after the conference version [28] and resulted in about 10x speed up in proof checking when n is large (e.g., 32). We next briefly discuss the technique. As before, assume there is a list of n bigrammars and bigrammar g_i is of type "bigrammar t_i" and map function f_i is of type $t_i \rightarrow t$; the goal is to combine the list of bigrammars into a single bigrammar of type t.

We first introduce a typetree for a tree of types and an interpretation function that converts a type tree to a disjoint-sum type according to the tree shape:

```
Inductive typetree: Type :=
| Leaf: Type -> typetree
| Node: typetree -> typetree -> typetree.

Fixpoint interp_ttree (tt: typetree): Type :=
  match tt with
  | Leaf t => t
  | Node tt1 tt2 =>
    (interp_ttree tt1) + (interp_ttree tt2)
  end.
```

Then an inductive type bgr_tree is defined for a tree of bigrammars. The type is indexed by t, which is the return type of map functions, and a type tree, which collects the types of bigrammars in the bigrammar tree. Each leaf carries a bigrammar of type t_i and a map function of type $t_i \rightarrow t$.

```
Inductive bgr_tree (t:Type): typetree -> Type :=
| BLeaf: forall (ti:Type) (g:bigrammar ti),
    (ti -> t) -> bgr_tree t (Leaf ti).
| BNode: forall (tt1 tt2:typetree),
    bgr_tree t tt1 -> bgr_tree t tt2 ->
    bgr_tree t (Node tt1 tt2).
```

It is straightforward to construct a Coq tactic that converts a list of pairs of bigrammars and map functions (g_i, f_i) to a balanced bigrammar tree. Furthermore, the following two dependently typed functions can be constructed: comb_bigrammar combines the bigrammars in the input bigrammar tree using the disjoint-sum operator according to the tree structure; comb_map combines the map functions in the bigrammar tree into a single map function. Their definitions are omitted for brevity.[6]

```
Fixpoint comb_bigrammar (t tt) (bt:bgr_tree t tt):
```

_____

[6] Their definitions can be found in x86model/Model/BiGrammar.v in our online GitHub code repository.

```
  bigrammar (interp_ttree tt) := ...

Fixpoint comb_map (t tt) (bt:bgr_tree t tt):
  interp_ttree tt -> t := ...
```

The benefit of these dependently typed functions is that we can use them to write down the combined bigrammar in a succinct way.

```
(* assume bt's type is "bgr_tree t tt" *)
Definition combined : bigrammar t :=
  (comb_bigrammar bt) @ (comb_map bt)
    & ...    (* the inverse function *)
    & ...    (* the invertibility proof *)
```

Our previous version of tactics directly constructed the expanded results of "comb_bigrammar bt" and "comb_map bt", both of which are large terms when n is large. The new dependently typed terms allow the proof development to expand these definitions when necessary, saving space and time. The development of the inverse function cannot be fully automated; however, to facilitate the development of the inverse function, for each g_i of type "bigrammar t_i" our implementation automatically generates an injection function going from values of t_i to values of type "interp_ttree tt", where tt is a tree type that combines t_1 to t_n. We do not go into details for brevity and refer readers to our Coq development.

6.1.4 Combining instruction bigrammars.

After individual instruction bigrammars have been developed, we need to combine them into a global bigrammar for all instructions. This was relatively easy to achieve when we developed grammars (without inverse functions): each instruction grammar produced values of the `instr` type and instruction grammars were combined through union. We have seen an example instruction grammar in Fig. 1, which returns `INC` instructions.

When constructing bigrammars, the situation is more complicated because of the desire to eliminate unions. To achieve that, we make our instruction bigrammars return arguments for instructions instead of instructions; then instruction bigrammars are combined via the disjoint-sum operator. That is, the global instruction bigrammar produces a parse tree; this is similar to the technique for dealing with cases in individual instruction bigrammars. We next illustrate this using an instruction set of only two instructions:

```
  Inductive instr :=
    | AAA | INC (w:bool)(op1:operand).
  Definition AAA_b : bigrammar unit_t := ...
  Definition INC_b : bigrammar (pair_t bool_t operand_t) := ...
  Program Definition instr_b : bigrammar instr_t :=
    (AAA_b + INC_b)
    @ (fun v =>
```

```
      match v with | inl _ => AAA | inr (w,op) => INC w op end)
  & (fun i =>
      match i with | AAA => inl tt | INC w op => inr (w,op) end)
  & _.
```

As before, part of `instr_b` can be automatically generated using special Coq tactics.

However, during development we encountered the difficulty of running out of memory in Coq when checking `instr_b`. This was because our x86 model had around 300 instructions and it used a flat instruction syntax in which the `instr` definition has around 300 cases, one for each instruction. The resulting `instr_b` had an enormous number of uses of `inl`s and `inr`s; each use had an implicit type parameter (the type parameter of `inl`/`inr` tells the type of the right/left branch of the sum type). As a result, Coq ran out of memory (on versions 8.4 and 8.5) when checking the invertibility proof in `instr_b` and this was the case even after using balanced parse trees and employing special tricks in Coq to save memory (e.g., using the `abstract` tactic).

To mitigate the situation, we introduced hierarchical abstract syntax, which groups instructions by their categories according to the x86 ISA manual. In particular, instructions were divided into integer, floating-point, MMX, and SSE instructions. Instructions in one category were grouped into a separate instruction type and a bigrammar was developed for those instructions alone; as there were much fewer instructions in one group, Coq was able to type check the bigrammar. Then, we defined a bigrammar that converted the hierarchical instruction syntax into the flat instruction syntax (and also an inverse function for going the other direction).

6.2 Statistics about the x86 and MIPS bigrammars

Our x86 bigrammar specifies the decoding and encoding of 297 instructions, including all x86 integer, floating-point, SSE, and MMX instructions. It also allows prefixes that can be added to the front of instructions. So for each decoding operation the decoder returns (p, i), where p is the set of prefixes and i is the instruction.

The following table lists the lines of Coq code for our previous x86 decoder grammar, a separately developed x86 encoder, and our new bigrammar for x86 encoding and decoding. Since our bigrammars enforce correctness by construction, the x86 bigrammar implies its extracted decoder and encoder are consistent. In contrast, the separately developed decoder and encoder lacked the correctness proof that relates them. We have also extracted executable OCaml decoder and encoder code from our x86 bigrammar and are using them in a separate OCaml project.

	Lines of Coq code
x86 decoder grammar	2,194
x86 encoder	2,891
x86 decoder/encoder bigrammar	7,254

Our MIPS bigrammar specifies the decoding and encoding of 53 instructions. The following table lists the lines of Coq code for the previous MIPS decoder grammar and the new bigrammar for MIPS encoding and decoding. Unlike the x86 case, previously we did not write a separate MIPS encoder. The development of the MIPS bigrammar did not require any extension to our bigrammar mechanism, which demonstrates its general applicability to specifying ISA encoding/decoding.

	Lines of Coq code
MIPS decoder grammar	342
MIPS decoder/encoder bigrammar	1,036

During the construction of the x86 bigrammar, we found around a dozen bugs in the previous decoder grammar and encoder. The decoder grammar was extensively tested in RockSalt [21], so most of the identified bugs were in the encoder. For example, the old encoder for MOVBE disallows the combination of using registers for both of its two operands (i.e., the encoder maps the case to None), but the decoder can actually produce operands of that combination. When developing the proof for the corresponding bigrammar, we could not prove the first condition in the invertibility definition and this was fixed by modifying the inverse function. As another example, the old encoders for instructions BT/BTC/BTR/BTS allow the combination of using a register for the first operand and a memory address for the second operand; however, this combination is not a possible decoding result for those instructions. As a result, we could not prove the second condition in the invertibility definition and this was fixed by mapping the combination to None in the inverse function. Even though the old decoder grammar was extensively tested, we did find a couple of bugs. For instance, the old decoder for the Test instruction had a case as follows:

```
"1010" $$ "1000" $$ byte @
  (fun b => TEST true (Reg_op EAX) (Imm_op (zero_extend8_32 b)))
```

The first parameter of Test tells whether the test operation should be for 32-bit words (when the parameter's value is true) or for 8-bit bytes (when the parameter's value is false). The above case actually compares bytes, so the first argument of the resulting Test instruction should have been false.

Before we started migrating the x86/MIPS grammars, we wanted to determine whether or not moving from an established grammar and abstract syntax to a bigrammar required extensive changes. Our experience tells us that the answer is yes, especially when one wants to eliminate the use of union. On the other hand, some of those changes could be alleviated through clever tactics.

6.3 Encoder performance comparison

We were also interested in knowing how the performance of decoders/encoders generated from bigrammars compared to the performance of separately developed decoders/encoders. Since the decoder generated from a bigrammar is the

same as the decoder generated from our previous decoder grammar language, there is no performance difference.[7] On the other hand, the encoder generated from a bigrammar is based on the `pretty-print` function in Fig. 3 and has a quite different structure from our manually written encoder. To compare their performance, we extracted two encoders in OCaml from our Coq development, one from the x86 bigrammar and one from the manually written encoder. We then used a set of binaries of common Linux utility programs listed below as the test programs. For each program, the following table lists its size, the number of instructions, the time it takes to encode all instructions in the program via the bigrammar encoder, and the time it takes to encode all instructions via the manual encoder.

	size	instr count	bigrammar encoder	manual encoder
tailf	14KB	2,020	1.19s	2.05s
pwd	26KB	3,938	2.50s	4.19s
cat	46KB	7,458	4.99s	8.28s
ls	103KB	18,377	10.73s	18.92s

We were surprised to see that the bigrammar encoder runs almost twice as fast as the manual encoder. We suspect the reason is that the manual encoder used many literal strings during encoding, resulting in higher memory consumption. For instance, the following encoder for the `LOOP` instruction expands the string "11100010" to a list of bools via `s2bl` and concatenates it with the encoding of a displacement. In contrast, the encoder derived from the x86 bigrammar directly outputs a list of bools. After inspecting statistics from OCaml's garbage collector, it was confirmed that the manual encoder consumes about 70% more memory than the bigrammar encoder.

```
Definition enc_LOOP (disp:int8) :=
  ret (s2bl "11100010" ++ enc_byte disp).
```

Although it is likely that we can optimize the performance of the manual encoder, the performance comparison shows that the generated encoder from a bigrammar produces competitive performance.

7 Related Work

Most computer systems contain one or more parsers and pretty-printers. Parsers are especially security critical as they often parse untrusted data. As one example, the Windows operating system has hundreds of parsers for parsing different file formats such as JPEG and MP3. As another example, a web browser contains a dozen parsers for parsing web data of various formats, including HTML, XML, JavaScript, and JSON. Many security-critical bugs

---

[7] We also performed a performance comparison between our decoder and Linux's `objdump`. The derivative-based x86 decoder introduced in Rocksalt [21] was about 100 times slower than `objdump`. We recently made improvements to produce a table-based decoder from the x86 grammar; its performance is within 2x compared to `objdump`.

were found in these parsers, for example, through the approach of white-box fuzzing [12]. The white-box fuzzing approach, however, is based on testing and cannot guarantee parser correctness.

Formal methods have been applied to the problem of parser correctness. Jourdan *et al.* used translation validation to validate the correspondence between a context-free grammar and its LR(1) parser [14]; furthermore, the validator is proven correct in Coq. The weakness of translation validation is that it does not show the correctness of translation, but only the correspondence for a particular run of translation. The decoder DSL on which the bigrammar language is built follows the derivative-based approach, which allows the correctness proof for the translation from a regular expression to its DFA implementation. The notion of derivatives is an old idea from Brzozozwski [7], but which has recently been used for building new lexing and parsing tools [22] and has been generalized to context-free languages [20]. Compared to the usual graph-theoretic NDFAs and DFAs, the primary benefit of working with derivatives is that in most proof assistants it is much easier to reason about algebraic transformations on symbolic expressions (e.g., through induction) than it is to reason about graphs.

Specialized DSLs have been designed for declarative machine-code decoding (e.g, [26]); most of them do not allow a bidirectional syntax. Similar to bigrammars, SLED [23] allows specifying both encoding and decoding in a bidirectional syntax; however, consistency requirements are not formally spelled out and proofs are not represented and machine checked, leaving room for errors. Fox [11] used a domain-specific language called L3 to specify the decoding and encoding of ISAs; however, he used testing to check the consistency between the derived decoders and encoders, instead of formal verification.

There are many general parsing and pretty-printing libraries. However, in general they do not allow a single syntax for both parsing and pretty-printing (other than [2, 13, 24], which we previously discussed). There have also been many bidirectional languages that have been designed for mapping between different data formats including XML processing [5, 15] and pickling/unpickling [16]. They all require the forward and the backward directions form bijections, which are too strong for practical parsing and pretty-printing. Boomerang [4] provides bidirectional lenses that can run backwards. However, a bidirectional lens is motivated by the view-update problem in the database community and the backward direction takes both an abstract value and the original concrete value as input, while a pretty printer takes only the abstract value as input.

8 Conclusions

Our bigrammar DSL allows declarative specification of decoding and encoding of machine instructions, with machine-checked proofs that show the consistency between decoding and encoding. We have shown how to migrate a grammar for x86 decoding to a bigrammar for decoding and encoding. As

future work, we plan to use the bigrammar DSL to specify the decoding/encoding of other machine architectures such as ARM. We also plan to extend the bigrammar DSL to support more expressive grammars such as parsing-expression grammars [10] and context-free grammars.

9 Acknowledgments

References

1. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity. In: 12th ACM Conference on Computer and Communications Security (CCS), pp. 340–353 (2005)
2. Alimarine, A., Smetsers, S., van Weelden, A., van Eekelen, M.C.J.D., Plasmeijer, R.: There and back again: arrows for invertible programming. In: Proceedings of the ACM SIGPLAN Workshop on Haskell, pp. 86–97 (2005)
3. Bergeron, J., Debbabi, M., Desharnais, J., Erhioui, M., Lavoie, Y., Tawbi, N.: Static detection of malicious code in executable programs. International Journal of Requirement Engineering (2001)
4. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: resourceful lenses for string data. In: 35th ACM Symposium on Principles of Programming Languages (POPL), pp. 407–419 (2008)
5. Brabrand, C., Møller, A., Schwartzbach, M.I.: Dual syntax for XML languages. In: 10th International Symposium on Database Programming Languages (DBPL), pp. 27–41 (2005)
6. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: a binary analysis platform. In: Computer Aided Verification (CAV), pp. 463–469 (2011)
7. Brzozowski, J.A.: Derivatives of regular expressions. Journal of the ACM **11**, 481–494 (1964)
8. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: 12th Usenix Security Symposium, pp. 169–186 (2003)
9. The Coq proof assistant. `https://coq.inria.fr/`
10. Ford, B.: Parsing expression grammars: A recognition-based syntactic foundation. In: 31st ACM Symposium on Principles of Programming Languages (POPL), pp. 111–122 (2004)
11. Fox, A.C.J.: Improved tool support for machine-code decompilation in HOL4. In: 6th International Conference on Interactive Theorem Proving (ITP), pp. 187–202 (2015)
12. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Network and Distributed System Security Symposium (NDSS) (2008)
13. Jansson, P., Jeuring, J.: Polytypic compact printing and parsing. In: 8th European Symposium on Programming (ESOP), pp. 273–287 (1999)
14. Jourdan, J.H., Pottier, F., Leroy, X.: Validating LR(1) parsers. In: European Symposium on Programming (ESOP), pp. 397–416 (2012)
15. Kawanaka, S., Hosoya, H.: biXid: a bidirectional transformation language for XML. In: ACM International Conference on Functional programming (ICFP), pp. 201–214 (2006)
16. Kennedy, A.: Pickler combinators. Journal of Functional Programming **14**(6), 727–739 (2004)

17. Kotha, A., Anand, K., Smithson, M., Yellareddy, G., Barua, R.: Automatic parallelization in a binary rewriter. In: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 547–557 (2010)
18. Kroll, J., Dean, D.: BakerSFIeld: Bringing software fault isolation to x64. URL `www.jkroll.com/papers/bakersfield_sfi.pdf`
19. McCamant, S., Morrisett, G.: Evaluating SFI for a CISC architecture. In: 15th Usenix Security Symposium (2006)
20. Might, M., Darais, D., Spiewak, D.: Parsing with derivatives: a functional pearl. In: ACM International Conference on Functional programming (ICFP), pp. 189–195 (2011)
21. Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.B., Gan, E.: Rocksalt: Better, faster, stronger SFI for the x86. In: ACM Conference on Programming Language Design and Implementation (PLDI), pp. 395–404 (2012)
22. Owens, S., Reppy, J., Turon, A.: Regular-expression derivatives re-examined. J. Funct. Program. **19**, 173–190 (2009)
23. Ramsey, N., Fernández, M.F.: Specifying representations of machine instructions. ACM Transactions on Programming Languages and Systems **19**(3), 492–524 (1997)
24. Rendel, T., Ostermann, K.: Invertible syntax descriptions: Unifying parsing and pretty printing. In: Third ACM Haskell Symposium on Haskell, pp. 1–12 (2010)
25. Reps, T., Lim, J., Thakur, A., Balakrishnan, G., Lal, A.: There's plenty of room at the bottom: Analyzing and verifying machine code. In: Computer Aided Verification (CAV), pp. 41–56 (2010)
26. Sepp, A., Kranz, J., Simon, A.: GDSL: A Generic Decoder Specification Language for Interpreting Machine Language. In: Tools for Automatic Program Analysis, pp. 53–64 (2012)
27. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A new approach to computer security via binary analysis. In: Proceedings of the 4th International Conference on Information Systems Security (2008)
28. Tan, G., Morrisett, G.: Bidirectional grammars for machine-code decoding and encoding. In: 8th International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE), pp. 73–89 (2016)
29. Wahbe, R., Lucco, S., Anderson, T., Graham, S.: Efficient software-based fault isolation. In: ACM SIGOPS Symposium on Operating Systems Principles (SOSP), pp. 203–216. ACM Press, New York (1993)
30. Wartell, R., Mohan, V., Hamlen, K.W., Lin, Z.: Securing untrusted code via compiler-agnostic binary rewriting. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 299–308 (2012)
31. Xu, Z., Miller, B., Reps, T.: Safety checking of machine code. In: ACM Conference on Programming Language Design and Implementation (PLDI), pp. 70–82 (2000)
32. Yee, B., Sehr, D., Dardyk, G., Chen, B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., Fullagar, N.: Native client: A sandbox for portable, untrusted x86 native code. In: IEEE Symposium on Security and Privacy (S&P) (2009)