Unparsed Patterns

Easy User-Extensibility
of Program Manipulation Tools

Nic Volanschi (*my***gcc**)
&
Christian Rinderknecht (Konkuk University)

Introduction

- Source code pattern matching:
  - essential within program manipulation tools
    - program transformation, compilers
    - model checking, code inspection...
    - meta-programming, reflective programming
  - also enables user-extensibility of such tools
- Well-known problem
  - Reducible to tree matching
  - Efficient algorithms

Existing source code patterns

1) Abstract (syntax) patterns
assign($x, add($x, const(int, 1)))

- the user provides a meta-AST
  + easy to implement / built-in
  – has to know both the AST *and* a notation for it

2) Concrete (syntax) patterns
%x = %x + 1

- the user provides a meta-code fragment
  + easy to use
  – usually difficult to implement: extend the parser / port to an advanced tool

A new approach to source code matching

- Unparsed patterns: “%x = %x + 1”
  - concrete syntax => easy to use
  - no parsing => easy to implement
- *How to avoid the parsing?*
  - break established ideas!
  - practice the anamnesis...

Idea #1: Unparse the AST

- Inverse the process:
  - the pattern is a string
  - the AST *t* is flattened as *TXT(t)*
  - match two flat strings
- match “a = a – b * c – d” “%x = %y - %z”
  - {x<--a, y<--a-b*c, z<--d}
  - {x<--a, y<--a, z<--b*c-d} (wrong !!)
- Very imprecise (almost all structure is lost)

Idea #2: Parenthesize the pattern

- Keep all the structure, by:
  - fully parenthesizing the pattern
  - unparsing the AST using parentheses
  - matching two structured strings
- match “(a = ((a – (b * c)) – d))”
  with “(%x = (%y - %z))”
- Trivial algorithm: ~Lisp reader
- Needs *escaped* parentheses!

Evaluation

- Examples:
  - match a = a – b*c – d
    with “%(%x = %(%y - %z%)%)”
  - match case v in 1) exit;; esac
    with “%(case %x in %(%y) %z;;%) esac%)”
- Patterns are unreadable!

A few definitions...

- Program fragment ~ sub-AST
  - Ex: “a – b * c” (but not: “a – b *”)
- Unparsed text of an AST
  - TXT(a – b * c) = “a – b * c”
- Matching
  - t *matches* p $\Leftarrow\!\!=\!\!\Rightarrow$ $\exists \mu = \{x_i \texttt{<--} t_i\}.\ p[x_i \texttt{<--} \mathrm{TXT}(t_i)] = \mathrm{TXT}(t)$
- Unparsed list of an AST
  - LST(a – b * c) = [a, “-”, b * c]

Idea #3: *Lazy* unparsing

- Exploit the AST structure during matching
  - fully parenthesized pattern
  - flatten the AST *t* incrementally as [“(” + LST(t) + “)”]
  - mix match & unparse steps
- match <u>a = a – b * c – d</u> “(%x = (%y - %z))”
  - --> [“(”, <u>a</u>, “=”, <u>a – b * c–d</u>, “)”]
  - --> [“(”, <u>a</u>, “=”, “(”, <u>a – b * c</u>, “–” <u>d</u>, “)”, “)”]
- May use non-escaped parentheses

Idea #4: Use lexical information

- Exploit tokens in the AST
  - pattern = list of characters & variables
  - LST(t) = list of tokens & variables
  - match a token with the pattern prefix
- Advantages
  - no lexical analysis of the pattern
  - language-independent approach for whitespace
    - (tokens may not start with whitespace)

Evaluation of F(0)

- Correct. Complete. Linear.
- Examples:
  - match a = a – b * c – d “(%x = (%y - %z))”
  - match case v in 1) exit;; esac
    with “(case %x in (%y) %z;;) esac)”
- Too many parentheses!

Idea #5: Compute the conflicts

- Parenthesize only “conflicting” subtrees
  - Unparse(t) =
    - “(” + LST(t) + “)” *if conflicting(t)*
    - LST(t) *otherwise*
  - conflicting(t) = (LST(t)=[t'|_])
  - see details in the paper

Evaluation of A(0)

- Correct. Complete. Linear.
- Examples:
  - match <u>a = a – b * c – d</u> “(%x = (%y - %z))”
  - match <u>case v in 1) exit;; esac</u>
    with “case %x in (%y) %z;;) esac”
- May the parentheses be further reduced?

Idea #6: Look ahead

- Eliminate parentheses using lookahead(1)
  - No parentheses in the pattern
  - No parentheses when unparsing the AST
  - Choose match/unparse step using lookahead
- match <u>a = a – b * c – d</u> “%x = %y - %z”
- Greedy algorithm, *incomplete!*
  - If ambiguous lookahead, choose match step
  - Fails on the pattern: “%x = %y - %z - %w”

Combine all ideas

- Algorithm ES(1): complete, reduced parentheses
  - Parenthesize conflicting constructs in pattern
  - Unparse with no parentheses
  - Escaped parentheses
  - Use lookahead(1)
- More complicated predicate conflicting()

Evaluation of ES(1)

- Correct. Complete. Linear.
- Examples:
  - match <u>a = a – b * c – d</u> “%x = %y - %z”
  - match <u>a = a – b * c – d</u> “%x = %(%y - %z%) - %w”
  - match <u>case v in 1) exit;; esac</u>
    with “case %x in %y) %z;; esac”

Discussion

- New approach
  - Without parsing, nor lexing
  - An (open) family of algorithms
  - Lazy unparsing + Tokens + Parenthesizing + computing conflicts + Lookahead
- Advantages
  - very light approach
  - language-independent
  - Ideal for extending *legacy* tools

Discussion

- Pre-requisites:
  - The source of a tool (including a grammar)
  - An unparsing function LST() --- may be generated!
- Limitations
  - A few parentheses are needed
  - Parentheses unveil some of the AST structure
  - Not applicable for rewriting
- Open question
  - What is the minimum amount of parentheses?

Prototypes

- Checking compiler: *my***gcc**
  - implements user-defined checks (~1KLOC)
  - based on unparsed patterns
  - development branch “graphite”
  - available at http://mygcc.free.fr
- AST matching library: matchbox
  - In C: engine=500LOC
  - available at: http://mypatterns.free.fr

Thank you

Questions?

A family of matching algorithms

- Input:
  - match(t : tree, pattern : string) : FAIL/OK(μ)
- Expressed by rewriting of “states”
- State = tuple:
  - <stack : list(tree), pattern : string, μ : subst>
- Initial/final state:
  - <[t], “pattern”, {}> --?--> <[], “”, μ>

Algorithm F(0)

- State rewriting algorithm:
  - <stack : list(tree), pattern : string, μ : subst>
  - <[t], “pattern”, {}> --?--> <[], “”, μ>

<[a = a–b*c–d], “(%x = (%y - %z))”, {}> -->

<[“(” , a, “=”, a–b*c–d, “)”],
“(%x = (%y - %z))”, {}> -->

<[a – b*c – d, “)”], “(%y - %z))”, {x<--a}> -->

<[“(”, a-b*c, “–”, d, “)”, “)”],
“(%y - %z))”, {x<--a}> -->

<[], “”, {x<--a, y<--a-b*c, z<--d}> --> OK.

Related work

- SDF, Refine, ...
  - Grammar formalisms generating concrete patterns
- Jmatch, Pizza, Tom, Scala
  - Extensions/Langages with specific parsed patterns
- Scruple: Multi-language (parsed) patterns
- PADS: Types --> parsers + matchers
- MatchO: Classes Java pour match + Antlr
- (StringTemplates: “Unparser generator”)