Semantyka i weryfikacja programów

Bartosz Klin

(slajdy Andrzeja Tarleckiego)

Instytut Informatyki
Wydział Matematyki, Informatyki i Mechaniki
Uniwersytet Warszawski

`http://www.mimuw.edu.pl/~klin` pok. 5680

`klin@mimuw.edu.pl`

Strona tego wykładu:

`http://www.mimuw.edu.pl/~klin/sem19-20.html`

Program Semantics & Verification

Bartosz Klin

(slides courtesy of Andrzej Tarlecki)

Institute of Informatics
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw

http://www.mimuw.edu.pl/~klin office: 5680
klin@mimuw.edu.pl

This course: http://www.mimuw.edu.pl/~klin/sem19-20.html

LR(k) parsing

- The word is processed from the left
- The rightmost derivation is produced

Here the pushdown automaton works a bit differently:

- Stack alphabet $\Gamma = N \cup T$, intial stack: empty, accept if at the end the stack is the starting nonterminal
- Two actions to choose from at each step:
  - shift: push the symbol from input to the stack, read the next symbol
  - reduce: if the top of the stack contains the right-hand side of some production, replace it with the left-hand side (nonterminal)

How to choose the action?

- This is more flexible than LL parsing, because we do not have to commit to a production until we see the entire right-hand side of it

Example

$$S \to S + P \mid P$$
$$P \to P * F \mid F$$
$$F \to (S) \mid n$$

Stack	Input	Action
ϵ	$1 + (2 * 3)$	shift
1	$+ (2 * 3)$	reduce $F \to n$
F	$+ (2 * 3)$	reduce $P \to F$
P	$+ (2 * 3)$	reduce $S \to P$
S	$+ (2 * 3)$	shift
$S +$	$(2 * 3)$	shift
$S + ($	$2 * 3)$	shift
$S + (2$	$* 3)$	reduce $F \to n$
$S + (F$	$* 3)$	reduce $P \to F$
$S + (P$	$* 3)$	shift (!)

Stack	Input	Action
$S + (P *$	$3)$	shift
$S + (P * 3$	$)$	reduce $F \to n$
$S + (P * F$	$)$	reduce $P \to P * F$
$S + (P$	$)$	reduce $S \to P$
$S + (S$	$)$	shift
$S + (S)$		reduce $F \to (S)$
$S + F$		reduce $P \to F$
$S + P$		reduce $S \to S + P$
S		

LR parser states

- Information about things read so far will be stored in parser states.
- A parser state is a closed set of handles such as:

$$S \to S \bullet + P$$

  That is, a handle is a production with one position marked. Intutively, the handle says: it is possible that we are just reading this production and we have read up to the marked place so far.

- The closure operator is: if $A \to w \bullet Bv$ is in the state and $B \to u$ is a production then $B \to \bullet u$ is in the state too.

$$S \to S + P \mid P$$
$$P \to P * F \mid F$$
$$F \to (S) \mid n$$

Example

$$1: \quad S \to \bullet S + P \mid \bullet P$$
$$P \to \bullet P * F \mid \bullet F$$
$$F \to \bullet(S) \mid \bullet n$$

$$2: \quad S \to S \bullet + P$$

$$3: \quad S \to P \bullet$$
$$P \to P \bullet * F$$

$$4: \quad P \to F \bullet$$

$$5: \quad S \to \bullet S + P \mid \bullet P$$
$$P \to \bullet P * F \mid \bullet F$$
$$F \to (\bullet S) \mid \bullet(S) \mid \bullet n$$

$$6: \quad P \to n \bullet$$

$$7: \quad S \to S + \bullet P$$
$$P \to \bullet P * F \mid \bullet F$$
$$F \to \bullet(S) \mid \bullet n$$

$$8: \quad P \to P * \bullet F$$
$$F \to \bullet(S) \mid \bullet n$$

$$9: \quad S \to S \bullet + P$$
$$F \to (S \bullet)$$

$$10: \quad S \to S + P \bullet$$
$$P \to P \bullet * F$$

$$11: \quad P \to P * F \bullet$$

$$12: \quad P \to (S) \bullet$$

The control table, part one

We build a transition relation between states, labeled with symbols from $T \cup N$:

$$\delta(p,x) = Closure\{A \to wx\bullet v \ : \ A \to w\bullet xv \in p\}$$

δ	1	2	3	4	5	6	7	8	9	10	11	12
S	2				9							
P	3				3		10					
F	4				4		4	11				
$+$		7							7			
$*$			8							8		
(	5				5		5	5				
)									12			
n	6				6		6	6				

GOTO part

ACTION part

The LR(0) automaton

- a pushdown automaton; on the stack, parser states alternate with symbols
- starting from the initial parser state on the stack
- If a state p is at the top of the stack, the current input symbol is a and $\delta(p, a) = q$ then push a, q to the stack (the shift action)
- But how to reduce?
- Add reduction actions to the control table: if P is a production $A \to w$, p is a state and $A \to w\bullet \in p$ then add an action R_P to $\delta(p, a)$ for every terminal a.
- If a state p is at the top of the stack, the current input symbol is a and $\delta(p, a) = R_P$ for $P = A \to w$, then:
  - pop $|w|$ pairs from the stack, revealing some state q,
  - push $A, \delta(q, A)$ to the stack.

The control table ctd.

$1 : S \to S + P$
$2 : S \to P$
$3 : P \to P * F$
$4 : P \to F$
$5 : F \to (S)$
$6 : F \to n$

δ	1	2	3	4	5	6	7	8	9	10	11	12
S	2				9							
P	3				3		10					
F	4				4		4	11				
$+$		7	R_2	R_4		R_6			7	R_1	R_3	R_5
$*$			$R_2, 8$	R_4		R_6				$R_1, 8$	R_3	R_5
$($	5		R_2	R_4	5	R_6	5	5		R_1	R_3	R_5
$)$			R_2	R_4		R_6			12	R_1	R_3	R_5
n	6		R_2	R_4	6	R_6	6	6		R_1	R_3	R_5

Shift/reduce conflicts!

Reduce/reduce conflicts also happen...

The SLR(1) automaton

- "Simple LR(1)"
- A way to avoid some shift/reduce and reduce/reduce conflicts.
- Intuition: LR(0) reduces whenever it can. In SLR(1), we reduce only when it has a basic chance of working.
- Add reduction actions to the control table: if P is a production $A \to w$, p is a state and $A \to w\bullet \in p$ then add an action R_P to $\delta(p, a)$ for every terminal a such that $a \in \mathsf{Follow}(A)$.
- The rest is as before

The SLR(1) control table

$1: S \to S+P$

$2: S \to P$

$3: P \to P*F$

$4: P \to F$

$5: F \to (S)$

$6: F \to n$

δ	1	2	3	4	5	6	7	8	9	10	11	12
S	2				9							
P	3				3		10					
F	4				4		4	11				
$+$		7	R_2	R_4		R_6			7	R_1	R_3	R_5
$*$			8	R_4		R_6				8	R_3	R_5
$($	5				5		5	5				
$)$			R_2	R_4		R_6			12	R_1	R_3	R_5
n	6				6		6	6				

Follow$(S) = \{+,)\}$, so

no conflicts.

Example revisited

$$S \to S + P \mid P$$
$$P \to P * F \mid F$$
$$F \to (S) \mid n$$

Stack	Input	Action
${}_1$	$1 + (2 * 3)$	6
${}_1 1_6$	$+ (2 * 3)$	R_6
${}_1 F_4$	$+ (2 * 3)$	R_4
${}_1 P_3$	$+ (2 * 3)$	R_2
${}_1 S_2$	$+ (2 * 3)$	7
${}_1 S_2 +_7$	$(2 * 3)$	5
${}_1 S_2 +_7 (_5$	$2 * 3)$	6
${}_1 S_2 +_7 (_5 2_6$	$* 3)$	R_6
${}_1 S_2 +_7 (_5 F_4$	$* 3)$	R_4
${}_1 S_2 +_7 (_5 P_3$	$* 3)$	8 (!)

Stack	Input	Action
${}_1 S_2 +_7 (_5 P_3 *_8$	$3)$	6
${}_1 S_2 +_7 (_5 P_3 *_8 3_6$	$)$	R_6
${}_1 S_2 +_7 (_5 P_3 *_8 F_{11}$	$)$	R_3
${}_1 S_2 +_7 (_5 P_3$	$)$	R_2
${}_1 S_2 +_7 (_5 S_9$	$)$	12
${}_1 S_2 +_7 (_5 S_9)_{12}$	$\#$	R_5
${}_1 S_2 +_7 F_4$	$\#$	R_4
${}_1 S_2 +_7 P_{10}$	$\#$	R_1
${}_1 S_2$	$\#$	

Expressive power

- More general ideas: LR(1)
- Here handles include the next symbol to be read, e.g:

$$(S \to S \bullet + P,\ a)$$

- LR(1) automata are huge, so LALR(1) is a simplified version
- LR(0) $\subset$ SLR(1) $\subset$ LALR(1) $\subset$ LR(1) $\subset$ LR(2) $\subset \cdots \subset$ LR(k) $\subset \cdots$
- Theorem (Knuth): Every deterministic context-free language has an LR(k) grammar, and even a (bigger) LR(1) grammar.
- There is a language in LR(1) but not in LL(k) for any k:

$$\{a^i b^j \ :\ i \geq j\}$$

Certified compilers

Do you trust your compiler?

- Most software errors arise from source code
- But what if the compiler itself is flawed?
- Testing is immune to this problem, since it is applied to target code

But good luck identifying the bug!

- Formal verification is harmed: even if the source program is proved correct, the compiled one may be wrong.
- Common practice for safety-critical systems:
  - turn off most optimizations
  - perform human audit of target code

It this paranoia?

- In 1995, 12 out of 20 commercially available C compilers were found to have flaws in optimizing integer division.
- In 2008, all 13 tested C compilers had flaws in dealing with volatile variables. Some GCC versions optimized this out:

```
extern volatile int WATCHDOG;
void reset_watchdog() { WATCHDOG = WATCHDOG;}
```

- CSmith: a tool for testing C compilers with randomly generated programs. In 2011, it found 325 errors in GCC, LLVM and other mainstream compilers.
- GCC shipped with Ubuntu 8.04.1 had this wrong on all optimization levels:

```
int foo(void) {
  signed char x = 1;
  unsigned char y = 255;
  return x > y; }
```

Solution I: Target code validation

After compilation, prove that the target code is equivalent to the source code.

Problems:

- Formal semantics of both source and target languages must be provided.
- Program equivalence is almost always undecidable.
- Typically needs human assistance.
- Even if it works, it is very time-consuming.

Solution II: Proof-carrying code

Augment target code with a formal proof of its desirable properties.

Advantages:

- Source code semantics is not needed
- Very robust framework, exending beyond compiler correctness
- Small burden on the user: checking proofs is not very costly
- Great for mobile code

Problems:

- Does not really check compiler correctness
- Huge burden on the developer

Solution III: Certified compiler

Formally prove that the compiler is correct.

Advantages:

- No burden on the developer or on the user
- Guarantees that source-code analyses apply to target code
- One-off effort

Problems:

- Formal semantics of both source and target languages must be provided.
- Huge burden on the compiler developer

CompCert

- A certified C compiler
- Developed since 2005 at INRIA Paris (principal: Xavier Leroy)
- Free for non-commercial use
- Licenses sold for commercial use

Main ingredients:

- Small-step operational semantics of the source language
- Small-step operational semantics of the target language
- A compiler written in (a functional sublanguage of) Coq
- A proof of correctness in Coq
- A translation from the functional sublanguage of Coq to Caml

Languages

The source language:

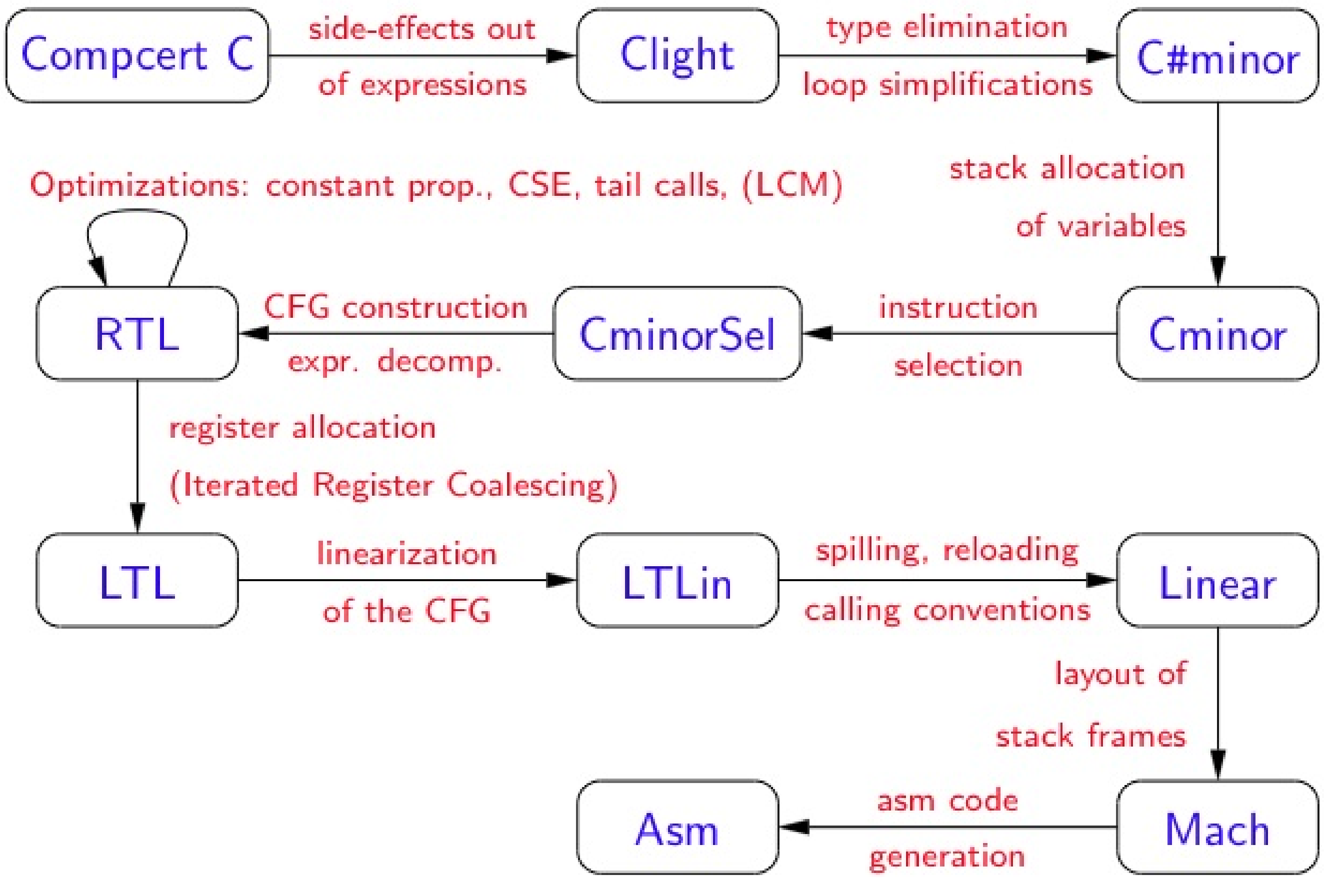
- A large subset of C
- No `longjmp` or `setjmp`
- Only structural `switch`, no "Duff's device"
- No variable-length array types

Supported target architectures:

- PowerPC
- RISC-V, ARMv8 64-bit
- Intel x86, 32- and 64-bit

The compiler structure

- 20 passes, 11 intermediate languages, each with its own small-step operational semantics



Example intermediate language

RTL: Register Transfer Language

$$
\begin{array}{rl}
i ::= & \mathtt{nop}(l) \mid \mathtt{op}(op, \vec{r}, r, l) \mid \mathtt{load}(k, m, \vec{r}, r, l) \mid \mathtt{store}(k, m, \vec{r}, r, l) \\
\mid & \mathtt{call}(sig, (r \mid id), \vec{r}, r, l) \mid \mathtt{tailcall}(sig, (r \mid id), \vec{r}, r) \\
\mid & \mathtt{cond}(b, \vec{r}, l_t, l_f) \mid \mathtt{return}(r)
\end{array}
$$

A CFG (Control Flow Graph) is a finite map $g : l \mapsto i$

Example semantic rule:

$$
\frac{g(l) = \mathtt{op}(op, \vec{r}, r, l') \qquad \mathtt{eval\_op}(G, \sigma, op, R(\vec{r})) = v}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \to \mathcal{S}(\Sigma, g, \sigma, l', R[r \mapsto v], M)}
$$

Example transformation

RTL to LTL: register allocation

- Purpose: divide pseudo-registers r into actual registers and stack allocations
- First step: back-propagation to check which r is alive in which point l
- Two pseudo-registers interfere if they are both alive at some point
- If r and r' do not interfere, they can be stored in the same register
- Coloring pseudo-registers with registers: an NP-complete problem, but good heuristics exist

Property to prove:

Each transition of program is "simulated"
by transitions of the transformed program

CompCert performance

- no errors uncovered so far (after years of attempts)
- compilation process: approx. 2 times slower than GCC with no optimization
- compiled code: approx. 10% slower than GCC with level 1 optimization, 20% slower than GCC with level 2 optimization
- main reason: lack of fancy loop optimizations etc.

What can go wrong?

Unverified parts of the compilation process:

- on the front end: preprocessing
- on the back end: assembling and linking

The verification process itself:

- What if one or both semantics are wrong?
- What if the translation from the functional sublanguage of Coq to Caml is wrong?
- What if the Caml compiler is wrong?
- What if the Coq proof system is wrong?
- What if mathematics is inconsistent?