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

Definition INC_g : grammar instr :=
  "1111111" $$ bit $ "11000" $$ reg
  @ (fun (width,r) => INC width (Reg_op r))
| | "0100" $$ "0" $$ reg
  @ (fun r => INC true (Reg_op r))
| | "1111" $$ "111" $$ bit $ (emodrm "000")
  @ (fun (w,op1) => INC w op1).

Alternatives

Decode pattern

Semantic action

Alternatives
Regular Grammar DSL

Inductive grammar : Type -> Type :=
| Char : char -> grammar char
| Eps : grammar unit
| Zero : ∀T, grammar T
| Cat : ∀T U, grammar T -> grammar U -> grammar (T*U)
| Alt : ∀T U, grammar T -> grammar U -> grammar (T+U)
| Map : ∀T U, grammar T -> (T -> U) -> grammar U
| Star : ∀T, grammar T -> grammar (list T)

Infix “+” := Alt.
Infix “$” := Cat.
Infix “@” := Map.

g1 || g2 := (g1 + g2) @
  (fun v => match v with inl v1 => v1 | inr v2 => v2)
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]]} ∪
                {(s, inr v) | (s, v) in [[g₂]]}

[[Cat g₁ g₂]] =
                {(s₁+s₂, (v₁,v₂)) | (sᵢ,vᵢ) in [[gᵢ]]}

[[Star g]] = {(nil, nil)} ∪
             {(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:
  \((s, v) \in [[g]] \iff \text{parse } g \ s = \text{Some } v\)
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: $\forall T, \text{(bigrammar } T) \rightarrow T \rightarrow \text{option } (\text{list char})$

- **Consistency property 1**
  
  If $\text{parse } g \ s = \text{Some } v$, then exists $s'$ so that $\text{pretty-print } g \ v = \text{Some } s'$.
Consistency Properties

• **Consistency property 2**

  If pretty-print $g \, v = \text{Some} \, s$, then parse $g \, s = \text{Some} \, v$

  ![Diagram](image)

  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

\[ s \xrightarrow{\text{parse}} v \xleftarrow{\text{pp}} s' \]

– 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

\[
\text{Inductive bigrammar : Type -> Type :=}
\]

| Char : char -> bigrammar char |
| \dots |
| Star : \forall T, bigrammar T -> bigrammar (list T) |
| Map: \forall T U, (f1: T -> U) (f2: U -> option T) |
| (g: bigrammar T)(pf: invertible(f1, f2, g)), |
| 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

\[ \text{pretty-print (Char } c \text{)} = \lambda c_0. \text{ if } c = c_0 \text{ then Some [c] else None} \]

\[ \text{pretty-print (Alt } g_1 g_2 \text{)} = \lambda v. \text{ match } v \text{ with} \]
\[ | \text{ inl } v_1 \Rightarrow \text{ pretty-print } g_1 v_1 \]
\[ | \text{ inr } v_2 \Rightarrow \text{ pretty-print } g_2 v_2 \text{ end} \]

\[ \text{pretty-print (Map } f_1 f_2 g \text{ pf)} = \lambda v. v_0 <- f_2 v; \text{ pretty-print } g v_0 \]

...
Pretty Printer Correctness

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

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

• 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 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 union operators
  – 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>Bigраммар</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>Bigраммар</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 bigrammars
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/CPUmmodels
More Related Work

• SLED [Ramsey and Fernandez 97]
  – Allows bigrammars for de-/encoding; but no formal consistency requirements and proofs

• Bidirectional XML parsing
  – [Brabrand et al. 05]; biXid [Kawanaka and Hosoya 06]

• Pickling/unpickling [Kennedy 04]

• Boomerang: bidirectional lenses
  – [Bohannon et al. 08]
  – Modelling the view-update problem in DBs; the reverse direction different from pretty printing
Engineering a Bigrammar for x86-32
Decoding and Encoding

• Retrofitted a previously developed 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
• We next discuss some examples
Tightening Semantic Domains

• In the old decoder grammar, many map functions are not surjective, causing loose semantic domains
• Some of these can be fixed by having tightened semantic domains
Example: Parsing 32- or 16-Bit immediates

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.

• Two reverse functions; one for each case
• It produces operands; however, the operand domain contains not just immediate operands
• So reverse functions have to do runtime tests
The Fix

- Producing 32-bit immediates instead; clients of the bigrammar applies Imm_op in their map functions when necessary

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 => ...)
    & _
  end.
The Use of the Union

Definition INC_g : grammar instr :=
  "1111" $$ "111" $$ bit $ "11000" $$ reg
@ (fun (w,r) => INC w (Reg_op r))
|| "0100" $$ "0" $$ reg
@ (fun r => INC true (Reg_op r))
|| "1111" $$ "111" $$ bit $ (emodrm "000")
@ (fun (w,op1) => INC w op1).

• One inefficient way to a bigrammar
  – Add three reverse functions for the three maps
  – For the first one, pattern match the two arguments; if they are
    of the form "(w, (Reg_op r)", return Some (w,r); otherwise,
    return None.
  – A special union bigrammar constructor: try each case and see
    which one succeeds (returns some value)
Eliminating the Use Of Union

• Use disjoin sums (Alt) to combine cases to get a parse tree
  "1111" \[ \text{Alt} \] "111" \[ \text{Alt} \] anybit \[ \text{Alt} \] "11000" \[ \text{Alt} \] reg
  + "0100" \[ \text{Alt} \] "0" \[ \text{Alt} \] reg
  + "1111" \[ \text{Alt} \] "111" \[ \text{Alt} \] anybit \[ \text{Alt} \] ext_op_modrm_noreg "000"

• A single map function from parse trees to instruction arguments

• A single reverse function from arguments to parse trees

• Use tactics to automate the process of generating map and reverse functions as well as the invertibility proof