Bidirectional and executable specifications of machine code decoding and encoding

**Gang Tan, Penn State Univ.**

*Joint work with Greg Morrisett, Cornell*

At **LangSec Workshop, San Francesco, May 24th, 2018**
Machine Code Decoding/Encoding

• Binary code analysis and transformation
  – Analyze binary code for security, for verification, ...
  – Binary rewriting: e.g. insert more instructions into the program for security, for automatic parallelization, ....

• Require machine code decoding
  – From bits to abstract syntax of machine instructions

• Also require machine code encoding
  – From abstract syntax of machine instructions to bits
Decoder Specification Language

- Part of RockSalt work (PLDI 2012)
- Formally encoded in Coq
- Type-indexed parsing combinators for regular grammars
  - Regular grammars: regular expressions + semantic actions
  - Allow transliteration of decoding tables to declarative grammars
  - Then automatically generate executable decoders from grammars, with correctness proofs in Coq
Example Grammar for INC in x86

\[
\text{INC – Increment by 1} \quad \begin{align*}
\text{reg} & \\
\text{reg (alternate encoding)} & \\
\text{memory} & 
\end{align*}
\]

Definition \( \text{INC}_g : \text{grammar} \ \text{instr} \cdot \)

\[
\text{"1111111" \ \$ \ \text{bit} \ \$ \ "11000" \ \$ \ \text{reg} \\
\ \@ \ \text{(fun} \ (w,\text{op}) \ \Rightarrow \ \text{INC} \ w \ \text{op}) \\
\text{"0100" \ \$ \ "0" \ \$ \ \text{reg} \\
\ \@ \ \text{(fun} \ r \ \Rightarrow \ \text{INC} \ \text{true} \ (\text{Reg}\_\text{op} \ r)) \\
\text{"1111" \ \$ \ "111" \ \$ \ \text{bit} \ \$ \ (\text{emodrm} \ "000")} \\
\ \@ \ \text{(fun} \ (w,\text{op}) \ \Rightarrow \ \text{INC} \ w \ \text{op}) 
\]

Alternatives

Decide pattern

Semantic action

Alternatives
Regular Grammar DSL

\textbf{Inductive} grammar : Type \rightarrow Type :=
\begin{align*}
\mid & \text{Char} : \text{char} \rightarrow \text{grammar char} \\
\mid & \text{Eps} : \text{grammar unit} \\
\mid & \text{Zero} : \forall T, \text{grammar } T \\
\mid & \text{Cat} : \forall T U, \text{grammar } T \rightarrow \text{grammar } U \rightarrow \text{grammar } (T \ast U) \\
\mid & \text{Alt} : \forall T U, \text{grammar } T \rightarrow \text{grammar } U \rightarrow \text{grammar } (T + U) \\
\mid & \text{Map} : \forall T U, \text{grammar } T \rightarrow (T \rightarrow U) \rightarrow \text{grammar } U \\
\mid & \text{Star} : \forall T, \text{grammar } T \rightarrow \text{grammar } (\text{list } T)
\end{align*}

\textbf{Infix} “+” := \text{Alt}.  \\
\text{Infix} “$” := \text{Cat}.  \\
\text{Infix} “@@” := \text{Map}.  \\
g1 \mid| g2 := (g1 + g2) @
\begin{align*}
\text{(fun } v \rightarrow \text{match } v \text{ with } \text{inl } v1 \rightarrow v1 \mid \text{inr } v2 \rightarrow v2)\end{align*}

Indexed by the type of semantic values returned by the grammar

Concatenation: return a pair

Alternation: return a tagged union

Map: apply a semantic action

Union: forgetful
Denotational Semantics

[[ ]] : grammar T -> (string * T) -> Prop.
[[Eps]] = {(nil, tt)}
[[Zero]] = {}
[[Char c]] = {(c::nil, c)}
[[Alt g₁ g₂]] = (s, inl v) | (s, v) in [[g₁]] U
               (s, inr v) | (s, v) in [[g₂]]
[[Cat g₁ g₂]] =
               (s₁++s₂, (v₁, v₂)) | (sᵢ, vᵢ) in [[gᵢ]]
[[Star g]] = {(nil, nil)} U
               (s, v) | s≠nil /
               s in [[Cat g (Star g)]]
[[Map g f]] = {(s, f v) | (s, v) in [[g]]}
From Grammars to Parsers

• An operational semantics (interpreter)
  – Derivative-based parsing: old idea due to Brzozowski (1964), revitalized by Reppy et al., and extended by Might
  – Proven correct in Coq w.r.t the denotational semantics

• A parser generator (compiler)
  – Compile to DFA tables with semantic actions
  – Also proven correct in Coq and with termination proofs

• Parser correctness:
  \[(s,v) \in \llbracket g \rrbracket \text{ iff } \text{parse } g \ s = \text{Some } v\]
What about the Encoder?

• Natural idea: have a bidirectional grammar for both decoding and encoding at the same time
  – Derive a decoder and an encoder from the bigrammar

• Benefits
  – Decoder and encoder spec can share parts
  – Can relate the derived decoder and encoder using some “round-trip” theorem
Relating Parsing and Pretty Printing

• Parser: from input strings to semantic values
• Pretty printer: from semantic values to input strings
• Ideally, a parser and its reverse pretty printer should form a bijection
• However, the requirement is too strong in practice
  – Information loss during parsing
  – Loose semantic domains
Information Loss During Parsing

• Parsing often loses information
• For example
  – A parser for source code forgets the amount of white spaces
  – In x86 decoding, multiple bit encoding for the same instruction
• As a result
  – Multiple input strings may be parsed to the same semantic value
  – When inverting such a semantic value, the pretty printer has to choose a specific input string (or list all possible ones)
Loose Semantic Domains

• For uniformity the semantic domain of a parser may include values that cannot be possible parsing results

• An example:
  – x86 instructions takes zero or more operands
  – An operand can be a memory operand, an immediate operand, or a register operand
  – But for a specific instruction, certain combinations of operands are not possible

• Some of these cases could be fixed by introducing tighter domains
  – But in general would make abstract syntax messy

• As a result
  – Pretty printing is partial: cannot invert some semantic values
Relating Input and Output Domains

- Multiple input strings can be parsed to the same semantic value
- Some semantic values may not be possible parsing results
- Parsing is also partial and may reject some input strings
Consistency Properties

• parse: \( \forall T, (\text{bigrammar } T) \rightarrow \text{list char} \rightarrow \text{option } T \)

• pretty-print: \( \forall T, (\text{bigrammar } T) \rightarrow T \rightarrow \text{option (list char)} \)

• Consistency property 1

  If \( \text{parse } g s = \text{Some } v \), then exists \( s' \) so that \( \text{pretty-print } g v = \text{Some } s' \).

\[
\begin{array}{c}
s \quad \text{parse} \quad V \\
\quad s' \quad pp \quad \text{v}
\end{array}
\]

\( s' \) may be different from \( s \)
Consistency Properties

- Consistency property 2
  If pretty-print \( g \ v = \text{Some} \ s \), then parse \( g \ s = \text{Some} \ v \)

Note: it places no obligation when the pretty printer cannot invert \( v \)
Some Related Work

• Haskell community: invertible syntax for both parsing and pretty printing
  – Jansson & Jeuring [ESOP 99]; Alimarine et al. [Haskell 05]; Rendel & Ostermann [Haskell 10]

• Our work is embedded in Coq, with machine-checked correctness proofs
Consistency Properties
In Related Work

- Jansson & Jeuring and Alimarine et al. require bijections; too strong
- Rendel & Ostermann require **partial isomorphisms**
  
  ![Diagram](image)
  
  - Specify an **explicit equivalence relation** and require s and s' in the equivalence relation in Prop 1
- Our approach uses an implicit equivalence relation: all input strings that are parsed to the same semantic value are considered equivalent
A Bigrammar DSL

\[
\text{Inductive bigrammar : Type \to Type :=}
\]
\[
| \text{Char : char \to bigrammar char} & | \text{\ldots}
| \text{Star : \forall T, bigrammar T \to bigrammar (list T)} & | \text{Map: \forall T U, (f1: T \to U) (f2: U \to option T)}
\]
\[
\quad (g: \text{bigrammar T})(pf: \text{invertible}(f1, f2, g)),
\quad \text{bigrammar U}
\]

- Constructors other than Map are reversible and exactly the same as the previous decoder grammar DSL
- Invertible def derived from the consistency properties
Pretty Printer

pretty-print (Char c) =
    λc0. if c=c0 then Some [c] else None

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. v0 <- f2 v; pretty-print g v0

...
Pretty Printer Correctness

• (1) If \((s,v) \in \llbracket \llbracket g \rrbracket \rrbracket\), then exists \(s'\) so that pretty-print \(g v = \text{Some } s'\)

• (2) If pretty-print \(g v = \text{Some } s\), then \((s,v) \in \llbracket \llbracket g \rrbracket \rrbracket\)

• Consistency properties follow from parser and pretty printer correctness
Engineering a Bigrammar for x86 Decoding and Encoding

• Previously
  – Developed a decoder grammar for x86
  – Manually wrote an x86 encoder (not grammar driven)

• Retrofitted the decoder grammar to get a bigrammar

• Unfortunately, had to change many places in the grammar
  – To make it easier to develop invertibility proofs
  – To make the pretty printer more efficient
Overcoming Engineering Challenges

• Eliminating the use of the union operator
  – The use of union results in runtime tests; inefficient
  – Use disjoint sums (tagged unions)

• Reducing proof-checking time
  – First version took hours to finish proof checking
  – Special Coq tactics and dependent types to speed up proof checking

• Tightening semantic domains
  – In the old decoder grammar, many map functions are not surjective, causing loose semantic domains
  – Resulting in runtime tests in the encoder
  – We fixed some of those by having tightened semantic domains

• ...
## x86 and MIPS Bigrammars

<table>
<thead>
<tr>
<th>Bigrammar</th>
<th>Lines of Coq code</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86 Decoder Grammar</td>
<td>2,194</td>
</tr>
<tr>
<td>x86 Encoder (Manually Written)</td>
<td>2,891</td>
</tr>
<tr>
<td>x86 Decoder/Encoder Bigrammar</td>
<td>7,254</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bigrammar</th>
<th>Lines of Coq code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPS Decoder Grammar</td>
<td>342</td>
</tr>
<tr>
<td>MIPS Decoder/Encoder Bigrammar</td>
<td>1,036</td>
</tr>
</tbody>
</table>

- Extracted OCaml code for x86/MIPS decoding and encoding
Speed Comparison: Encoder Generated from the Bigrammar vs. the Manually Developed Encoder

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
<th>Instr count</th>
<th>Bigrammar encoder</th>
<th>Manual encoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>tailf</td>
<td>14KB</td>
<td>2,020</td>
<td>1.19s</td>
<td>2.05s</td>
</tr>
<tr>
<td>pwd</td>
<td>26KB</td>
<td>3,938</td>
<td>2.50s</td>
<td>4.19s</td>
</tr>
<tr>
<td>cat</td>
<td>46KB</td>
<td>7,458</td>
<td>4.99s</td>
<td>8.28s</td>
</tr>
<tr>
<td>ls</td>
<td>103KB</td>
<td>18,377</td>
<td>10.73s</td>
<td>18.92s</td>
</tr>
</tbody>
</table>

- Manual encoder used many literal strings during encoding, resulting in higher memory consumption
  - 70% more memory than the bigrammar encoder
More Info in Papers

• Decoder specification language
  – RockSalt [PLDI 2012]
  – Used the x86 decoder for proving the correctness of a machine code verifier

• Bidirectional decoder/encoder language
  – Conference version [VSTTE 2016]
Future Work: Beyond Regular Grammars

• Parsing (and pretty-printing) are security critical
  – Windows: hundreds of parsers for different file formats; many security-critical bugs were found [GoDefRoiD et al. CACM 2012]

• Beyond regular grammars (dependent grammars, CFG, PEG)
  – [Barthwal and Norrish 09]: verified SLR parsing
  – [Jourdan, Pottier, and Leroy 12]: translation validation for LR(1) parsing
The End

• The bigrammar development in Coq can be found at

https://github.com/gangtan/CPUnomodels