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:

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

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:

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

Grammar "`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 := `Map` f g
g_1 `$` g_2 := `Cat` g_1 g_2
`literal` $[c_1, \ldots, c_n]$:= (`Char` c_1) `$` $\ldots$ `$` (`Char` c_n)
g_1 `$$` g_2 := ((`literal` g_1) `$` g_2) `@` `snd`
$g_1 \,||\, g_2$:= (`Alt` g_1 g_2)@(λv. `match` v `with` `inl` $v_1 \Rightarrow v_1$ | `inr` $v_2 \Rightarrow v_2$ `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.

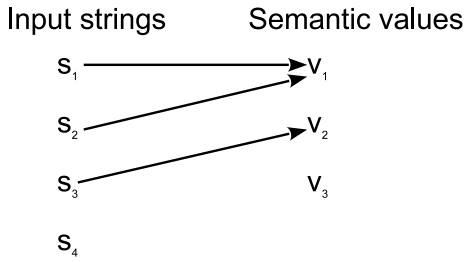
Because information is lost in a typical parser, multiple input strings may be parsed to the same semantic value. Therefore, for such a semantic value, the pretty printer has to either list all possible input strings, or choose a particular one, which may not be the same as the original input string. 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 $\texttt{parse}\ g\ s = \texttt{Some}\ v$*, 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 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.