Bidirectional Grammars for Machine-Code Decoding and Encoding

Gang Tan[1] and Greg Morrisett[2]

[1] Pennsylvania State University, gtan@cse.psu.edu
[2] Cornell University, jgm19@cornell.edu

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 a subset of x86-32 that includes around 300 instructions.

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 [5], vulnerability identification [14], and safety verification [16]. As a prominent example, Google's Native Client (NaCl [16]) statically 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, 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, 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.

In our previous work [10], we developed a Domain-Specific Language (DSL) for constructing high-fidelity machine-code decoders. It 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. Finally, we can automatically extract efficient recognizers and parsers from grammars in the DSL, with a proof of correctness about the extraction process based on the semantics.

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 [15]. 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 extract a decoder and an encoder, as well as a machine-checked consistency proof that relates the decoder and encoder. Major contributions of the paper is 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 extracted 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, which demonstrates the practicality of our proposed DSL. In this process we identified a dozen bugs in our previous x86 encoder and decoder, which were written separately and without a correctness proof.
- The bigrammar DSL and its semantics are formally encoded in Coq [6] and all proofs are machine-checked.

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 [7, 1, 12]. 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 [7, 1] required that parsers and pretty-printers are complete inverses (i.e., they form bijections). Rendel and Ostermann [12] 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 Sec. 3.

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 [10], 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:

$$
\begin{array}{l}
\texttt{Inductive grammar} : \texttt{Type} \to \texttt{Type} := \\
\mid \texttt{Char} : \ \forall \texttt{ch} \to \texttt{grammar ch} \\
\mid \texttt{Eps} : \ \texttt{grammar unit} \\
\mid \texttt{Zero} : \ \forall t, \texttt{grammar}\ t \\
\mid \texttt{Cat} : \ \forall t_1\ t_2, \texttt{grammar}\ t_1 \to \texttt{grammar}\ t_2 \to \texttt{grammar}\ (t_1 * t_2) \\
\mid \texttt{Alt} : \ \forall t_1\ t_2, \texttt{grammar}\ t_1 \to \texttt{grammar}\ t_2 \to \texttt{grammar}\ (t_1 + t_2) \\
\mid \texttt{Map} : \ \forall t_1\ t_2, (t_1 \to t_2) \to \texttt{grammar}\ t_1 \to \texttt{grammar}\ t_2 \\
\mid \texttt{Star} : \ \forall t, \texttt{grammar}\ t \to \texttt{grammar}\ (\texttt{list}\ t)
\end{array}
$$

A grammar is parameterized by `ch`, the type for input characters. For machine decoders, the `ch` type contains bits 0 and 1. A value of type "`grammar` t" represents a relation between input strings 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:

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

Grammar "`Char` c" matches strings containing only the character c, and returns that character as the semantic value. `Eps` matches only the empty string and returns `tt` (Coq's unit value). Grammar `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 "`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 $t_1 + t_2$. "`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. `Star` matches zero or more occurrences of a pattern, returning the result as a list.

`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 "`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.

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 defi-

```
Definition INC_p : grammar instr :=
   ("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.

nitions for the notation used.

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

Note that "$g_1 \,||\, g_2$" 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.

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* [4]. 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 [10].

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 Sec. 5, we will present how to generalize to ambiguous grammars.

Ideally, a parser and its corresponding pretty printer should form a bijection [7, 1]: (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 spaces in the AST produced by the parser. Another typical example happens when the union operator (i.e., $||$) is used during parsing. For example, the `INC` grammar in Fig. 1 forgets which branch is taken because of the uses of the union operator. Our x86 decoder grammar has many such uses.

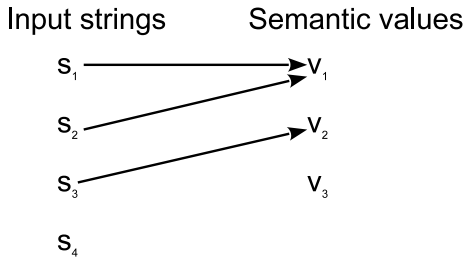
Because information is lost in a typical parser, multiple input strings may be parsed to the same semantic value. Therefore, for such a semantic value, the pretty printer has to either list all possible input strings, or choose a particular one, which may not be the same as the original input string. 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) to a possible value[3] of type t, according to a grammar indexed by t. A pretty printer encodes a semantic value in a possible string, according to a grammar.

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

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

Definition 1. *(Consistency between parsers and pretty printers)*

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

Property 1 says that if a parser turns an input string s to 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).

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

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 [12] 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 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, 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 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 takes 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 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

```
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)
```

Fig. 2: Bigrammar DSL syntax.

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 Sec. 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. We write $\mathrm{rng}(g)$ to be the set $\{\, v \mid \exists s, (s, v) \in [\![g]\!] \,\}$.

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 according 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.[4] Take the case of "`cat` g_1 g_2" 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 returned; in the second case, return $s_1{+}{+}s_2$, which is s_1 concatenated with s_2. For "`map` f_1 f_2 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` f_1 f_2 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 correctness properties in

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

```
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
                                     | inl 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.

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 correctness proof of the pretty printer. The correctness of the parser extracted from a grammar has been shown in the RockSalt paper [10] 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.)*

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

The second theorem is about pretty-printer correctness.

Theorem 2. *(Pretty-printer correctness.)*

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

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

5 Generalization to Ambiguous Grammars

An ambiguous bigrammar g can relate the same input string s to multiple semantic values. A simple example is "`alt` (`char` c) (`char` c)", where c is some character; it relates the single-character string c to "`inl` c" and "`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.)*

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

We write "$v \in$ `parse` $g\ s$" to mean that v is in the list produced by `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 grammars.)*

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

6 Engineering Bigrammars for x86 Decoding and Encoding

We previously defined a 32-bit x86 grammar in the decoder DSL and used the grammar to extract a decoder. We retrofitted the grammar into a bigrammar by adding inverse functions and invertibility proofs to where `Map` is used. In the process we often needed to tweak grammar rules (sometimes with substantial changes) and introduce new constructors to make it easier to develop invertibility proofs and make the encoder more efficient. Using representative examples, we next discuss this experience about how we engineered the x86-32 decoder/encoder bigrammar.

Tighten semantic domains. Most of the changes were because the map functions used in the original 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-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.
```

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
```

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`.

The problem 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 functions when necessary. This makes the inverse function more efficient by avoiding some 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.
```

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 the x86 decoder grammar. 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` $(\mathtt{w}, \mathtt{r})$; 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 =>
        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 semantics: 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 bigrammars, 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 grammar for `MOV` has a dozen cases. In general, if g is the result of combining n bigrammars via union and each bigrammar is of size m, then running the pretty printer on g takes $O(n * m)$ time in the worst case since it may try all n possibilities and pretty printing each bigrammar may take time $O(m)$. 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 function 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 in many instructions, 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 function f_i is of type $t_i \to t$. 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.

Our tactics construct balanced parse trees when combining bigrammars via the disjoint-sum operator; this makes a substantial difference for the speed of checking proofs 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 $O(\texttt{log}\, n)$ number of `inl`s and `inr`s 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 would take five seconds in Coq 8.4 to finish checking the bigrammar

definition for the right-associative combination and take only two seconds for the balanced combination.

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 of eliminating the use of union. 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 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 deal with the situation, we introduced hierarchical abstract syntax for instructions. For instance, floating-point instructions are grouped into a separate instruction type and a bigrammar is developed for floating-point instructions; as there are much fewer floating-point instructions, Coq was able to type check the bigrammar. Then, we have a bigrammar that converts the hierarchical instruction syntax into the flat instruction syntax (and also an inverse function for going the other direction).

Statistics of the x86 bigrammar. 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 in 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 decoder grammar, a separately developed 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
Decoder grammar	2,194
Encoder	2,891
Decoder/Encoder bigrammar	7,254

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 [10], 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 should be `false`.

Before we started migrating the x86 grammar, we wanted to determine whether or not moving from an established grammar and abstract syntax to a bigrammar required extensive changes. Our x86 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.

7 Related Work

Specialized DSLs have been designed for declarative machine-code decoding (e.g, [13]); most of them do not allow a bidirectional syntax. Similar to bigrammars, SLED [11] 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.

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 [7, 1, 12], which we previously discussed). There has also been many bidirectional languages that have been designed for mapping between different data formats including XML processing [3, 8] and pickling/unpickling [9]. They all require the forward and the backward directions form bijections, which are too strong for practical parsing and pretty-printing. Boomerang [2] 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 and context-free grammars.

The bigrammar development and the x86-32 bigrammar are open sourced and available at the following URL: `https://github.com/gangtan/CPUmodels/tree/master/x86model/Model`.

Acknowledgement

We thank the anonymous reviewers for their comments. This research is supported by NSF grants CCF-1217710, CCF-1149211, and CNS-1408826.

References

1. 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)

2. 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)
3. 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)
4. Brzozowski, J.A.: Derivatives of regular expressions. Journal of the ACM 11, 481–494 (1964)
5. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: 12th Usenix Security Symposium. pp. 169–186 (2003)
6. The Coq proof assistant. `https://coq.inria.fr/`
7. Jansson, P., Jeuring, J.: Polytypic compact printing and parsing. In: 8th European Symposium on Programming (ESOP). pp. 273–287 (1999)
8. Kawanaka, S., Hosoya, H.: biXid: a bidirectional transformation language for XML. In: ACM International Conference on Functional programming (ICFP). pp. 201–214 (2006)
9. Kennedy, A.: Pickler combinators. Journal of Functional Programming 14(6), 727–739 (2004)
10. 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)
11. Ramsey, N., Fernández, M.F.: Specifying representations of machine instructions. ACM Transactions on Programming Languages and Systems 19(3), 492–524 (May 1997)
12. Rendel, T., Ostermann, K.: Invertible syntax descriptions: Unifying parsing and pretty printing. In: Proceedings of the Third ACM Haskell Symposium on Haskell. pp. 1–12 (2010)
13. 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)
14. 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)
15. 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)
16. 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) (May 2009)