Interfacing ITP to the Real World

Daniel Kroening Philipp Rümmer Georg Weissenbacher

Oxford University Computing Laboratory

ITP Workshop
MSR Cambridge
25 August 2009

Motivation

We would like to use ITP to reason about

- Software
- Hardware
- Control systems

The Weakest Link

There is a potential semantic gap. These are typically given as

- Software: C, C++, Java, maybe UML State Machine Diagrams
- Hardware: Verilog, VHDL
- Control systems: Simulink, ...

✘ITPs don't accept these as inputs. Semantics?

Possible Answers

A. Don't

- Instead: model in ITP's language, and then refine to target system
  (e.g., B, PVS $\rightarrow$ Verilog, ...)

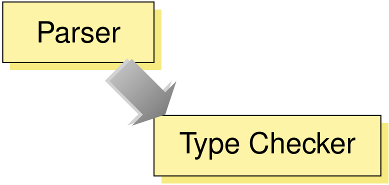
✔ Translation may be buggy, but this is usually a small tool

✔ Semantics question can be limited to a small subset of the target langauge

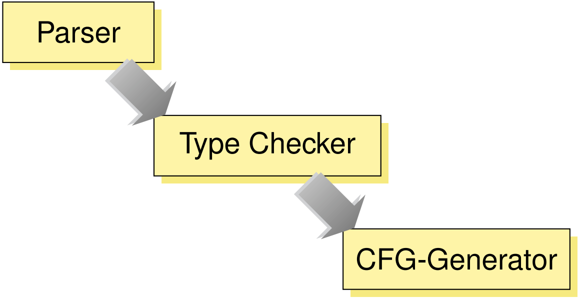
Models from Source Code: Overview

Parser

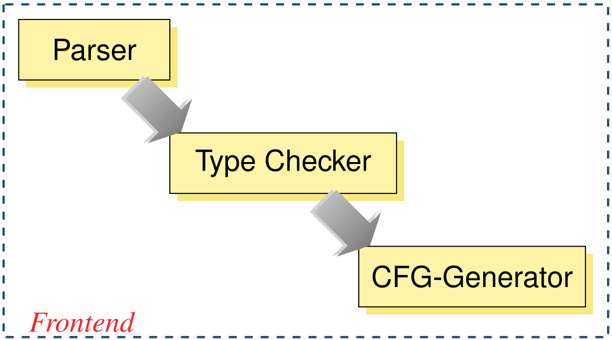
Models from Source Code: Overview



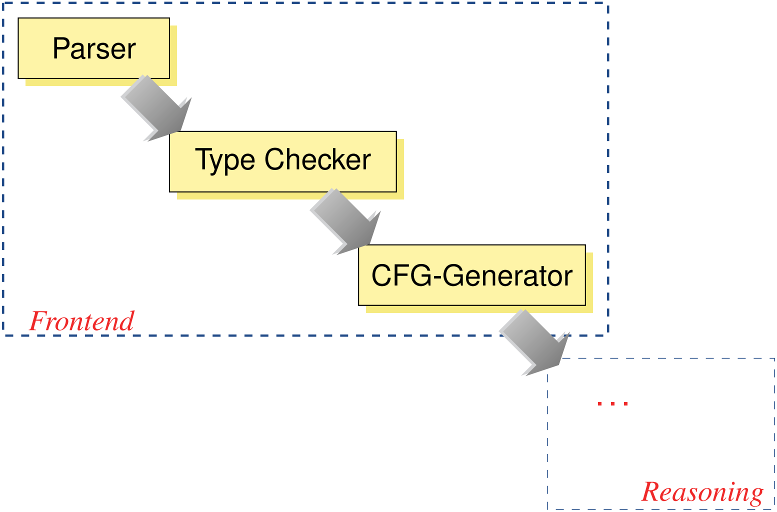
Models from Source Code: Overview



Models from Source Code: Overview



Models from Source Code: Overview



Scanner and Parser

Some overlap with compiler course here.

- A program is a sequence of tokens, which follows a grammar.
- A token is a sequence of characters drawn from an alphabet.

Tokenization

A scanner (lexical analyzer) turns a sequence of characters into a sequence of tokens.
Example: `flex`.

```
digit          [0-9]
octdigit       [0-7]
hexdigit       [0-9a-fA-F]
letter         ([A-Z]|[a-z])
identifier     (({letter}|"_")({letter}|{digit}|"_")*)
integer        {digit}+
decinteger     [1-9]{digit}*
octinteger     "0"{octdigit}*
hexinteger     "0"[xX]{hexdigit}+
decinteger_u   {decinteger}[uU]
octinteger_u   {octinteger}[uU]
hexinteger_u   {hexinteger}[uU]
```

Grammars

- Grammars are typically given in *Backus Normal Form* (BNF)
- Distinguishes terminals (from scanner) and non-terminals

Example from ISO/IEC 9899:1999 (ANSI–C)

(6.5.1) *primary-expression:*

identifier
constant
string-literal
(*expression* **)**

(6.5.2) *postfix-expression:*

primary-expression
postfix-expression **[** *expression* **]**
postfix-expression **(** *argument-expression-list*$_{opt}$ **)**
postfix-expression **.** *identifier*
postfix-expression **->** *identifier*
postfix-expression **++**
postfix-expression **--**
(*type-name* **)** **{** *initializer-list* **}**
(*type-name* **)** **{** *initializer-list* **,** **}**

(6.5.2) *argument-expression-list:*

assignment-expression
argument-expression-list **,** *assignment-expression*

Example: bison Grammar

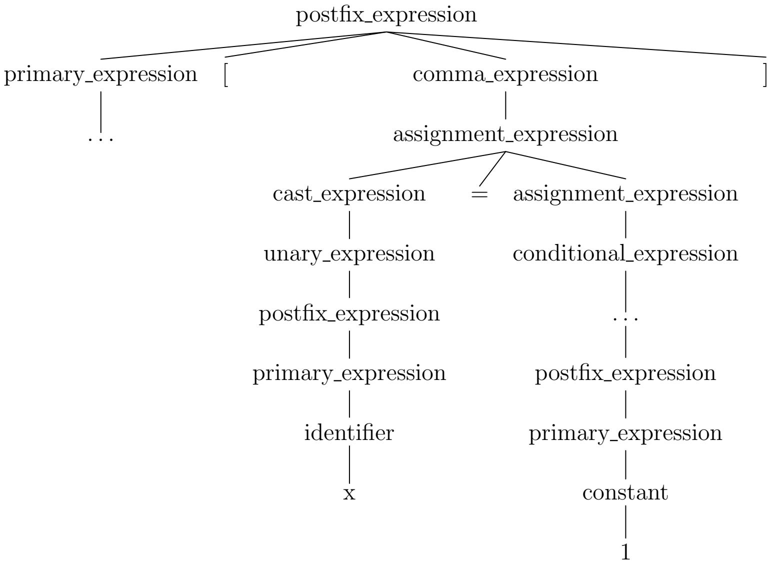
```
primary_expression:
    identifier
  | constant
  | '(' comma_expression ')'
  ;

postfix_expression:
    primary_expression
  | postfix_expression '[' comma_expression ']'
  | postfix_expression '(' ')'
  | postfix_expression '(' argument_expression_list ')'
  | postfix_expression '.' member_name
  | postfix_expression TOK_ARROW member_name
  ...
```

Parse Trees

Each rule is typically associated with some code fragment that constructs a parse tree.

- The internal nodes are non-terminals of the grammar
- The leaf nodes are terminals of the grammar

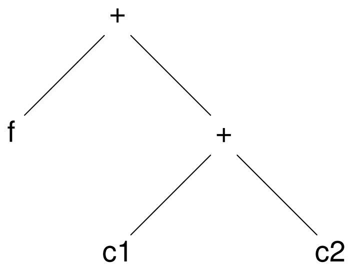
Parse Trees


postfix_expression
primary_expression
[
comma_expression
]
. . .
assignment_expression
cast_expression
=
assignment_expression
unary_expression
conditional_expression
postfix_expression
. . .
primary_expression
postfix_expression
identifier
primary_expression
x
constant
1

Type Checker

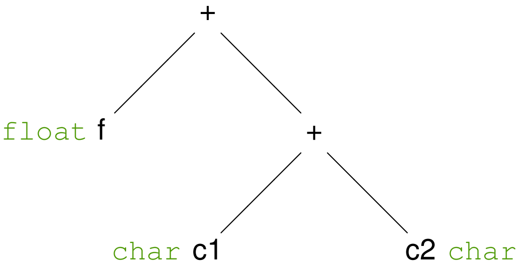
- Parse tree to symbol table: maps identifiers to types

- Expressions are annotated with types
  $\rightarrow$ promotion rules in the case of C



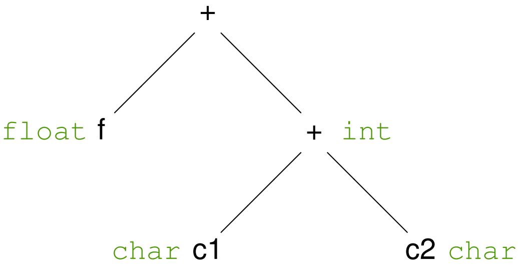
Type Checker

- Parse tree to symbol table: maps identifiers to types
- Expressions are annotated with types
  → promotion rules in the case of C



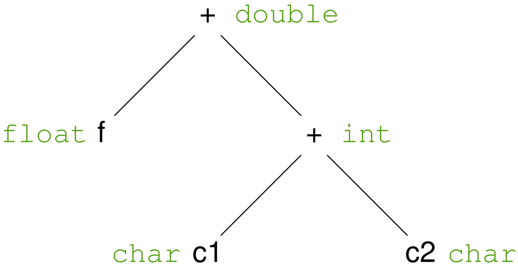
Type Checker

- Parse tree to symbol table: maps identifiers to types
- Expressions are annotated with types
  $\rightarrow$ promotion rules in the case of C



Type Checker

- Parse tree to symbol table: maps identifiers to types
- Expressions are annotated with types
  $\rightarrow$ promotion rules in the case of C



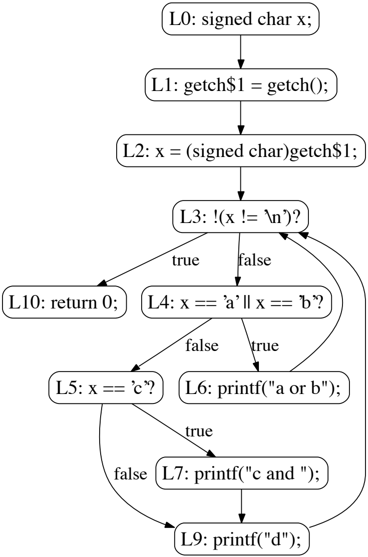
Control Flow Graph

- The code of each procedure is converted to a Control Flow Graph (CFG)

- Think of this as a program with GOTOs

Control Flow Graph

```
int main( void ) {
  char x;
  x = getch();
  while (x!='\n') {
    switch(x) {
      case 'a':
      case 'b':
        printf("a or b");
        break;
      case 'c':
        printf("c and ");
        /* fall-through */
      default:
        printf("d");
        break;
    }
  }
  return 0;
}
```



Where and How?

- All of this can be done inside the ITP
- A tool like ACL2 might even be fast
- Or: do externally, and grab any of the intermediate stages (possibly verify the external tool)

STL

- Standard Template Library
- Encapsulates complex data structures and algorithms

```
typedef std::hash_map<
  std::string, symbolt, string_hash> symbolst;
...

typedef std::vector<nodet> nodest;
```

STL

- "Interesting" programs using STL
  have $>$1000 data structures

- STL implementation highly complex and optimized
- Don't want to verify STL together with program

- Let's *assume the STL is correct*,
  and let's map these to theorem prover types!

Simulink

- We have models from Airbus, Ford, ...

- This looks like a dataflow description, but it isn't
- This looks like there are modules, but there aren't
- This looks like there is concurrency, but there isn't

$\longrightarrow$ Use sequential semantics

✔ We are building a converter to CFGs