Guards and Guarantees for Programming

David Schmidt

Kansas State University

Tools: hammers, paper cutters, lawn mowers

- Tools are designed for a speci£c purpose; if misused (intentionally or unintentionally), they can cause physical harm.
- To provide protection against unintentional misuse, tools come with guards.
- To promote proper use, tools come with instructions and guarantees of performance with proper use.

A programming language is a tool

- Like a physical tool, it can be misused, unintentionally or intentionally.
- Like a physical tool, a programming language should come with guards that protect against unintentional (and most intentional) misuse.
- Like a physical tool, a programming language should come with instructions and guarantees of proper performance.

Typical language guards and guarantees

Forms of guards:

- static (pre-execution) checks: declaration checks, type checks, signature (module-interface) compatibility checks, uninitialization checks, other safety checks
- dynamic (execution) checks: virtual-method message checks, type-cast checks, array-bounds checks, dereference and jump checks, liveness checks, and other safety checks

Forms of instructions:

- language descriptions in English verbiage)-:
- syntax de£nitions in grammatical format
- semantic de£nitions in operational/denotational/axiomatic format

Forms of guarantees:

- ???

Typical dynamic checks: trapping malicious actions

- `... r -> f ...` Does object `r` have value null ? Does `r` possess a £eld named `f` ?
- `a[i + 2] = ... ;` Does the value of `i + 2` fall in the range of `a`'s indices ?
- `goto x;` Is `x`'s value a location in this user's program partition ?
- `assert{ exponentiation == loop_counter * base }` Are the run-time values of the variables supporting the invariance (safety) check ?

Typical static checks: identifying dubious phrases

- (I) `... Math.sqrt("hello") ...` Is this an appropriate argument to the function ? (II) `... Math.sqrt(2, 9) ...` Is this an appropriate quantity of arguments to the function ?
- (I) `int x = "hello";` Is this an appropriate assignment to the variable ? Will the value £t into the storage cell ? (II) `int pi = 3.14159;` Is this an appropriate assignment to the variable ? Will the value £t into the storage cell ?
- `... a[x() + y() * z()] ...` Does the value of the index expression fall in the range of `a`'s indices ? Does the absence of brackets in the index expression make it ambiguous ? (The functions `x, y, z` might possess side effects.)

Can a static checker identify these situations as well as those on the previous slide? Some checkers (e.g., LC-Lint, ESC-Java) can.

Assessment of dynamic checks

✘ Requires extra code, either embedded into the application (e.g., compiler embeds array bounds check at each indexing) or coded within an interpreter that monitors the application (e.g., Java byte-code interpreter in web browser) or “woven” into the application (e.g., AspectJ and other aspect-oriented-programming weavers).

This increases code size and slows execution.

✔ Prevents almost all inadvertant misuses and many malicious security violations. (Standard C-trick, responsible for 50% of all security attacks: deliberately over-index an array (called a “buffer overrun”) to the base of the application’s C-implemented activation-record stack, resetting the application’s return address so its exit jumps to malicious code but with OS-level read-write privileges.)

Assessment of static checks

- ✘ Occasionally rejects sensible programs that exploit programming shortcuts in control/data representation — forces conformance to standard and perhaps “boring” programming style.
- ✔ Protects a programmer from making foolish errors when tired or confused and protects the programmer from writing patterns that can be exploited by a malicious user/hacker.
- ✔ Forces the programmer to question her/his programming technique and forces her/him to study the language’s instructions.

History

Dif£cult experiences with machine programming in the 1940's and 1950's caused John Backus to introduce both static and dynamic checks on arrays into Fortran. Their great success caused their extension in subsequent languages, notably the Algols, Pascal and Modula, where modern type checking matured. Even the fundamentally dynamic languages Lisp and Snobol employed dynamic checks.

But these experiences were forgotten in the 1980's, when C-programs were written for mini- and micro-computers, whose limited storage and processor power made the user pay a performance penalty for such guards. Software reliability decreased, and malicious users regularly exploit ¤awed software developed in "un-guarded" languages.

The consequences have been severe, and the software community is relearning what Backus knew in the 1950's.

Instructions for programming languages

Unfortunately, most programming languages do not possess a comprehensible set of instructions, that is, a formal language de£nition. The situation is worse today that it was in the 1970's and 1980's !

A formal language de£nition should, in principle, consist of

- syntax de£nition: lexicon (vocabulary) and grammatical rules (sentence structure)
- semantics de£nition: description of the operation of the syntactically legal programs
- pragmatics description: how to employ the language to best effect

Sample syntax de£nition for a baby assignment language

Lexicon:

$n \in \mathrm{Numeral} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^*$

$v \in \mathrm{Var} = \{a, b, c, d, \ldots, z\}^*$

Punctuation and operators: $@, :=, +, ;$

Grammar:

$e \in \mathrm{Expr}$

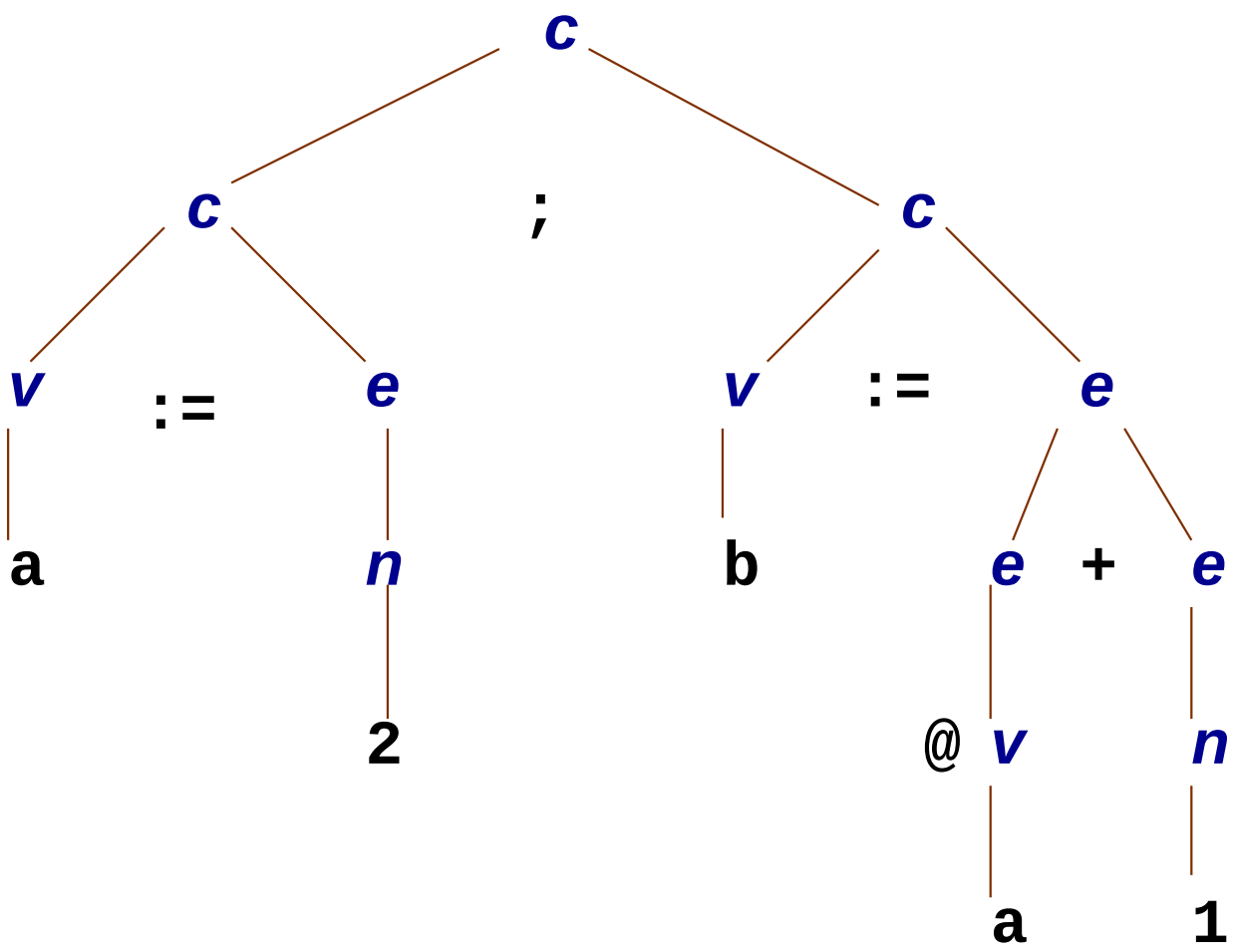
$$e \quad ::= \quad @v \quad | \quad n \quad | \quad e_1 + e_2$$

$c \in \mathrm{Command}$

$$c \quad ::= \quad v := e \quad | \quad c_1; c_2$$

The grammar gives a precise de£nition how programs appear.

Example syntax tree for `a:= 2; b:= @a + 1`



Sample (big-step) operational semantics

Data structures: $s \in StorageVector = (Var \times Nat)^*$

$n \in Nat = \{0, 1, 2, ...\}$

Expression computation rule format: $s \vdash e \to n$

$$s \vdash @v \to lookup(v, s) \qquad x \vdash n \to n$$

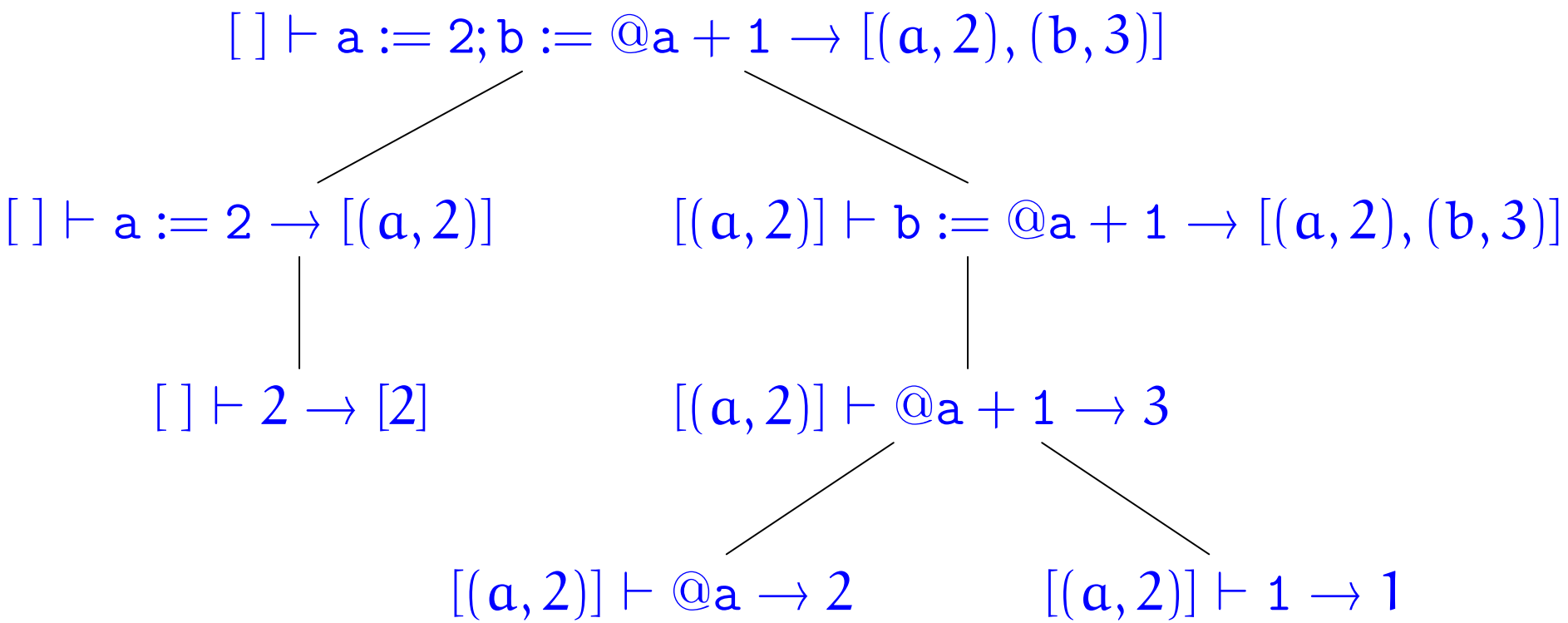
$$\frac{s \vdash e_1 \to n_1 \quad s \vdash e_2 \to n_2}{s \vdash e_1 + e_2 \to add(n_1, n_2)}$$

Command computation rule format: $s \vdash c \to s'$

$$\frac{s \vdash e \to n}{s \vdash v := e \to update(s, v, n)} \qquad \frac{s \vdash c_1 \to s_1 \quad s_1 \vdash c_2 \to s_2}{s \vdash c_1; c_2 \to s_2}$$

Assuming that we de£ne precisely $lookup, update$, and add, the semantics explains precisely the meanings of the phrases.

Example semantics for `a:= 2; b:= @a + 1`



What are a programming language's guarantees ?

A typical "guarantee":

> "This software is sold 'as is,' and no warranty is expressed or implied. The user takes full responsibility for use of aforementioned software and the Manufacturer is not liable for any use that leads to loss of any kind to the user ... blah blah blah ...."

You are a software engineer; you build applications whose input-data formats de£ne a specialized "programming language" (consisting of appropriate sequences of keyboard presses and mouse clicks).

What are the guards, instructions, and guarantees that you offer to your clients ?

Final remark: guards and guarantees are needed for program components that are "used" by other components — this is programming by contract

```
/** factorial computes  n!  for input  n  in the range 0..20.
  * @param n - must be in the range 0..20 <---PRECONDITION - guard
  * @return  n! <---POSTCONDITION - guaranteed if precondition true
  * @throw RuntimeException, if  n < 0  or  n > 20  */
public long factorial(int n)
{ long answer;  // holds the result
  if ( n < 0  ||  n > 20 )  // <---IMPLEMENTS THE GUARD
       { throw new RuntimeException("illegal input"); }
  else { answer = 1;  int count = 0;
         while ( count != n )
           // invariant:  answer == count!  <---SAFETY-CHECK GUARD
           { count = count + 1;  answer = count * answer; }
       }
  return answer;  // assert: answer == n!  <---SAFETY-CHECK GUARD
}
```

A few references

The slides for this talk: `www.cis.ksu.edu/~schmidt/papers`

C.A.R. Hoare. Notes on data structuring. In Structured Programming, O.-J. Dahl, et al., editors. Academic Press, 1972. The origin of modern data types.

Neil Jones and Steve Muchnick. TEMPO: A uni£ed treatment of binding time concepts. Springer Lecture Notes in Computer Science 66, 1978. How guards are introduced into a language and moved from dynamic checks into static checks.

D.A. Schmidt. Denotational Semantics: A Methodology for Language Development W.C. Brown, 1988. One explanation of how to read, write, and use formal language de£nitions.

D. Detlefs, et al. Extended static checking. Compaq SRC Report 159, 1998. A static checker that does many checks considered “dynamic.” John Hatcliff’s CIS771 web page has a nice intro.