Software Verification (6CFU)

- **Program semantics:** small step operational and denotational semantics (2.5CFU)

- **Static analysis**: abstract interpretation and program analysis (3CFU)

- **Software verification tools and applications**: Interproc, Verification of Machine Learning (0.5CFU)

Static program analysis

From Wikipedia, the free encyclopedia
(Redirected from Static code analysis)

Static program analysis is the analysis of computer software that is performed without actually executing programs (analysis performed on executing programs is known as dynamic analysis).[1] In most cases the analysis is performed on some version of the source code, and in the other cases, some form of the object code.

The term is usually applied to the analysis performed by an automated tool, with human analysis being called program understanding, program comprehension, or code review. Software inspections and Software walkthroughs are also used in the latter case.

Contents [hide]

List of tools for static code analysis

From Wikipedia, the free encyclopedia

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed.
(February 2015) (Learn how and when to remove this template message)

This is a list of tools for static code analysis.

Contents [hide]

Polyspace® static code analysis products use formal methods to prove the absence of critical run-time errors under all possible control flows and data flows. They include checkers for coding rules, security vulnerabilities, code metrics, and hundreds of additional classes of bugs.

Polyspace Code Prover

Formally prove the absence of critical run-time errors without executing code

Learn more

Request a Trial

Polyspace Bug Finder

Check coding rules, security standards, code metrics, and find bugs

Learn more

Request a Trial

Goal: to be able to read and fully understand

STATIC VERIFICATION OF DYNAMIC PROPERTIES

Dr. Alain Deutsch
Chief Technical Officer
PolySpace Technologies

ABSTRACT

This paper is a tutorial on the principles and applications of static verification of dynamic properties to development, verification and validation of embedded applications. The topics covered include what static verification of dynamic properties is, how it works, how it can help in verification and validation activities. It will also present an industrial tool for the automatic detection of run-time errors.

Tools and libraries for static analysis and verification

Libraries

- Camllib: addition to OCaml standard libraries.
- Fixpoint: generic fixpoint engine in OCaml.
- APRON: numerical abstract domains library
- Bddapron: higher-level interface to (CUDD) BDDs, manipulaton of mixed Boolean/numerical formulas using BDDs, abstract domains combining BDDs and APRON domains.

Tools

- NBac: Numerical and Boolean Automaton Checker: a verification tool based on dynamic partitioning to verify safety properties of programs or systems. Connected to the LUSTRE synchronous language and Auto symbolic hybrid automata formalism.
- Interproc: an analyzer for an (academic) imperative language with recursion, that infers invariants about numerical variables.
- InterprocStack: an extension of Interproc, with both finite-type and numerical variables, and the choice between standard relational analysis, and an analysis using lattice automata [GJ07] for abstracting stacks.

The Interproc Analyzer

This is a web interface to the Interproc analyzer connected to the APRON Abstract Domain Library and the Fixpoint Solver Library, whose goal is to demonstrate the features of the APRON library and, to a less extent, of the Analyzer fixpoint engine, in the static analysis field.

There are two compiled versions: interprocweb, in which all the abstract domains use underlying multiprecision integer/rational numbers, and interprocwebf, in which box and octagon domains use underlying floating-point numbers in safe way.

This is the **Interproc** version

Arguments

Please type a program, upload a file from your hard-drive, or choose one the provided examples:

Choose File no file selected

user-supplied

```
/* type your program here ! */
```

Numerical Abstract Domain: convex polyhedra (polka)

Kind of Analysis: f (sequence of forward and/or backward analysis)

Iterations/Widening options:

guided iterations 1 widening delay 2 descending steps

0 debugging level (0 to 6)

Hit the OK button to proceed: OK ! Reset

This is the **Interproc** version

Arguments

Please type a program, upload a file from your hard-drive, or choose one the provided examples:

Choose File no file selected

user-supplied

```
var x:int, y:int;
begin
  x = 0; y=9;
  while (x<=7) do
    x=x+2;
    y=y-1;
  done;
end
```

Numerical Abstract Domain: convex polyhedra (PPL)

Kind of Analysis: f (sequence of forward and/or backward analysis)

Iterations/Widening options:

☐ guided iterations 1 widening delay 2 descending steps

0 debugging level (0 to 6)

Hit the OK button to proceed: OK ! Reset

Analysis Result

Run interprocweb or interprocwebf ?

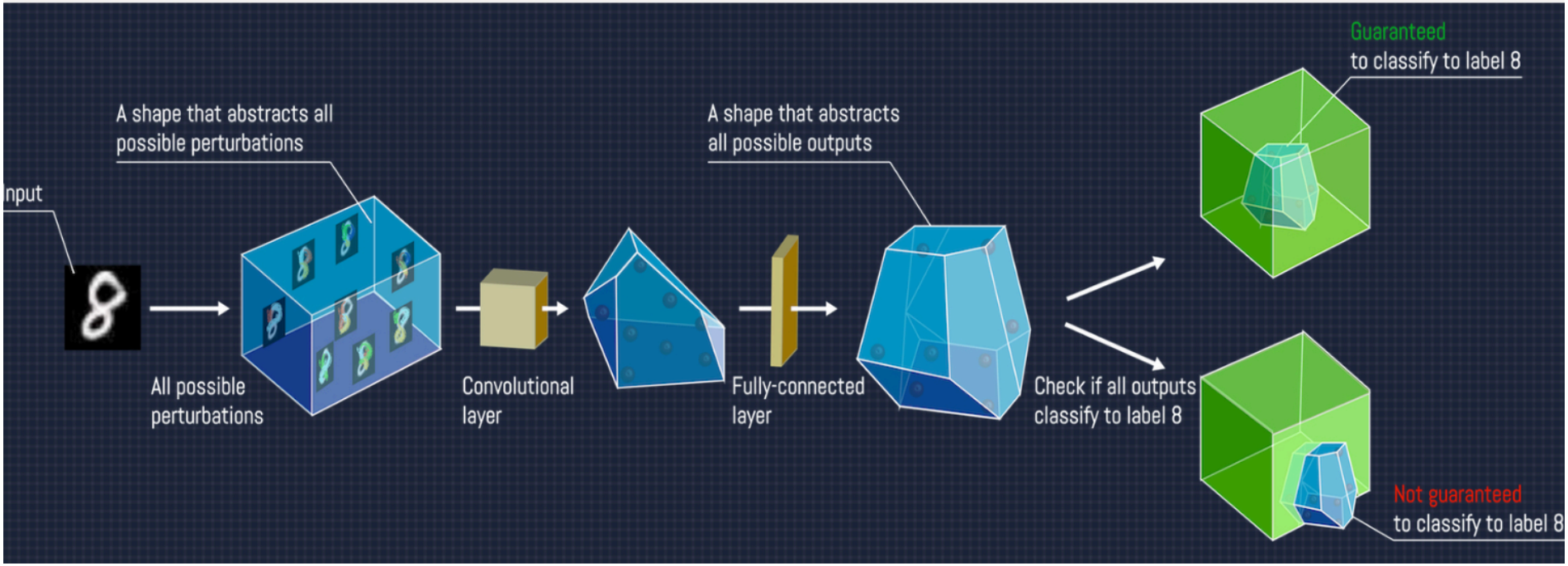
Result

```
Annotated program after forward analysis
var x : int, y : int;
begin
  /* (L2 C5) top */
  x = 0; /* (L3 C8) [|x=0|] */
  y = 9; /* (L3 C13)
            [|x+2y-18=0; -y+9>=0; y-5>=0|] */
  while x <= 7 do
    /* (L4 C17) [|x+2y-18=0; x>=0; -x+7>=0|] */
    x = x + 2; /* (L5 C10)
                  [|x+2y-20=0; -y+9>=0; 2y-11>=0|] */
    y = y - 1; /* (L6 C10)
                  [|x+2y-18=0; -y+8>=0; y-5>=0|] */
  done; /* (L7 C7) [|y-5=0; x-8=0|] */
end
```

Source

```
var x:int, y:int;
begin
  x = 0; y=9;
  while (x<=7) do
    x=x+2;
    y=y-1;
  done;
end
```

In the SafeAI project at the SRI lab, ETH Zurich, we explore new methods and systems which can ensure Artificial Intelligence (AI) systems such as deep neural networks are more robust, safe and interpretable. Our work tends to sit at the intersection of machine learning, optimization and symbolic reasoning methods. For example, among other results, we recently introduced new approaches and systems that can certify and train deep neural networks with symbolic methods, illustrated in the figure below.



Motivating program verification

The cost of software failure

- Patriot MIM-104 failure, 25 February 1991
  (death of 28 soldiers[1])
- Ariane 5 failure, 4 June 1996
  (cost estimated at more than 370 000 000 US$[2])
- Toyota electronic throttle control system failure, 2005
  (at least 89 death[3])
- Heartbleed bug in OpenSSL, April 2014
- Stagefright bug in Android, Summer 2015
  (multiple array overflows in 900 million devices, some exploitable)
- economic cost of software bugs is tremendous[4]

---

[1] R. Skeel. "Roundoff Error and the Patriot Missile". SIAM News, volume 25, nr 4.

[2] M. Dowson. "The Ariane 5 Software Failure". Software Engineering Notes 22 (2): 84, March 1997.

[3] CBSNews. Toyota "Unintended Acceleration" Has Killed 89. 20 March 2014.

[4] NIST. Software errors cost U.S. economy $59.5 billion annually. Tech. report, NIST Planning Report, 2002.

Zoom on: Ariane 5, Flight 501

Maiden flight of the Ariane 5 Launcher, 4 June 1996.

Zoom on: Ariane 5, Flight 501

40s after launch. . .

Cluster (spacecraft)

From Wikipedia, the free encyclopedia
(Redirected from Ariane 5 Flight 501)

This article is about the original, failed mission. For the successful replacemer

Cluster was a constellation of four European Space Agency spacecraft which were launched on the maiden flight of the Ariane 5 rocket, Flight 501, and subsequently lost when that rocket failed to achieve orbit. The launch, which took place on Tuesday, 4 June 1996, ended in failure due to multiple errors in the software design: Dead code (running, but purposeful so only for Ariane 4) with inadequate protection against integer overflow led to an exception handled inappropriately—halting the whole inertial navigation system that otherwise would have been unaffected. This resulted in the rocket veering off its flight path 37 seconds after launch, beginning to disintegrate under high aerodynamic forces, and finally self-destructing by its automated flight termination system. The failure has become known as one of the most infamous and expensive software bugs in history.[1] The failure resulted in a loss of more than US$370 million.[2]

Aftermath [edit]

Following the failure, four replacement Cluster II satellites were built. These were launched in pairs aboard Soyuz-U/Fregat rockets in 2000.

The launch failure brought the high risks associated with complex computing systems to the attention of the general public, politicians, and executives, resulting in increased support for research on ensuring the reliability of safety-critical systems. The subsequent automated analysis of the Ariane code (written in Ada) was the first example of large-scale static code analysis by abstract interpretation.[6]

Zoom on: Ariane 5, Flight 501

Cause: software error[5]

- arithmetic overflow in unprotected data conversion from 64-bit float to 16-bit integer types[6]

```
P_M_DERIVE(T_ALG.E_BH) :=
  UC_16S_EN_16NS (TDB.T_ENTIER_16S
    ((1.0/C_M_LSB_BH) * G_M_INFO_DERIVE(T_ALG.E_BH)));
```

- software exception not caught
  $\Longrightarrow$ computer switched off
- all backup computers run the same software
  $\Longrightarrow$ all computers switched off, no guidance
  $\Longrightarrow$ rocket self-destructs

---

[5] J.-L. Lions et al., Ariane 501 Inquiry Board report.

[6] J.-J. Levy. Un petit bogue, un grand boum. Séminaire du Département d'informatique de l'ENS, 2010.

FOREWORD

On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded. Engineers from the Ariane 5 project teams of CNES and Industry immediately started to investigate the failure. Over the following days, the Director General of ESA and the Chairman of CNES set up an independent Inquiry Board and nominated the following members :

- Prof. Jacques-Louis Lions (Chairman)	Académie des Sciences (France)
- Dr. Lennart Lübeck (Vice-Chairman)	Swedish Space Corporation (Sweden)
- Mr. Jean-Luc Fauquembergue	Délégation Générale pour l'Armement (France)
- Mr. Gilles Kahn	Institut National de Recherche en Informatique et en Automatique (INRIA), (France)
- Prof. Dr. Ing. Wolfgang Kubbat	Technical University of Darmstadt (Germany)
- Dr. Ing. Stefan Levedag	Daimler Benz Aerospace (Germany)
- Dr. Ing. Leonardo Mazzini	Alenia Spazio (Italy)
- Mr. Didier Merle	Thomson CSF (France)
- Dr. Colin O'Halloran	Defence Evaluation and Research Agency (DERA), (U.K.)

The terms of reference assigned to the Board requested it

- to determine the causes of the launch failure,
- to investigate whether the qualification tests and acceptance tests were appropriate in relation to the problem encountered,
- to recommend corrective action to remove the causes of the anomaly and other possible weaknesses of the systems found to be at fault.

The Board started its work on 13 June 1996. It was assisted by a Technical Advisory Committee composed of :

- Dr Mauro Balduccini (BPD)
- Mr Yvan Choquer (Matra Marconi Space)
- Mr Remy Hergott (CNES)
- Mr Bernard Humbert (Aerospatiale)
- Mr Eric Lefort (ESA)

Meltdown and Spectre

Vulnerabilities in modern computers leak passwords and sensitive data.

Meltdown and Spectre exploit critical vulnerabilities in modern processors. These hardware vulnerabilities allow programs to steal data which is currently processed on the computer. While programs are typically not permitted to read data from other programs, a malicious program can exploit Meltdown and Spectre to get hold of secrets stored in the memory of other running programs. This might include your passwords stored in a password manager or browser, your personal photos, emails, instant messages and even business-critical documents.

Meltdown and Spectre work on personal computers, mobile devices, and in the cloud. Depending on the cloud provider's infrastructure, it might be possible to steal data from other customers.

Meltdown

Meltdown breaks the most fundamental isolation between user applications and the operating system. This attack allows a program to access the memory, and thus also the secrets, of other programs and the operating system.

If your computer has a vulnerable processor and runs an unpatched operating system, it is not safe to work with sensitive information without the chance of leaking the information. This applies both to personal computers as well as cloud infrastructure. Luckily, there are software patches against Meltdown.

Spectre

Spectre breaks the isolation between different applications. It allows an attacker to trick error-free programs, which follow best practices, into leaking their secrets. In fact, the safety checks of said best practices actually increase the attack surface and may make applications more susceptible to Spectre

Spectre is harder to exploit than Meltdown, but it is also harder to mitigate. However, it is possible to prevent specific known exploits based on Spectre through software patches.

Spectre (security vulnerability)

From Wikipedia, the free encyclopedia

Spectre is a vulnerability that affects modern microprocessors that perform branch prediction.[1][2][3] On most processors, the speculative execution resulting from a branch misprediction may leave observable side effects that may reveal private data to attackers. For example, if the pattern of memory accesses performed by such speculative execution depends on private data, the resulting state of the data cache constitutes a side channel through which an attacker may be able to extract information about the private data using a timing attack.[4][5][6]

Two Common Vulnerabilities and Exposures IDs related to Spectre, CVE-2017-5753 (bounds check bypass, Spectre-V1, Spectre 1.0) and CVE-2017-5715 (branch target injection, Spectre-V2), have been issued.[7] JIT engines used for JavaScript were found to be vulnerable. A website can read data stored in the browser for another website, or the browser's memory itself.[8]

On 15 March 2018, Intel reported that it will redesign its CPUs (performance losses to be determined) to help protect against the Spectre and related Meltdown vulnerabilities (especially, Spectre variant 2 and Meltdown, but not Spectre variant 1), and expects to release the newly redesigned processors later in 2018.[9][10][11][12] On 8 October 2018, Intel is reported to have added hardware and firmware mitigations regarding Spectre and Meltdown vulnerabilities to its latest processors.[13] On 18 October 2018, MIT researchers suggested a new mitigation approach, called DAWG (Dynamically Allocated Way Guard), which may promise better security without compromising performance.[14]

Spectre Attacks: Exploiting Speculative Execution

Paul Kocher[1], Jann Horn[2], Anders Fogh[3], Daniel Genkin[4],
Daniel Gruss[5], Werner Haas[6], Mike Hamburg[7], Moritz Lipp[5],
Stefan Mangard[5], Thomas Prescher[6], Michael Schwarz[5], Yuval Yarom[8]
[1] Independent (www.paulkocher.com), [2] Google Project Zero,
[3] G DATA Advanced Analytics, [4] University of Pennsylvania and University of Maryland,
[5] Graz University of Technology, [6] Cyberus Technology,
[7] Rambus, Cryptography Research Division, [8] University of Adelaide and Data61

VII. Mitigation Options

Several countermeasures for Spectre attacks have been proposed. Each addresses one or more of the features that the attack relies upon. We now discuss these countermeasures and their applicability, effectiveness, and cost.

A. Preventing Speculative Execution

Alternatively, the software could be modified to use *serializing* or *speculation blocking* instructions that ensure that instructions following them are not executed speculatively. Intel and AMD recommend the use of the `lfence` instruction [4, 36]. The safest (but slowest) approach to protect conditional branches would be to add such an instruction on the two outcomes of every conditional branch. However, this amounts to disabling branch prediction and our tests indicate that this would dramatically reduce performance [36]. An improved approach is to use static analysis [36] to reduce the number of speculation blocking instructions required, since many code paths do not have the potential to read and leak out-of-bounds memory. In contrast, Microsoft's C compiler MSVC [54] takes an approach of defaulting to unprotected code unless the static analyzer detects a known-bad code pattern, but as a result misses many vulnerable code patterns [40].

Abstract Interpretation under Speculative Execution

Meng Wu
Virginia Tech
Blacksburg, VA, USA

Chao Wang
University of Southern California
Los Angeles, CA, USA

Abstract

Analyzing the behavior of a program running on a processor that supports speculative execution is crucial for applications such as execution time estimation and side channel detection. Unfortunately, existing static analysis techniques based on abstract interpretation do not model speculative execution since they focus on functional properties of a program while speculative execution does not change the functionality. To fill the gap, we propose a method to make abstract interpretation sound under speculative execution. There are two contributions. First, we introduce the notion of *virtual control flow* to augment instructions that may be speculatively executed and thus affect subsequent instructions. Second, to make the analysis efficient, we propose optimizations to handle merges and loops and to safely bound the speculative execution depth. We have implemented and evaluated the proposed method in a static cache analysis for execution time estimation and side channel detection. Our experiments show that the new method, while guaranteed to be sound under speculative execution, outperforms state-of-the-art abstract interpretation techniques that may be unsound.

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314647

How can we avoid such failures?

- Choose a safe programming language.

  C (low level) / Ada, Java (high level)

- Carefully design the software.

  many software development methods exist

- Test the software extensively.

How can we avoid such failures?

- Choose a safe programming language.

  C (low level) / Ada, Java (high level)

  yet, Ariane 5 software is written in Ada

- Carefully design the software.

  many software development methods exist

  yet, critical embedded software follow strict development processes

- Test the software extensively.

  yet, the erroneous code was well tested... on Ariane 4

  $\Longrightarrow$ **not sufficient!**

How can we avoid such failures?

- Choose a safe programming language.

  C (low level) / Ada, Java (high level)

  yet, Ariane 5 software is written in Ada

- Carefully design the software.

  many software development methods exist

  yet, critical embedded software follow strict development processes

- Test the software extensively.

  yet, the erroneous code was well tested... on Ariane 4

  $\Longrightarrow$ **not sufficient!**

We should use **formal methods.**

provide rigorous, mathematical insurance

Part I: Program Semantics

Semantics (computer science)

From Wikipedia, the free encyclopedia

In programming language theory, **semantics** is the field concerned with the rigorous mathematical study of the meaning of programming languages. It does so by evaluating the meaning of syntactically legal strings defined by a specific programming language, showing the computation involved. In such a case that the evaluation would be of syntactically illegal strings, the result would be non-computation. Semantics describes the processes a computer follows when executing a program in that specific language. This can be shown by describing the relationship between the input and output of a program, or an explanation of how the program will execute on a certain platform, hence creating a model of computation.

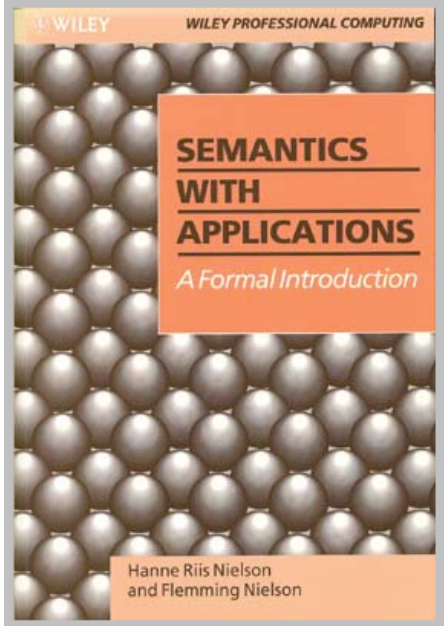
Formal semantics, for instance, helps to write compilers, better understand what a program is doing and to *prove*, e.g., that the following if statement

```
if 1 = 1 then S1 else S2
```

has the same effect as *S1* alone.

Book on program semantics

Semantics with Applications



Hanne Riis Nielson, Flemming Nielson: *Semantics with Applications: A Formal Introduction*.

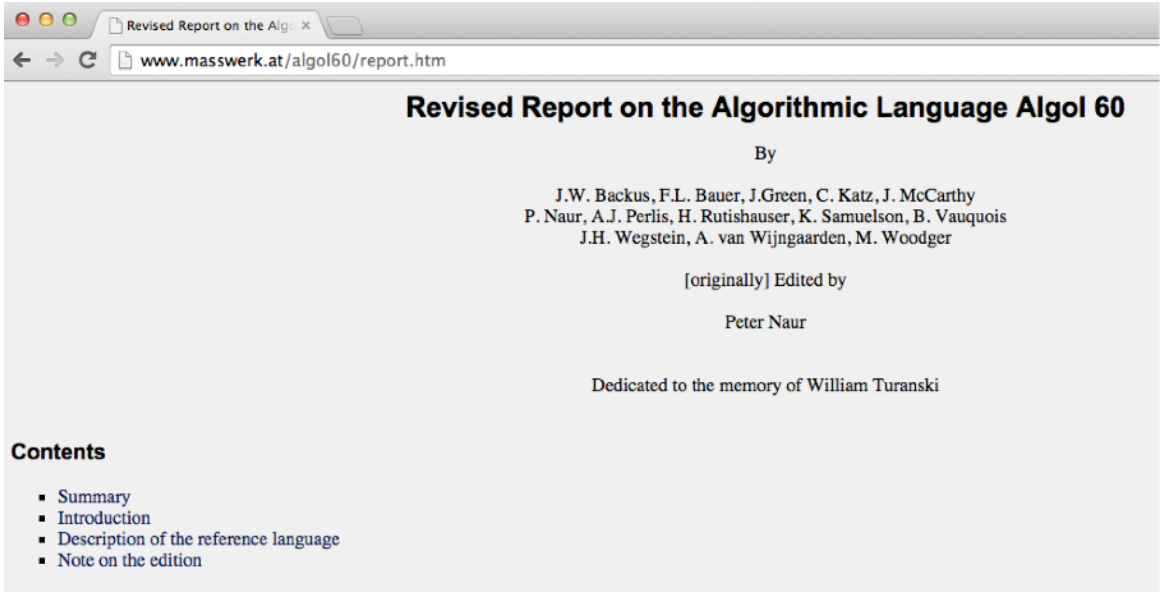
Wiley Professional Computing, (240 pages, ISBN 0 471 92980 8), Wiley, 1992.

In July 1999, a revised edition has been made available for download, in gzip'ed postscript, postcript (recommended), or pdf formats.

Program semantics

Why semantics?

- precise specification of software and hardware
- facilitate reasoning about systems: testing may reveal errors but not their absense
- form the basis for prototype implementations, e.g. interpreters and compilers



On identifiers conflicts in Algol60 (~1960)

An extract from the Algol60 report concerning procedure calls:

"Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed. If a procedure is called from a place outside the scope of any nonlocal quantity of the procedure body, the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers."

ISO/IEC JTC1 SC22 WG21 N3690
Date: 2013-05-15
ISO/IEC CD 14882
ISO/IEC JTC1 SC22
Secretariat: ANSI

Programming Languages — C++

Langages de programmation — C++

Contents

5.2.2 Function call [expr.call]

1 There are two kinds of function call: ordinary function call and member function[65] (9.3) call. A function call is a postfix expression followed by parentheses containing a possibly empty, comma-separated list of *initializer-clause*s which constitute the arguments to the function. For an ordinary function call, the postfix expression shall be either an lvalue that refers to a function (in which case the function-to-pointer standard conversion (4.3) is suppressed on the postfix expression), or it shall have pointer to function type. Calling a function through an expression whose function type has a language linkage that is different from the language linkage of the function type of the called function's definition is undefined (7.5). For a member function call, the postfix expression shall be an implicit (9.3.1, 9.4) or explicit class member access (5.2.5) whose *id-expression* is a function member name, or a pointer-to-member expression (5.5) selecting a function member; the call is as a member of the class object referred to by the object expression. In the case of an implicit class member access, the implied object is the one pointed to by `this`. [*Note:* a member function call of the form `f()` is interpreted as `(*this).f()` (see 9.3.1). — *end note*] If a function or member function name is used, the name can be overloaded (Clause 13), in which case the appropriate function shall be selected according to the rules in 13.3. If the selected function is non-virtual, or if the *id-expression* in the class member access expression is a *qualified-id*, that function is called. Otherwise, its final overrider (10.3) in the dynamic type of the object expression is called; such a call is referred to as a *virtual function call*. [*Note:* the dynamic type is the type of the object referred to by the current value of the object expression. 12.7 describes the behavior of virtual function calls when the object expression refers to an object under construction or destruction. — *end note*]

CompCert

From Wikipedia, the free encyclopedia

CompCert is a formally verified optimizing compiler for a large subset of the C99 programming language which currently targets 32-bit PowerPC, ARM, x86 and x86-64[2] architectures.[3] This project, led by Xavier Leroy, started officially in 2005, funded by the French institutes ANR and INRIA. The compiler is specified, programmed and proved in Coq. It aims to be used for programming embedded systems requiring reliability. The performance of its generated code is often close to that of GCC (version 3) at optimization level -O1, and always better than that of GCC without optimizations.[4]

Since 2015 AbsInt offers commercial licenses, provides industrial-strength support and maintenance, and contributes to the advancement of the tool.

CompCert

Stable release	3.0.1 / February 2017
Type	Compiler
License	free for noncommercial use[1]
Website	http://compcert.inria.fr/

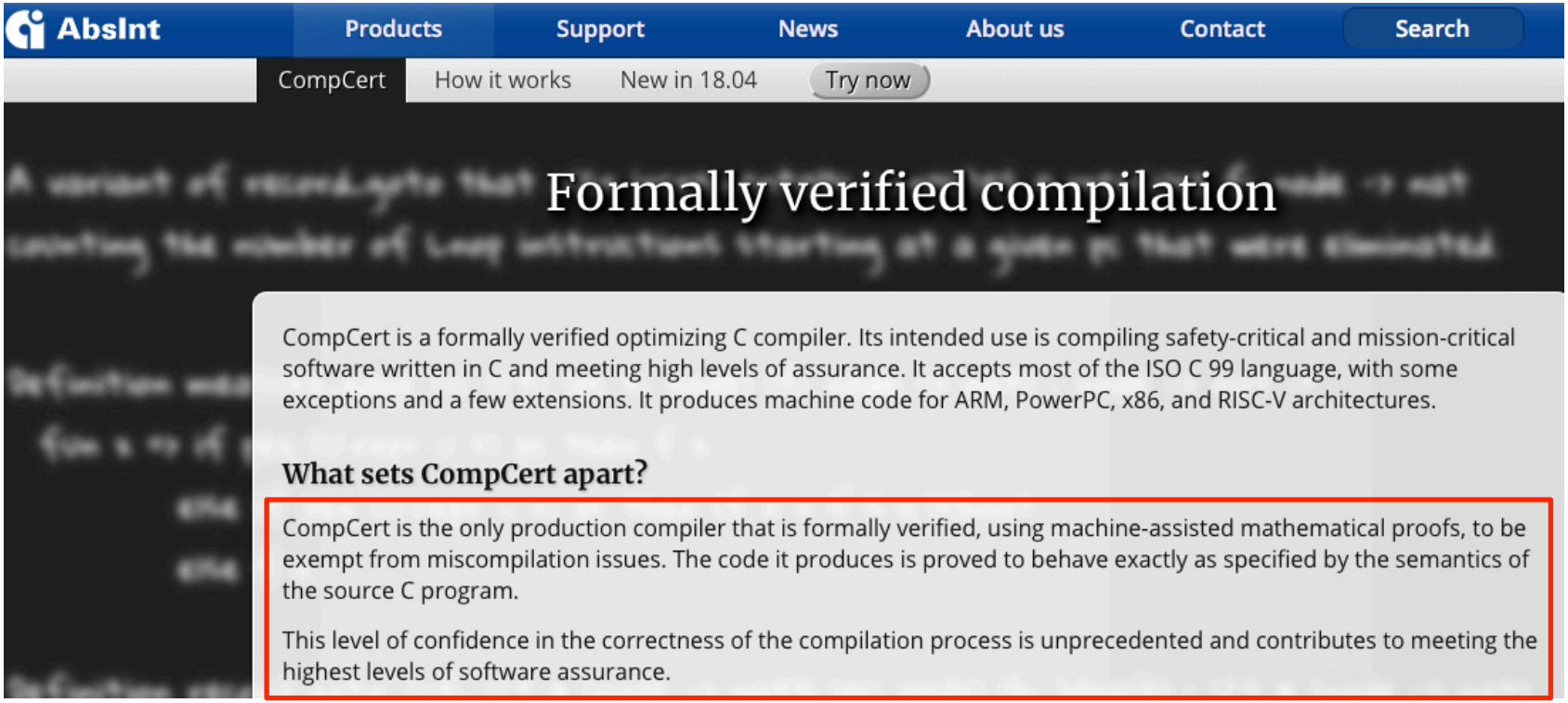
References [edit]

1. ^ http://compcert.inria.fr/doc/LICENSE
2. ^ v3.0 Release Notes
3. ^ CompCert Website
4. ^ CompCert Performance

External links [edit]

- Official website
- absint/compcert - Official Source Code Repository on GitHub

This computing article is a stub. You can help Wikipedia by expanding it.


AbsInt
Products
Support
News
About us
Contact
Search
CompCert
How it works
New in 18.04
Try now
Formally verified compilation
CompCert is a formally verified optimizing C compiler. Its intended use is compiling safety-critical and mission-critical software written in C and meeting high levels of assurance. It accepts most of the ISO C 99 language, with some exceptions and a few extensions. It produces machine code for ARM, PowerPC, x86, and RISC-V architectures.
What sets CompCert apart?
CompCert is the only production compiler that is formally verified, using machine-assisted mathematical proofs, to be exempt from miscompilation issues. The code it produces is proved to behave exactly as specified by the semantics of the source C program.
This level of confidence in the correctness of the compilation process is unprecedented and contributes to meeting the highest levels of software assurance.

Operational Semantics

$y := 1;$
`while` $\neg(x = 1)$ `do` $(y := x * y; x := x - 1)$

Operational Semantics

$y := 1;$
`while` $\neg(x = 1)$ `do` $(y := x * y; x := x - 1)$

First we assign 1 to y, then we test whether x is 1 or not. If it is then we stop and otherwise we update y to be the product of x and the previous value of y and then we decrement x by one. Now we test whether the new value of x is 1 or not $\cdots$

Two kinds of operational semantics:

- Natural Semantics
- Structural Operational Semantics

Denotational Semantics

$y := 1;$
`while` $\neg(x = 1)$ `do` $(y := x * y; x := x - 1)$

The program computes a partial function from states to states: the final state will be equal to the initial state except that the value of x will be 1 and the value of y will be equal to the factorial of the value of x in the initial state.

Two kinds of denotational semantics:

- Direct Style Semantics
- Continuation Style Semantics

Axiomatic Semantics

$y := 1;$
`while` $\neg(x = 1)$ `do` $(y := x * y; x := x - 1)$

If $x = n$ holds before the program is executed then $y = n!$ will hold when the execution terminates (if it terminates)

Two kinds of axiomatic semantics:

- Partial Correctness
- Total Correctness

Which approach?

Programming Language

Semantics

- natural semantics
- structural operational semantics
- direct style denotational semantics
- continuation style denotational semantics

Selection criteria

- constructs of the language
  - imperative
  - functional
  - concurrent/parallel
  - object oriented
  - non-deterministic
  - ...
- what is the semantics used for
  - understanding the language
  - verification of programs
  - prototyping
  - compiler construction
  - program analysis
  - ...

While language

Theoretical Development

$$
\begin{array}{rcl}
S & ::= & x := a \mid \texttt{skip} \mid S_1; S_2 \\
& & \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \\
& & \mid \texttt{while } b \texttt{ do } S \\
& & \mid \texttt{repeat } S \texttt{ until } b
\end{array}
$$

1. Structural Operational Semantics
2. Denotational Semantics

Approach

concrete syntax	LL(1)/LALR(1) grammar concrete syntax trees

↓

abstract syntax	syntactic categories abstract syntax trees

↓

semantics	semantic categories semantic definitions

Syntactic Categories for While language

- numerals
  $n \in$ Num
- variables
  $x \in$ Var

Syntax of numerals and variables is left unspecified

Naming in Java

Every programming language has its own set of rules and conventions for the kinds of names that you're allowed to use, and the Java programming language is no different. The rules and conventions for naming your variables can be summarized as follows:

- Variable names are case-sensitive. A variable's name can be any legal identifier — an unlimited-length sequence of Unicode letters and digits, beginning with a letter, the dollar sign "$", or the underscore character "_". The convention, however, is to always begin your variable names with a letter, not "$" or "_". Additionally, the dollar sign character, by convention, is never used at all. You may find some situations where auto-generated names will contain the dollar sign, but your variable names should always avoid using it. A similar convention exists for the underscore character; while it's technically legal to begin your variable's name with "_", this practice is discouraged. White space is not permitted.
- Subsequent characters may be letters, digits, dollar signs, or underscore characters. Conventions (and common sense) apply to this rule as well. When choosing a name for your variables, use full words instead of cryptic abbreviations. Doing so will make your code easier to read and understand. In many cases it will also make your code self-documenting; fields named `cadence`, `speed`, and `gear`, for example, are much more intuitive than abbreviated versions, such as `s`, `c`, and `g`. Also keep in mind that the name you choose must not be a keyword or reserved word.
- If the name you choose consists of only one word, spell that word in all lowercase letters. If it consists of more than one word, capitalize the first letter of each subsequent word. The names `gearRatio` and `currentGear` are prime examples of this convention. If your variable stores a constant value, such as `static final int NUM_GEARS = 6`, the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character. By convention, the underscore character is never used elsewhere.

Java Syntax Specification

Programs

<compilation unit> ::= *<package declaration>*? *<import declarations>*? *<type declarations>*?

Declarations

<package declaration> ::= **`package`** *<package name>* **`;`**

<import declarations> ::= *<import declaration>* | *<import declarations>* *<import declaration>*

<import declaration> ::= *<single type import declaration>* | *<type import on demand declaration>*

<single type import declaration> ::= **`import`** *<type name>* **`;`**

<type import on demand declaration> ::= **`import`** *<package name>* **`. * ;`**

<type declarations> ::= *<type declaration>* | *<type declarations>* *<type declaration>*

<type declaration> ::= *<class declaration>* | *<interface declaration>* | **`;`**

<class declaration> ::= *<class modifiers>*? **`class`** *<identifier>* *<super>*? *<interfaces>*? *<class body>*

<class modifiers> ::= *<class modifier>* | *<class modifiers>* *<class modifier>*

<class modifier> ::= **`public`** | **`abstract`** | **`final`**

<super> ::= **`extends`** *<class type>*

<interfaces> ::= **`implements`** *<interface type list>*

<interface type list> ::= *<interface type>* | *<interface type list>* **`,`** *<interface type>*

<class body> ::= **`{`** *<class body declarations>*? **`}`**

<class body declarations> ::= *<class body declaration>* | *<class body declarations>* *<class body declaration>*

<class body declaration> ::= *<class member declaration>* | *<static initializer>* | *<constructor declaration>*

<class member declaration> ::= *<field declaration>* | *<method declaration>*

<static initializer> ::= **`static`** *<block>*

<constructor declaration> ::= *<constructor modifiers>*? *<constructor declarator>* *<throws>*? *<constructor body>*

<constructor modifiers> ::= *<constructor modifier>* | *<constructor modifiers>* *<constructor modifier>*

<constructor modifier> ::= **`public`** | **`protected`** | **`private`**

<constructor declarator> ::= *<simple type name>* **`(`** *<formal parameter list>*? **`)`**

<formal parameter list> ::= *<formal parameter>* | *<formal parameter list>* **`,`** *<formal parameter>*

<formal parameter> ::= *<type>* *<variable declarator id>*

Backus-Naur Form (BNF) syntax for specifying **context-free grammars**

nonterminal

```
<postal-address> ::= <name-part> <street-address> <zip-part>

     <name-part> ::= <personal-part> <last-name> <opt-suffix-part> <EOL>
                   | <personal-part> <name-part>

 <personal-part> ::= <first-name> | <initial> "."

<street-address> ::= <house-num> <street-name> <opt-apt-num> <EOL>

      <zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>

<opt-suffix-part> ::= "Sr." | "Jr." | <roman-numeral> | ""
   <opt-apt-num> ::= <apt-num> | ""
```

terminal symbol

Syntactic Categories for While language

- numerals
  $n \in$ Num
- variables
  $x \in$ Var
- arithmetic expressions
  $a \in$ Aexp
  $a ::= n \mid x \mid a_1 + a_2$
  $\mid a_1 * a_2 \mid a_1 - a_2$

BNF

Syntactic Categories for While language

- numerals
  $n \in$ Num
- variables
  $x \in$ Var
- arithmetic expressions
  $a \in$ Aexp
  $a ::= n \mid x \mid a_1 + a_2$
  $\quad \mid a_1 * a_2 \mid a_1 - a_2$
- boolean expressions
  $b \in$ Bexp
  $b ::= \texttt{true} \mid \texttt{false} \mid a_1 = a_2$
  $\quad \mid a_1 \leq a_2 \mid \neg\, b \mid b_1 \wedge b_2$

Syntactic Categories for While language

- numerals
  $n \in$ Num
- variables
  $x \in$ Var
- arithmetic expressions
  $a \in$ Aexp
  $a ::= n \mid x \mid a_1 + a_2$
  $\quad \mid a_1 * a_2 \mid a_1 - a_2$
- boolean expressions
  $b \in$ Bexp
  $b ::= \texttt{true} \mid \texttt{false} \mid a_1 = a_2$
  $\quad \mid a_1 \leq a_2 \mid \neg\, b \mid b_1 \wedge b_2$
- statements
  $S \in$ Stm
  $S ::= x := a \mid \texttt{skip} \mid S_1; S_2$
  $\quad \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2$
  $\quad \mid \texttt{while}\ b\ \texttt{do}\ S$

Semantic Categories

Integer numbers

N = $\{..., -2, -1, 0, 1, 2, ...\}$ N should be $\mathbb{Z}$

Truth values

T = {tt, ff}

Semantic Categories

Integer numbers

$\mathsf{N} = \{..., -2, -1, 0, 1, 2, ...\}$

Truth values

$\mathsf{T} = \{\mathsf{tt}, \mathsf{ff}\}$

States

- $\mathsf{State} = \mathsf{Var} \to \mathsf{N}$
- $\mathsf{State}' = (\mathsf{Var} \times \mathsf{N})^*$
- $\mathsf{State}'' = \mathsf{Var}^* \times \mathsf{N}^*$

Lookup in a state: $s\ x$

Update a state: $s' = s[y \mapsto v]$

$$s'\ x = \begin{cases} s\ x & \text{if } x \neq y \\ v & \text{if } x = y \end{cases}$$

Meanings of the syntactic categories

Numerals

$\mathcal{N} : \mathsf{Num} \to \mathsf{N}$

Variables

$s \in \mathsf{State} = \mathsf{Var} \to \mathsf{N}$

Binary numerals

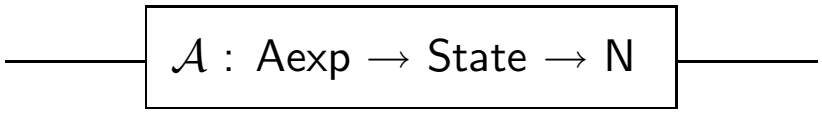
$$n ::= \mathtt{0} \mid \mathtt{1} \mid n\ \mathtt{0} \mid n\ \mathtt{1}$$

$$\mathcal{N}\colon \mathbf{Num} \to \mathbf{Z}$$

$$\begin{aligned}
\mathcal{N}[\![\mathtt{0}]\!] &= \mathbf{0} \\
\mathcal{N}[\![\mathtt{1}]\!] &= \mathbf{1} \\
\mathcal{N}[\![n\ \mathtt{0}]\!] &= \mathbf{2} \cdot \mathcal{N}[\![n]\!] \\
\mathcal{N}[\![n\ \mathtt{1}]\!] &= \mathbf{2} \cdot \mathcal{N}[\![n]\!] + \mathbf{1}
\end{aligned}$$

Fact 1.5

The equations above for $\mathcal{N}$ define a total function $\mathcal{N}\colon \mathbf{Num} \to \mathbf{Z}$.


$\mathcal{A}$: Aexp → State → N

$\mathcal{A}$: Aexp $\rightarrow$ State $\rightarrow$ N

$$
\begin{aligned}
\mathcal{A}[n]s &= \mathcal{N}[n] \\
\mathcal{A}[x]s &= s\ x \\
\mathcal{A}[a_1 + a_2]s &= \mathcal{A}[a_1]s + \mathcal{A}[a_2]s \\
\mathcal{A}[a_1\ *\ a_2]s &= \mathcal{A}[a_1]s\ *\ \mathcal{A}[a_2]s \\
\mathcal{A}[a_1 - a_2]s &= \mathcal{A}[a_1]s - \mathcal{A}[a_2]s
\end{aligned}
$$

$\mathcal{A} : \text{Aexp} \to \text{State} \to \text{N}$

$$
\begin{aligned}
\mathcal{A}[n]s &= \mathcal{N}[n] \\
\mathcal{A}[x]s &= s\ x \\
\mathcal{A}[a_1 + a_2]s &= \mathcal{A}[a_1]s + \mathcal{A}[a_2]s \\
\mathcal{A}[a_1\ *\ a_2]s &= \mathcal{A}[a_1]s\ *\ \mathcal{A}[a_2]s \\
\mathcal{A}[a_1 - a_2]s &= \mathcal{A}[a_1]s - \mathcal{A}[a_2]s
\end{aligned}
$$

Compositional definition: the semantics of an expression *exp* depends only on the semantics of (proper) subexpressions of *exp*

Principle of compositionality

From Wikipedia, the free encyclopedia

In mathematics, semantics, and philosophy of language, the **principle of compositionality** is the principle that the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them. This principle is also called **Frege's principle**, because Gottlob Frege is widely credited for the first modern formulation of it.

It is frequently taken to mean that every operation of the syntax should be associated with an operation of the semantics that acts on the meanings of the constituents combined by the syntactic operation.

Example 1.6

Suppose that $s\ \mathtt{x} = \mathbf{3}$. Then we may calculate:

$$\begin{aligned}
\mathcal{A}[\![\mathtt{x+1}]\!]s &= \mathcal{A}[\![\mathtt{x}]\!]s + \mathcal{A}[\![\mathtt{1}]\!]s \\
&= (s\ \mathtt{x}) + \mathcal{N}[\![\mathtt{1}]\!] \\
&= \mathbf{3} + \mathbf{1} \\
&= \mathbf{4}
\end{aligned}$$

Note that here $\mathtt{1}$ is a numeral (enclosed in the brackets '$[\![$' and '$]\!]$'), whereas $\mathbf{1}$ is a number. □

Example 1.7

Suppose we add the arithmetic expression $-a$ to our language. An acceptable semantic clause for this construct would be

$$\mathcal{A}[\![-a]\!]s = \mathbf{0} - \mathcal{A}[\![a]\!]s$$

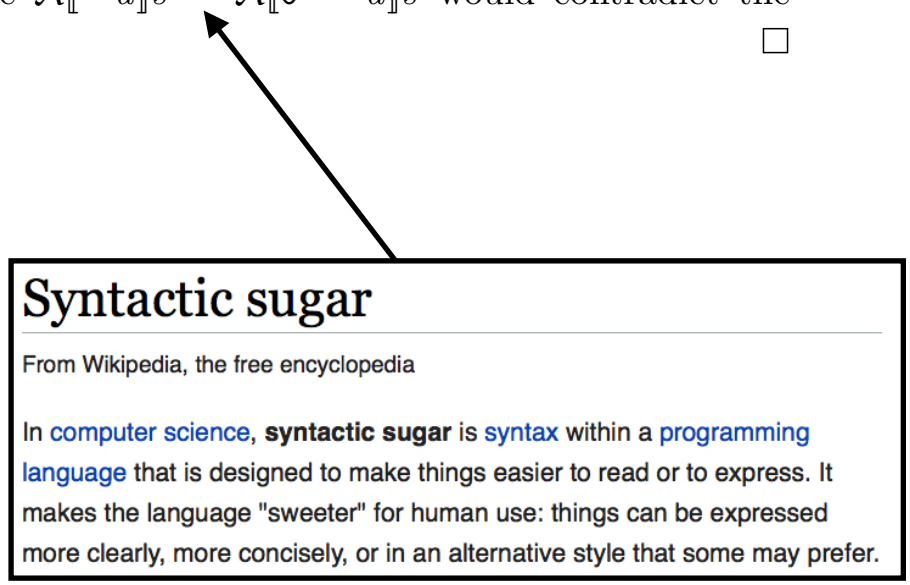
whereas the alternative clause $\mathcal{A}[\![-a]\!]s = \mathcal{A}[\![0 - a]\!]s$ would contradict the compositionality requirement. □

Example 1.7

Suppose we add the arithmetic expression $-a$ to our language. An acceptable semantic clause for this construct would be

$$\mathcal{A}[\![-a]\!]s = \mathbf{0} - \mathcal{A}[\![a]\!]s$$

whereas the alternative clause $\mathcal{A}[\![-a]\!]s = \mathcal{A}[\![0 - a]\!]s$ would contradict the compositionality requirement. □



Syntactic sugar

From Wikipedia, the free encyclopedia

In computer science, **syntactic sugar** is syntax within a programming language that is designed to make things easier to read or to express. It makes the language "sweeter" for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer.

Exercise 1.8

Prove that the equations of Table 1.1 define a total function $\mathcal{A}$ in $\mathbf{Aexp} \to (\mathbf{State} \to \mathbf{Z})$: First argue that it is sufficient to prove that for each $a \in \mathbf{Aexp}$ and each $s \in \mathbf{State}$ there is exactly one value $\mathbf{v} \in \mathbf{Z}$ such that $\mathcal{A}[\![a]\!]s = \mathbf{v}$. Next use structural induction on the arithmetic expressions to prove that this is indeed the case. □

Structural induction

From Wikipedia, the free encyclopedia

Structural induction is a proof method that is used in mathematical logic (e.g., in the proof of Łoś' theorem), computer science, graph theory, and some other mathematical fields. It is a generalization of mathematical induction over natural numbers, and can be further generalized to arbitrary Noetherian induction. **Structural recursion** is a recursion method bearing the same relationship to structural induction as ordinary recursion bears to ordinary mathematical induction.

Structural induction is used to prove that some proposition $P(x)$ holds for all x of some sort of recursively defined structure, such as formulas, lists, or trees. A well-founded partial order is defined on the structures ("subformula" for formulas, "sublist" for lists, and "subtree" for trees). The structural induction proof is a proof that the proposition holds for all the minimal structures, and that if it holds for the immediate substructures of a certain structure S, then it must hold for S also. (Formally speaking, this then satisfies the premises of an axiom of well-founded induction, which asserts that these two conditions are sufficient for the proposition to hold for all x.)

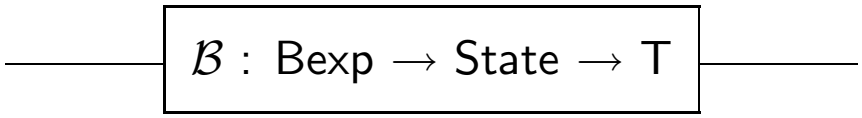
A structurally recursive function uses the same idea to define a recursive function: "base cases" handle each minimal structure and a rule for recursion. Structural recursion is usually proved correct by structural induction; in particularly easy cases, the inductive step is often left out. The *length* and ++ functions in the example below are structurally recursive.

Exercise 1.8

Prove that the equations of Table 1.1 define a total function $\mathcal{A}$ in $\mathbf{Aexp} \to (\mathbf{State} \to \mathbf{Z})$: First argue that it is sufficient to prove that for each $a \in \mathbf{Aexp}$ and each $s \in \mathbf{State}$ there is exactly one value $\mathbf{v} \in \mathbf{Z}$ such that $\mathcal{A}[\![a]\!]s = \mathbf{v}$. Next use structural induction on the arithmetic expressions to prove that this is indeed the case. □

Structural induction for BNF: how to prove that a property P(x) holds for all objects x generated by a BNF.

(base) P holds on all BNF base objects
(induction) for each BNF recursive rule x ::= ... | E(...x_1...x_k…) | ...
if P holds for x_1,...,x_k then P holds for E(...x_1...x_k…)


$\mathcal{B}$: Bexp → State → T

$\mathcal{B} : \mathsf{Bexp} \to \mathsf{State} \to \mathsf{T}$

$$\mathcal{B}[\texttt{true}]s = \mathsf{tt}$$

$$\mathcal{B}[\texttt{false}]s = \mathsf{ff}$$

$$\mathcal{B}[a_1 = a_2]s = \begin{cases} \mathsf{tt} & \text{if } \mathcal{A}[a_1]s = \mathcal{A}[a_2]s \\ \mathsf{ff} & \text{if } \mathcal{A}[a_1]s \neq \mathcal{A}[a_2]s \end{cases}$$

$$\mathcal{B}[a_1 \leq a_2]s = \begin{cases} \mathsf{tt} & \text{if } \mathcal{A}[a_1]s \leq \mathcal{A}[a_2]s \\ \mathsf{ff} & \text{if } \mathcal{A}[a_1]s \not\leq \mathcal{A}[a_2]s \end{cases}$$

$$\mathcal{B}[\neg b]s = \begin{cases} \mathsf{tt} & \text{if } \mathcal{B}[b]s = \mathsf{ff} \\ \mathsf{ff} & \text{if } \mathcal{B}[b]s = \mathsf{tt} \end{cases}$$

$$\mathcal{B}[b_1 \wedge b_2]s = \begin{cases} \mathsf{tt} & \text{if } \mathcal{B}[b_1]s = \mathsf{tt} \\ & \text{and } \mathcal{B}[b_2]s = \mathsf{tt} \\ \mathsf{ff} & \text{if } \mathcal{B}[b_1]s = \mathsf{ff} \\ & \text{or } \mathcal{B}[b_2]s = \mathsf{ff} \end{cases}$$

Exercise 1.10

Prove that Table 1.2 defines a total function $\mathcal{B}$ in $\mathbf{Bexp} \to (\mathbf{State} \to \mathbf{T})$. $\square$

Exercise 1.11

The syntactic category $\mathbf{Bexp}'$ is defined as the following extension of **Bexp**:

$$\begin{aligned} b \quad ::= \quad & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \neq a_2 \mid a_1 \leq a_2 \mid a_1 \geq a_2 \\ \mid \quad & a_1 < a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \\ \mid \quad & b_1 \Rightarrow b_2 \mid b_1 \Leftrightarrow b_2 \end{aligned}$$

Give a *compositional* extension of the semantic function $\mathcal{B}$ of Table 1.2.

Two boolean expressions b_1 and b_2 are *equivalent* if for all states s:

$$\mathcal{B}[\![b_1]\!]s = \mathcal{B}[\![b_2]\!]s$$

Show that for each b' of $\mathbf{Bexp}'$ there exists a boolean expression b of **Bexp** such that b' and b are equivalent. □

Exercise: Show that the evaluation of an arithmetic expression *E* does not depend on the variables that do not occur in *E*

$$\mathcal{A}[\![x + y + 2]\!]\{x/3, y/1, z/4\} = 6 = \mathcal{A}[\![x + y + 2]\!]\{x/3, y/1, z/8\}$$

Free variables

$$\mathrm{FV} : \mathbf{Aexp} \to \wp(\mathbf{Var})$$

$$\begin{aligned}
\mathrm{FV}(n) &= \emptyset \\
\mathrm{FV}(x) &= \{\, x \,\} \\
\mathrm{FV}(a_1 + a_2) &= \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \\
\mathrm{FV}(a_1 \star a_2) &= \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \\
\mathrm{FV}(a_1 - a_2) &= \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2)
\end{aligned}$$

$\mathrm{FV}(\texttt{x+1}) = \{\ \texttt{x}\ \}$ and $\mathrm{FV}(\texttt{x+y}\star\texttt{x}) = \{\ \texttt{x},\ \texttt{y}\ \}$

Lemma 1.12

Let s and s' be two states satisfying that $s\ x = s'\ x$ for all x in $\mathrm{FV}(a)$. Then $\mathcal{A}[\![a]\!]s = \mathcal{A}[\![a]\!]s'$.

Lemma 1.12

Let s and s' be two states satisfying that $s\ x = s'\ x$ for all x in $\mathrm{FV}(a)$. Then $\mathcal{A}[\![a]\!]s = \mathcal{A}[\![a]\!]s'$.

Proof: We shall give a fairly detailed proof of the lemma using structural induction on the arithmetic expressions. We shall first consider the basis elements of **Aexp**.

The case n: From Table 1.1 we have $\mathcal{A}[\![n]\!]s = \mathcal{N}[\![n]\!]$ as well as $\mathcal{A}[\![n]\!]s' = \mathcal{N}[\![n]\!]$. So $\mathcal{A}[\![n]\!]s = \mathcal{A}[\![n]\!]s'$ and clearly the lemma holds in this case.

The case x: From Table 1.1, we have $\mathcal{A}[\![x]\!]s = s\ x$ as well as $\mathcal{A}[\![x]\!]s' = s'\ x$. From the assumptions of the lemma, we get $s\ x = s'\ x$ because $x \in \mathrm{FV}(x)$, so clearly the lemma holds in this case.

Next we turn to the composite elements of **Aexp**.

The case $a_1 + a_2$: From Table 1.1, we have $\mathcal{A}[\![a_1 + a_2]\!]s = \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s$ and similarly $\mathcal{A}[\![a_1 + a_2]\!]s' = \mathcal{A}[\![a_1]\!]s' + \mathcal{A}[\![a_2]\!]s'$. Since a_{i} (for i = 1, 2) is an immediate subexpression of $a_1 + a_2$ and $\mathrm{FV}(a_{\mathrm{i}}) \subseteq \mathrm{FV}(a_1 + a_2)$, we can apply the induction hypothesis (that is, the lemma) to a_{i} and get $\mathcal{A}[\![a_{\mathrm{i}}]\!]s = \mathcal{A}[\![a_{\mathrm{i}}]\!]s'$. It is now easy to see that the lemma holds for $a_1 + a_2$ as well.

The cases $a_1 - a_2$ and $a_1 \star a_2$ follow the same pattern and are omitted. This completes the proof. □

Substitutions in expressions

$$a, a_0 \in \mathbf{Aexp}, y \in \mathbf{Var} \quad a[y \mapsto a_0]$$

Substitutions in expressions

$$a, a_0 \in \mathbf{Aexp}, y \in \mathbf{Var} \quad a[y \mapsto a_0]$$

$$\begin{array}{rcl}
n[y{\mapsto}a_0] & \stackrel{\Delta}{=} & n \\
x[y{\mapsto}a_0] & \stackrel{\Delta}{=} & \begin{cases} a_0 & \text{if } x = y \\ x & \text{if } x \neq y \end{cases} \\
(a_1 + a_2)[y{\mapsto}a_0] & \stackrel{\Delta}{=} & (a_1[y{\mapsto}a_0]) + (a_2[y{\mapsto}a_0]) \\
(a_1 \star a_2)[y{\mapsto}a_0] & \stackrel{\Delta}{=} & (a_1[y{\mapsto}a_0]) \star (a_2[y{\mapsto}a_0]) \\
(a_1 - a_2)[y{\mapsto}a_0] & \stackrel{\Delta}{=} & (a_1[y{\mapsto}a_0]) - (a_2[y{\mapsto}a_0])
\end{array}$$

$(\mathtt{x}{+}\mathtt{1})[\mathtt{x}{\mapsto}\mathtt{3}] = \mathtt{3}{+}\mathtt{1}$ and $(\mathtt{x}{+}\mathtt{y}{\star}\mathtt{x})[\mathtt{x}{\mapsto}\mathtt{y}{-}\mathtt{5}] = (\mathtt{y}{-}\mathtt{5}){+}\mathtt{y}{\star}(\mathtt{y}{-}\mathtt{5})$.

Substitutions in states (i.e. state updates)

$$s \in \mathbf{States}, y \in \mathbf{Var}, v \in \mathbb{Z} \qquad s[y \mapsto v]$$

Substitutions in states (i.e. state updates)

$$s \in \mathbf{States}, y \in \mathbf{Var}, v \in \mathbb{Z} \quad s[y \mapsto v]$$

$$(s[y \mapsto v])\ x \triangleq \begin{cases} v & \text{if } x = y \\ s\ x & \text{if } x \neq y \end{cases}$$

Exercise 1.14

Prove that $\mathcal{A}[\![a[y \mapsto a_0]]\!]s = \mathcal{A}[\![a]\!](s[y \mapsto \mathcal{A}[\![a_0]\!]s])$ for all states s.

Statements

Syntactic Category

$$
\begin{array}{rcl}
S & ::= & x := a \mid \texttt{skip} \mid S_1; S_2 \\
& & \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \\
& & \mid \texttt{while } b \texttt{ do } S
\end{array}
$$

input/output meaning of the syntactic category

$\mathcal{S} : \text{Stm} \to (\text{State} \hookrightarrow \text{State})$

Two operational semantics

- Natural Semantics
- Structural Operational Semantics

specified by transition systems

Transition System

$(\Gamma, T, \triangleright)$

- Γ: a set of configurations
- T: a set of terminal configurations
  $T \subseteq \Gamma$
- $\triangleright$: a transition relation
  $\triangleright \subseteq \Gamma \times \Gamma$

Natural semantics

Idea: describe how the overall result of the computation is obtained

Transition system: $(\Gamma, T, \rightarrow)$

- $\Gamma = \{(S, s) \mid S \in \text{While}, s \in \text{State}\} \cup \text{State}$
- $T = \text{State}$
- $\rightarrow \subseteq \{(S, s) \mid S \in \text{While}, s \in \text{State}\} \times \text{State}$

Typical transition:

$$(S, s) \rightarrow s'$$

where

S is the program
s is the initial state
s' is the final state

Structural Operational Semantics

Idea: describe how the individual steps of the computation takes place

Transition system: $(\Gamma, T, \Rightarrow)$

- $\Gamma = \{(S, s) \mid S \in \text{While}, s \in \text{State}\} \cup \text{State}$
- $T = \text{State}$
- $\Rightarrow \subseteq \{(S, s) \mid S \in \text{While}, s \in \text{State}\} \times \Gamma$

Two typical transitions:

- the computation has not been completed after one step of computation:
  $$(S, s) \Rightarrow (S', s')$$
- the computation is completed after one step of computation:
  $$(S, s) \Rightarrow s'$$

Structural Operational Semantics

$$[\mathrm{ass}_{\mathrm{sos}}] \qquad \langle x := a,\ s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$$

Structural Operational Semantics

$[\mathrm{ass}_{\mathrm{sos}}]$ $\quad \langle x := a,\ s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$

$[\mathrm{skip}_{\mathrm{sos}}]$ $\quad \langle \mathtt{skip},\ s\rangle \Rightarrow s$

Structural Operational Semantics

$[\text{ass}_{\text{sos}}]$ $\quad \langle x := a,\ s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$

$[\text{skip}_{\text{sos}}]$ $\quad \langle \texttt{skip},\ s\rangle \Rightarrow s$

$[\text{comp}^1_{\text{sos}}]$ $\quad \dfrac{\langle S_1,\ s\rangle \Rightarrow \langle S_1',\ s'\rangle}{\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_1';S_2,\ s'\rangle}$

$[\text{comp}^2_{\text{sos}}]$ $\quad \dfrac{\langle S_1,\ s\rangle \Rightarrow s'}{\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_2,\ s'\rangle}$

Structural Operational Semantics

$[\text{ass}_{\text{sos}}]$ $\quad \langle x := a,\ s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$

$[\text{skip}_{\text{sos}}]$ $\quad \langle \texttt{skip},\ s\rangle \Rightarrow s$

$[\text{comp}^1_{\text{sos}}]$ $\quad \dfrac{\langle S_1,\ s\rangle \Rightarrow \langle S_1',\ s'\rangle}{\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_1';S_2,\ s'\rangle}$

$[\text{comp}^2_{\text{sos}}]$ $\quad \dfrac{\langle S_1,\ s\rangle \Rightarrow s'}{\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_2,\ s'\rangle}$

$[\text{if}^{\text{tt}}_{\text{sos}}]$ $\quad \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_1,\ s\rangle \text{ if } \mathcal{B}[\![b]\!]s = \mathbf{tt}$

$[\text{if}^{\text{ff}}_{\text{sos}}]$ $\quad \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_2,\ s\rangle \text{ if } \mathcal{B}[\![b]\!]s = \mathbf{ff}$

Structural Operational Semantics

$[\text{ass}_{\text{sos}}] \qquad \langle x := a,\ s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$

$[\text{skip}_{\text{sos}}] \qquad \langle \texttt{skip},\ s\rangle \Rightarrow s$

$$[\text{comp}^1_{\text{sos}}] \qquad \frac{\langle S_1,\ s\rangle \Rightarrow \langle S_1',\ s'\rangle}{\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_1';S_2,\ s'\rangle}$$

$$[\text{comp}^2_{\text{sos}}] \qquad \frac{\langle S_1,\ s\rangle \Rightarrow s'}{\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_2,\ s'\rangle}$$

$[\text{if}^{\text{tt}}_{\text{sos}}] \qquad \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_1,\ s\rangle \text{ if } \mathcal{B}[\![b]\!]s = \mathbf{tt}$

$[\text{if}^{\text{ff}}_{\text{sos}}] \qquad \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_2,\ s\rangle \text{ if } \mathcal{B}[\![b]\!]s = \mathbf{ff}$

$[\text{while}_{\text{sos}}] \qquad \langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \Rightarrow$

$\qquad\qquad \langle \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip},\ s\rangle$

unfolding rule

Swap program

$$s_0 = \langle \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7 \rangle$$

$$\langle (\mathtt{z} := \mathtt{x};\ \ \mathtt{x} := \mathtt{y});\ \mathtt{y} := \mathtt{z},\ s_0 \rangle$$

$$\Rightarrow \langle \mathtt{x} := \mathtt{y};\ \mathtt{y} := \mathtt{z},\ s_0[\mathtt{z} \mapsto \mathbf{5}] \rangle$$

$$\Rightarrow \langle \mathtt{y} := \mathtt{z},\ (s_0[\mathtt{z} \mapsto \mathbf{5}])[\mathtt{x} \mapsto \mathbf{7}] \rangle$$

$$\Rightarrow ((s_0[\mathtt{z} \mapsto \mathbf{5}])[\mathtt{x} \mapsto \mathbf{7}])[\mathtt{y} \mapsto \mathbf{5}]$$

Swap program

$$s_0 = \langle \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7 \rangle$$

$$\begin{aligned}
&\langle (\mathtt{z} := \mathtt{x};\ \ \mathtt{x} := \mathtt{y});\ \mathtt{y} := \mathtt{z},\ s_0 \rangle \\
&\Rightarrow \langle \mathtt{x} := \mathtt{y};\ \mathtt{y} := \mathtt{z},\ s_0[\mathtt{z} \mapsto \mathbf{5}] \rangle \\
&\Rightarrow \langle \mathtt{y} := \mathtt{z},\ (s_0[\mathtt{z} \mapsto \mathbf{5}])[\mathtt{x} \mapsto \mathbf{7}] \rangle \\
&\Rightarrow ((s_0[\mathtt{z} \mapsto \mathbf{5}])[\mathtt{x} \mapsto \mathbf{7}])[\mathtt{y} \mapsto \mathbf{5}]
\end{aligned}$$

Corresponding to *each* of these steps, we have *derivation trees* explaining why they take place. For the first step

$$\langle (\mathtt{z} := \mathtt{x};\ \mathtt{x} := \mathtt{y});\ \mathtt{y} := \mathtt{z},\ s_0 \rangle \Rightarrow \langle \mathtt{x} := \mathtt{y};\ \mathtt{y} := \mathtt{z},\ s_0[\mathtt{z} \mapsto \mathbf{5}] \rangle$$

the derivation tree is

$$\dfrac{\dfrac{\langle \mathtt{z} := \mathtt{x},\ s_0 \rangle \Rightarrow s_0[\mathtt{z} \mapsto \mathbf{5}]}{\langle \mathtt{z} := \mathtt{x};\ \mathtt{x} := \mathtt{y},\ s_0 \rangle \Rightarrow \langle \mathtt{x} := \mathtt{y},\ s_0[\mathtt{z} \mapsto \mathbf{5}] \rangle}}{\langle (\mathtt{z} := \mathtt{x};\ \mathtt{x} := \mathtt{y});\ \mathtt{y} := \mathtt{z},\ s_0 \rangle \Rightarrow \langle \mathtt{x} := \mathtt{y};\ \mathtt{y} := \mathtt{z},\ s_0[\mathtt{z} \mapsto \mathbf{5}] \rangle}$$

Factorial program

$$s = \langle \mathrm{x} \mapsto 3 \rangle$$

$$\frac{\langle \mathtt{y{:=}1},\ s\rangle \Rightarrow s[\mathrm{y}\mapsto \mathbf{1}]}{\begin{array}{l}\langle \mathtt{y{:=}1;\ while}\ \neg(\mathrm{x}{=}1)\ \mathtt{do}\ (\mathrm{y}{:=}\mathrm{y}\star\mathrm{x};\ \mathrm{x}{:=}\mathrm{x}{-}1),\ s\rangle \Rightarrow \\ \quad \langle \mathtt{while}\ \neg(\mathrm{x}{=}1)\ \mathtt{do}\ (\mathrm{y}{:=}\mathrm{y}\star\mathrm{x};\ \mathrm{x}{:=}\mathrm{x}{-}1),\ s[\mathrm{y}\mapsto\mathbf{1}]\rangle\end{array}}$$

$$\Rightarrow$$

$$\begin{array}{l}\langle \mathtt{if}\ \neg(\mathrm{x}{=}1)\ \mathtt{then}\ \ ((\mathrm{y}{:=}\mathrm{y}\star\mathrm{x};\ \mathrm{x}{:=}\mathrm{x}{-}1); \\ \qquad\qquad\qquad\qquad \mathtt{while}\ \neg(\mathrm{x}{=}1)\ \mathtt{do}\ (\mathrm{y}{:=}\mathrm{y}\star\mathrm{x};\ \mathrm{x}{:=}\mathrm{x}{-}1)) \\ \qquad\qquad\quad \mathtt{else\ skip},\ s[\mathrm{y}\mapsto\mathbf{1}]\rangle\end{array}$$

$$\Rightarrow$$

$$\langle (\mathrm{y}{:=}\mathrm{y}\star\mathrm{x};\ \mathrm{x}{:=}\mathrm{x}{-}1);\ \mathtt{while}\ \neg(\mathrm{x}{=}1)\ \mathtt{do}\ (\mathrm{y}{:=}\mathrm{y}\star\mathrm{x};\ \mathrm{x}{:=}\mathrm{x}{-}1),\ s[\mathrm{y}\mapsto\mathbf{1}]\rangle$$

$$\Rightarrow$$

$$\langle \mathrm{x}{:=}\mathrm{x}{-}1;\ \mathtt{while}\ \neg(\mathrm{x}{=}1)\ \mathtt{do}\ (\mathrm{y}{:=}\mathrm{y}\star\mathrm{x};\ \mathrm{x}{:=}\mathrm{x}{-}1),\ s[\mathrm{y}\mapsto\mathbf{3}]\rangle$$

$$\Rightarrow$$

$$\langle \mathtt{while}\ \neg(\mathrm{x}{=}1)\ \mathtt{do}\ (\mathrm{y}{:=}\mathrm{y}\star\mathrm{x};\ \mathrm{x}{:=}\mathrm{x}{-}1),\ s[\mathrm{y}\mapsto\mathbf{3}][\mathrm{x}\mapsto\mathbf{2}]\rangle$$

$$\Rightarrow \cdots \Rightarrow \quad s[\mathrm{y}\mapsto\mathbf{6}][\mathrm{x}\mapsto\mathbf{1}]$$

Rules and axioms

$$\frac{(S_1, s_1) \to s_1', \quad \cdots, \quad (S_n, s_n) \to s_n'}{(S, s) \to s'}$$

if $\cdots$

Requirement:

> $S_1, \cdots, S_n$ are immediate constituents of S or are statements constructed from the immediate constituents of S

Terminology:

- premise: $(S_i, s_i) \to s_i'$
- conclusion: $(S, s) \to s'$
- side condition: if $\cdots$

Axiom: Rule with an empty set of premises.

Inferences

Schemes vs. instances

- axiom schemes vs. instances of axioms
- rule schemes vs. instances of rules

Derivation tree

- simple
- composite

Derivation sequence:

$\gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots$

- finite
  $\gamma_1, \gamma_2, \cdots, \gamma_k$ where $\gamma_i \Rightarrow \gamma_{i+1}$ and $\gamma_k \not\Rightarrow$
- infinite
  $\gamma_1, \gamma_2, \cdots$ where $\gamma_i \Rightarrow \gamma_{i+1}$

Transition systems

Deterministic transition system

Transition systems

Deterministic transition system

Whenever $\gamma \rhd \gamma'$ and $\gamma \rhd \gamma''$
we have $\gamma' = \gamma''$

Transition systems

Deterministic transition system

Whenever $\gamma \rhd \gamma'$ and $\gamma \rhd \gamma''$
we have $\gamma' = \gamma''$

γ is a stuck configuration

if there are no γ' such that $\gamma \rhd \gamma'$

Looping programs

How do we model that programs loop?

Looping programs

How do we model that programs loop?

Not all programs have a finite derivation sequence for a given state:

$(\texttt{while true do skip}, s) \Rightarrow^*$
$(\texttt{while true do skip}, s) \Rightarrow$
$\ldots$

Terminating and Looping Programs

In analogy with the terminology of the previous section, we shall say that the execution of a statement S on a state s

– *terminates* if and only if there is a finite derivation sequence starting with $\langle S,\ s\rangle$ and

Terminating and Looping Programs

In analogy with the terminology of the previous section, we shall say that the execution of a statement S on a state s

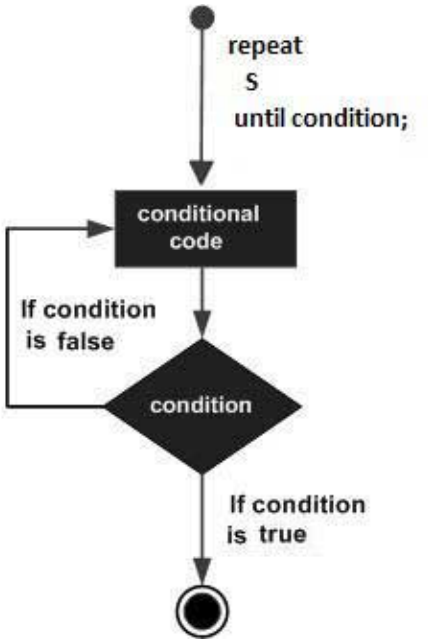
– *terminates* if and only if there is a finite derivation sequence starting with $\langle S,\ s\rangle$

– *loops* if and only if there is an infinite derivation sequence starting with $\langle S,\ s\rangle$.

We shall say that the execution of S on s *terminates successfully* if $\langle S,\ s\rangle \Rightarrow^* s'$ for some state s'; in **While** an execution terminates successfully if and only if it terminates because there are no stuck configurations.

Exercise 2.17

Extend **While** with the construct `repeat` S `until` b and specify a structural operational semantics for it. (The semantics for the `repeat`-construct is not allowed to rely on the existence of a `while`-construct.) □



Exercise 2.17

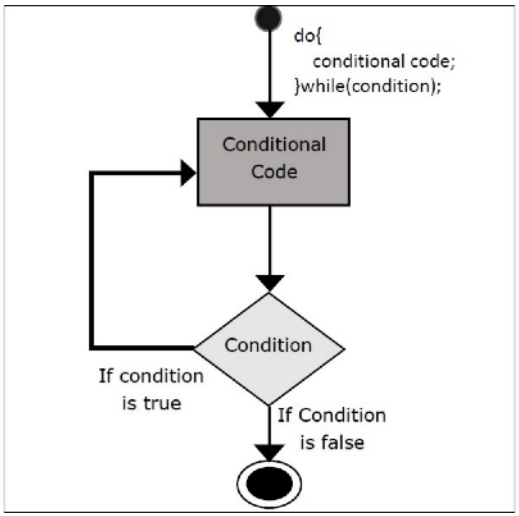
Extend **While** with the construct `repeat` S `until` b and specify a structural operational semantics for it. (The semantics for the `repeat`-construct is not allowed to rely on the existence of a `while`-construct.) □

$$\langle \texttt{repeat } S \texttt{ until } b, s\rangle \Rightarrow \langle S; \texttt{if } b \texttt{ then skip else } (\texttt{repeat } S \texttt{ until } b), s\rangle$$

Exercise 2.17

Extend **While** with the construct `repeat` S `until` b and specify a structural operational semantics for it. (The semantics for the `repeat`-construct is not allowed to rely on the existence of a `while`-construct.) □

$$\langle \texttt{repeat}\ S\ \texttt{until}\ b, s\rangle \Rightarrow \langle S; \texttt{if}\ b\ \texttt{then skip else}\ (\texttt{repeat}\ S\ \texttt{until}\ b), s\rangle$$



$$\langle \texttt{do}\ S\ \texttt{while}\ b, s\rangle \Rightarrow \langle S; \texttt{if}\ b\ \texttt{then}\ (\texttt{do}\ S\ \texttt{while}\ b)\ \ \texttt{else skip}, s\rangle$$

Exercise 2.18

Extend **While** with the construct `for` $x := a_1$ `to` a_2 `do` S and specify the structural operational semantics for it. Hint: You may need to assume that you have an "inverse" to $\mathcal{N}$ so that there is a numeral for each number that may arise during the computation. (The semantics for the `for`-construct is not allowed to rely on the existence of a `while`-construct.) □

Exercise 2.18

Extend **While** with the construct `for` $x := a_1$ `to` a_2 `do` S and specify the structural operational semantics for it. Hint: You may need to assume that you have an "inverse" to $\mathcal{N}$ so that there is a numeral for each number that may arise during the computation. (The semantics for the `for`-construct is not allowed to rely on the existence of a `while`-construct.) □

What semantics would you give to **for** $x := 1$ **to** $x + 1$ **do** **skip** ? And to **for** $x := 1$ **to** 3 **do** $x := x - 1$?

Exercise 2.18

Extend **While** with the construct `for` $x := a_1$ `to` a_2 `do` S and specify the structural operational semantics for it. Hint: You may need to assume that you have an "inverse" to $\mathcal{N}$ so that there is a numeral for each number that may arise during the computation. (The semantics for the `for`-construct is not allowed to rely on the existence of a `while`-construct.) □

Pascal

loop with iterator variable

`for` used along with `to`/`downto` and `do` constructs a loop in which the value of a control variable is incremented or decremented by `1` passing every iteration.

behavior

```
for controlVariable := start to finalValue do
begin
        statement;
end;
```

In this example `controlVariable` is first initialized with the value of `start` (but cmp. § "legacy" below). If and as long as `controlVariable` is not greater than `finalValue`, the `begin` ... `end` block with all its statements is executed. By reaching `end` `controlVariable` is incremented by `1` and the comparison is made, whether another iteration, whether the statement-block is executed again.

Exercise 2.18

Extend **While** with the construct `for` $x := a_1$ `to` a_2 `do` S and specify the structural operational semantics for it. Hint: You may need to assume that you have an "inverse" to $\mathcal{N}$ so that there is a numeral for each number that may arise during the computation. (The semantics for the `for`-construct is not allowed to rely on the existence of a `while`-construct.) □

$$\langle \texttt{for}\ x := a_1 \texttt{to}\ a_2\ \texttt{do}\ S, s\rangle \Rightarrow$$
$$\langle x := a_1; \texttt{if}\ (x < a_2)\ \texttt{then}\ (S; \texttt{for}\ x := a_1 + 1\ \texttt{to}\ a_2\ \texttt{do}\ S)\ \texttt{else skip}, s\rangle$$

vs

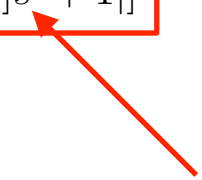
$$\langle \texttt{for}\ x := a_1\ \texttt{to}\ a_2\ \texttt{do}\ S, s\rangle \Rightarrow$$
$$\langle x := a_1; \texttt{if}\ x < a_2\ \texttt{then}\ (S; \texttt{for}\ x := x + 1\ \texttt{to}\ a_2\ \texttt{do}\ S)\ \ \texttt{else skip}, s\rangle$$

Exercise 2.18

Extend **While** with the construct `for` $x := a_1$ `to` a_2 `do` S and specify the structural operational semantics for it. Hint: You may need to assume that you have an "inverse" to $\mathcal{N}$ so that there is a numeral for each number that may arise during the computation. (The semantics for the `for`-construct is not allowed to rely on the existence of a `while`-construct.) □

$$\frac{< x := a_1, s >\Rightarrow s'}{< \textbf{for } x := a_1 \textbf{ to } a_2 \textbf{ do } S,\ s >\Rightarrow < \textbf{if } \neg(x = a_2) \textbf{ then } (S; \textbf{for } x := a_3 \textbf{ to } a_2 \textbf{ do } S) \textbf{ else } skip, s' >}$$

where $a_3 = \mathcal{N}^{-1}[|\mathcal{A}[|a_1|]s' + 1|]$



C++ for loop

C++ | C++ language | Statements

for loop

Executes *init-statement* once, then executes *statement* and *iteration_expression* repeatedly, until the value of *condition* becomes false. The test takes place before each iteration.

Syntax

formal syntax:

attr(optional) **for** **(** *init-statement* *condition*(optional) **;** *iteration_expression*(optional) **)** *statement*

informal syntax:

attr(optional) **for** **(** *declaration-or-expression*(optional) **;** *declaration-or-expression*(optional) **;** *expression*(optional) **)** *statement*

***attr*(C++11)**	-	any number of attributes
init-statement	-	either • an expression statement (which may be a *null statement* ";") • a simple declaration, typically a declaration of a loop counter variable with initializer, but it may declare arbitrary many variables Note that any *init-statement* must end with a semicolon **;**, which is why it is often described informally as an expression or a declaration followed by a semicolon.
condition	-	either • an expression which is contextually convertible to bool. This expression is evaluated before each iteration, and if it yields `false`, the loop is exited. • a declaration of a single variable with a brace-or-equals initializer. the initializer is evaluated before each iteration, and if the value of the declared variable converts to `false`, the loop is exited.
iteration_expression	-	any expression, which is executed after every iteration of the loop and before re-evaluating *condition*. Typically, this is the expression that increments the loop counter
statement	-	any statement, typically a compound statement, which is the body of the loop

Explanation

The above syntax produces code equivalent to:

```
{
    init_statement
    while ( condition ) {
        statement
        iteration_expression ;
    }
}
```

C++ for loop (instantianted to **While**)

$$\texttt{for}\ (\texttt{skip};\ b;\ y := a_2)\ S$$

$$\texttt{for}\ (x := a_1;\ b;\ y := a_2)\ S$$

C++ for loop (instantianted to **While**)

$$\texttt{for}\ (\texttt{skip};\ b;\ y := a_2)\ S$$

$$\texttt{for}\ (x := a_1;\ b;\ y := a_2)\ S$$

$$S_{init} ::= \texttt{skip} \mid x := a$$

C++ for loop (instantianted to **While**)

$$\texttt{for}\ (\texttt{skip};\ b;\ y := a_2)\ S$$

$$\texttt{for}\ (x := a_1;\ b;\ y := a_2)\ S$$

$$S_{init} ::= \texttt{skip} \mid x := a$$

$$\langle \texttt{for}\ (S_{init};\ b;\ y := a_2)\ S, s\rangle \Rightarrow$$
$$\langle S_{init};\ y := a_2;\ \texttt{if}\ b\ \texttt{then}\ (S; \texttt{for}\ (\texttt{skip};\ b;\ y := a_2)\ S)\ \texttt{else skip}, s\rangle$$

Semantic equivalence

S_1 and S_2 are semantically equivalent if for all states s

- for all configurations γ that are either stuck or final

$$(S_1, s) \Rightarrow^* \gamma \text{ if and only if } (S_2, s) \Rightarrow^* \gamma$$

- there is an infinite derivation sequence starting in (S_1, s) if and only if there is one starting in (S_2, s)

Definitions and Proofs

Three approaches to semantics

- compositional definitions
- natural semantics
- structural operational semantics

Three proof principles

- structural induction
- induction on the shape of derivation trees
- induction on the length of derivation sequences

Structural Induction

- Prove that the property holds for all the basis elements of the syntactic category.
- Prove that the property holds for all the composite elements by assuming that the property holds for the immediate constituents of the element (this is called the induction hypothesis) and proving that it also holds for the element itself.

Ind. on Shape of Derivation Trees

- Prove that the property holds for all the simple derivation trees by showing that it holds for all the axioms of the transition system
- Prove that the property holds for all the composite derivation trees: For each rule assume that the property holds for its premises (this is called the induction hypothesis) and prove that it also holds for the conclusion of the rule provided that the conditions of the rule are satisfied

Ind. on Length of Derivation Seq.

- Prove that the property holds for all derivation sequences of length 0
- Prove that the property holds for all other derivation sequences: Assume that the property holds for derivation sequences of length at most k (this is called the induction hypothesis) and prove that it holds for derivation sequences of length $k + 1$

Notation

$\gamma \Rightarrow \gamma'$ transition relation between configurations

$\gamma \Rightarrow^k \gamma'$, where $k \geq 1$ (by induction)

Convention: $\gamma \Rightarrow^0$ when γ is stuck

$\gamma \Rightarrow^* \gamma'$ means $\exists k \geq 1.\, \gamma \Rightarrow^k \gamma'$ or $\gamma \Rightarrow^0$

Proof by Ind. on Length of Derivation Seq.

Lemma 2.19 (Decomposition Lemma)

If $\langle S_1;S_2,\ s\rangle \Rightarrow^{\mathrm{k}} s''$, then there exists a state s' and natural numbers k_1 and k_2 such that $\langle S_1,\ s\rangle \Rightarrow^{\mathrm{k}_1} s'$ and $\langle S_2,\ s'\rangle \Rightarrow^{\mathrm{k}_2} s''$, where $\mathrm{k} = \mathrm{k}_1{+}\mathrm{k}_2$.

Proof by Ind. on Length of Derivation Seq.

Lemma 2.19 (Decomposition Lemma)

If $\langle S_1;S_2,\ s\rangle \Rightarrow^{\mathrm{k}}\ s''$, then there exists a state s' and natural numbers k_1 and k_2 such that $\langle S_1,\ s\rangle \Rightarrow^{\mathrm{k}_1}\ s'$ and $\langle S_2,\ s'\rangle \Rightarrow^{\mathrm{k}_2}\ s''$, where $\mathrm{k} = \mathrm{k}_1{+}\mathrm{k}_2$.

Corollary

If $\langle S_1; S_2, s\rangle \Rightarrow^*\ s''$ then there exists s' such that $\langle S_1, s\rangle \Rightarrow^*\ s'$ and $\langle S_2, s'\rangle \Rightarrow^*\ s''$.

Proof: The proof is by induction on the number k; that is, by induction on the length of the derivation sequence $\langle S_1;S_2,\ s\rangle \Rightarrow^{\mathrm{k}} s''$.

If $\mathrm{k} = 0$, then the result holds vacuously (because $\langle S_1;S_2,\ s\rangle$ and s'' are different).

For the induction step, we assume that the lemma holds for $k \leq k_0$, and we shall prove it for k_0+1. So assume that

$$\langle S_1;S_2,\ s\rangle \Rightarrow^{k_0+1}\ s''$$

This means that the derivation sequence can be written as

$$\langle S_1;S_2,\ s\rangle \Rightarrow \gamma \Rightarrow^{k_0}\ s''$$

for some configuration γ. Now one of two cases applies depending on which of the two rules $[\mathrm{comp}^1_{\mathrm{sos}}]$ and $[\mathrm{comp}^2_{\mathrm{sos}}]$ was used to obtain $\langle S_1;S_2,\ s\rangle \Rightarrow \gamma$.

In the first case, where $[\mathrm{comp}^1_{\mathrm{sos}}]$ is used, we have $\gamma = \langle S_1'; S_2, s' \rangle$ and

$$\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_1';S_2,\ s'\rangle$$

because

$$\langle S_1,\ s\rangle \Rightarrow \langle S_1',\ s'\rangle$$

We therefore have

$$\langle S_1';S_2,\ s'\rangle \Rightarrow^{\mathrm{k}_0}\ s''$$

and the induction hypothesis can be applied to this derivation sequence because it is shorter than the one with which we started. This means that there is a state s_0 and natural numbers k_1 and k_2 such that

$$\langle S_1',\ s'\rangle \Rightarrow^{\mathrm{k}_1}\ s_0 \text{ and } \langle S_2,\ s_0\rangle \Rightarrow^{\mathrm{k}_2}\ s''$$

where $\mathrm{k}_1{+}\mathrm{k}_2{=}\mathrm{k}_0$. Using that $\langle S_1,\ s\rangle \Rightarrow \langle S_1',\ s'\rangle$ and $\langle S_1',\ s'\rangle \Rightarrow^{\mathrm{k}_1}\ s_0$, we get

$$\langle S_1,\ s\rangle \Rightarrow^{\mathrm{k}_1+1}\ s_0$$

We have already seen that $\langle S_2,\ s_0\rangle \Rightarrow^{\mathrm{k}_2}\ s''$, and since $(\mathrm{k}_1{+}1){+}\mathrm{k}_2 = \mathrm{k}_0{+}1$, we have proved the required result.

The second possibility is that $[\mathrm{comp}^2_{\mathrm{sos}}]$ has been used to obtain the derivation $\langle S_1;S_2,\ s\rangle \Rightarrow \gamma$. Then we have

$$\langle S_1,\ s\rangle \Rightarrow s'$$

and γ is $\langle S_2,\ s'\rangle$ so that

$$\langle S_2,\ s'\rangle \Rightarrow^{\mathrm{k}_0}\ s''$$

The result now follows by choosing $\mathrm{k}_1{=}1$ and $\mathrm{k}_2{=}\mathrm{k}_0$. □

Exercise 2.21 (Composition Lemma)

Prove that

$$\text{if } \langle S_1,\, s\rangle \Rightarrow^{\mathrm{k}} s' \text{ then } \langle S_1;S_2,\, s\rangle \Rightarrow^{\mathrm{k}} \langle S_2,\, s'\rangle$$

I.e., the execution of S_1 is not influenced by the statement following it.

Exercise 2.21 (Composition Lemma)

Prove that

$$\text{if } \langle S_1,\, s\rangle \Rightarrow^{\mathrm{k}} s' \text{ then } \langle S_1;S_2,\, s\rangle \Rightarrow^{\mathrm{k}} \langle S_2,\, s'\rangle$$

I.e., the execution of S_1 is not influenced by the statement following it.

Corollary

If $\langle S_1, s\rangle \Rightarrow^* s'$ then $\langle S_1; S_2, s\rangle \Rightarrow^* \langle S_2, s'\rangle$.

The SOS semantics is deterministic if

$$\forall S, s, \gamma, \gamma'$$

$$\langle S,\ s\rangle \Rightarrow \gamma \text{ and } \langle S,\ s\rangle \Rightarrow \gamma' \text{ imply } \gamma = \gamma'$$

Easy structural induction

Exercise 2.22 (Essential)

Show that the structural operational semantics of Table 2.2 is deterministic. Deduce that there is exactly one derivation sequence starting in a configuration $\langle S, s\rangle$. Argue that a statement S of **While** cannot both terminate and loop on a state s and hence cannot both be always terminating and always looping. □

We skip the proof.

S_1 and S_2 are SOS equivalent if $\forall s$

- $\langle S_1,\, s\rangle \Rightarrow^* \gamma$ if and only if $\langle S_2,\, s\rangle \Rightarrow^* \gamma$, whenever γ is a configuration that is either stuck or terminal, and
- there is an infinite derivation sequence starting in $\langle S_1,\, s\rangle$ if and only if there is one starting in $\langle S_2,\, s\rangle$.

No stuck configuration, thus:
S_1 and S_2 are equivalent if and only if $\forall s, s'.\, \langle S_1, s\rangle \Rightarrow^* s'$ iff $\langle S_2, s\rangle \Rightarrow^* s'$

Exercise 2.23

Show that the following statements of **While** are semantically equivalent in the sense above:

– S;`skip` and S

– `while` b `do` S and `if` b `then` (S; `while` b `do` S) `else skip`

– S_1;(S_2;S_3) and (S_1;S_2);S_3

$$\langle S; \texttt{skip}, s\rangle \Rightarrow^* s' \quad \text{iff} \quad \langle S, s\rangle \Rightarrow^* s'$$

($\Rightarrow$):
from $\langle S; \texttt{skip}, s\rangle \Rightarrow^* s'$, by Lemma, $\langle S, s\rangle \Rightarrow^* s''$ and $\langle \texttt{skip}, s''\rangle \Rightarrow^* s'$
The only choice for $\langle \texttt{skip}, s''\rangle \Rightarrow^* s'$ is $\langle \texttt{skip}, s''\rangle \Rightarrow s''$, i.e., $s' = s''$.

($\Leftarrow$):
Assume that $\langle S, s\rangle \Rightarrow^* s'$. By composition lemma, we obtain $\langle S; \texttt{skip}, s\rangle \Rightarrow^* \langle \texttt{skip}, s'\rangle$. Since $\langle \texttt{skip}, s'\rangle \Rightarrow s'$, we conclude that $\langle S; \texttt{skip}, s\rangle \Rightarrow^* s'$.

Semantic functions

$$\mathcal{S}_{\text{sos}}\colon \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State})$$

$$\mathcal{S}_{\text{sos}}[\![S]\!]s = \begin{cases} s' & \text{if } \langle S,\ s\rangle \Rightarrow^*\ s' \\ \underline{\text{undef}} & \text{otherwise} \end{cases}$$

Semantic functions

$$\mathcal{S}_{\text{sos}}\colon \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State})$$

$$\mathcal{S}_{\text{sos}}[\![S]\!]s = \begin{cases} s' & \text{if } \langle S,\ s\rangle \Rightarrow^*\ s' \\ \underline{\text{undef}} & \text{otherwise} \end{cases}$$

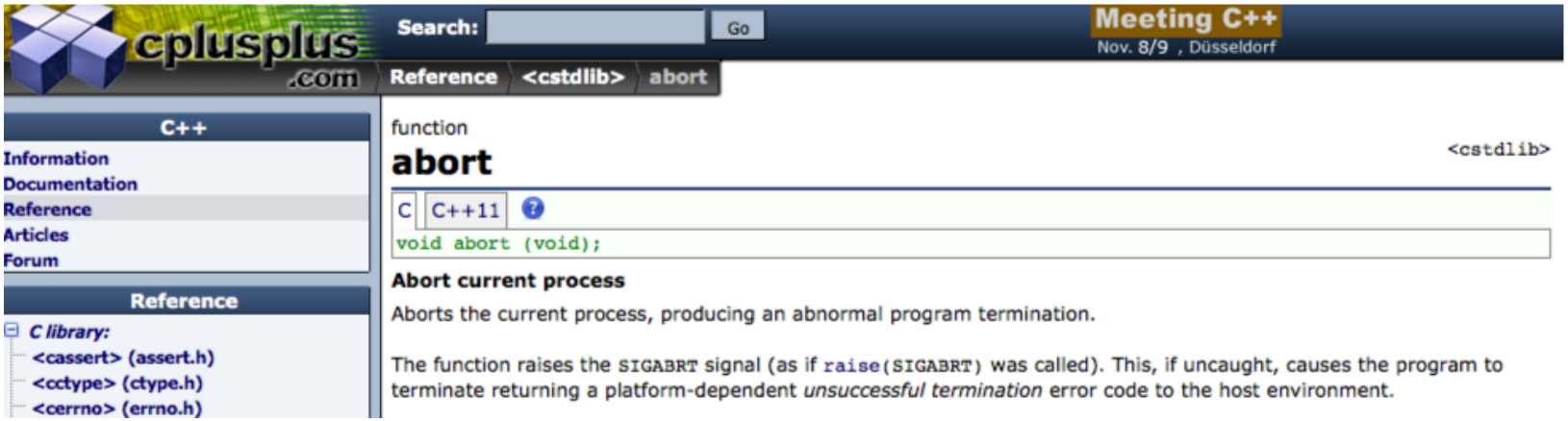
This is a well defined function

Extended While language

$$
\begin{array}{rcl}
S & ::= & x := a \mid \texttt{skip} \mid S_1; S_2 \\
& \mid & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \\
& \mid & \texttt{while } b \texttt{ do } S \\
& \mid & \texttt{abort} \\
& \mid & S_1 \texttt{ or } S_2 \\
& \mid & S_1 \texttt{ par } S_2
\end{array}
$$

How is the semantics modified?

Adding abortion `abort`



exit

```
public static void exit(int status)
```

Terminates the currently running Java Virtual Machine. The argument serves as a status code; by convention, a nonzero status code indicates abnormal termination.

This method calls the `exit` method in class `Runtime`. This method never returns normally.

The call `System.exit(n)` is effectively equivalent to the call:

```
Runtime.getRuntime().exit(n)
```

Parameters:

`status` - exit status.

Throws:

`SecurityException` - if a security manager exists and its `checkExit` method doesn't allow exit with the specified status.

Adding abortion `abort`

Configurations:

$$\{(S, s) \mid S \in \text{While}^{abort},\ s \in \text{State}\} \cup \text{State}$$

Adding abortion `abort`

Configurations:

$$\{(S, s) \mid S \in \text{While}^{abort},\ s \in \text{State}\} \cup \text{State}$$

Transition relation for SOS:

unchanged

$\texttt{skip} \cong_{sos} \texttt{abort}\ ?$

$\texttt{abort} \cong_{sos} \texttt{while true do skip}\ ?$

$\texttt{skip} \cong_{sos} \texttt{abort}$? No

$\texttt{abort} \cong_{sos} \texttt{while true do skip}$? No

SOS is able to distinguish between stuck and loop behaviors (this does not happen for big step semantics)

Adding nondeterminism S_1 `or` S_2

Configurations:

$$\{(S, s) \mid S \in \text{While}^{or},\ s \in \text{State}\} \cup \text{State}$$

Adding nondeterminism S_1 `or` S_2

Configurations:

$$\{(S, s) \mid S \in \text{While}^{or},\ s \in \text{State}\} \cup \text{State}$$

Transition relation for SOS:

$$(S_1 \texttt{ or } S_2, s) \Rightarrow (S_1, s)$$

$$(S_1 \texttt{ or } S_2, s) \Rightarrow (S_2, s)$$

For the program

(`x`:=1) or (`while true do skip`)

we have two derivation sequences,
a finite one and an infinite one.

In SOS, nondeterminism does not remove possible nontermination

(in big step semantics, instead, nondeterminism removes possible nontermination)

Adding parallelism $S_1 \texttt{ par } S_2$

Configurations:

$$\{(S, s) \mid S \in \text{While}^{par},\ s \in \text{State}\} \cup \text{State}$$

Adding parallelism S_1 `par` S_2

Configurations:

$$\{(S, s) \mid S \in \text{While}^{par},\ s \in \text{State}\} \cup \text{State}$$

(x:=1) `par` (x:=2; x:= x+2) may evaluate in x↦1, x↦3, x↦4

Transition relation for SOS:

$$\frac{(S_1, s) \Rightarrow (S_1', s')}{(S_1 \texttt{ par } S_2, s) \Rightarrow (S_1' \texttt{ par } S_2, s')}$$

$$\frac{(S_1, s) \Rightarrow s'}{(S_1 \texttt{ par } S_2, s) \Rightarrow (S_2, s')}$$

Transition relation for SOS:

$$\frac{(S_1, s) \Rightarrow (S_1', s')}{(S_1 \texttt{ par } S_2, s) \Rightarrow (S_1' \texttt{ par } S_2, s')}$$

$$\frac{(S_1, s) \Rightarrow s'}{(S_1 \texttt{ par } S_2, s) \Rightarrow (S_2, s')}$$

$$\frac{(S_2, s) \Rightarrow (S_2', s')}{(S_1 \texttt{ par } S_2, s) \Rightarrow (S_1 \texttt{ par } S_2', s')}$$

$$\frac{(S_2, s) \Rightarrow s'}{(S_1 \texttt{ par } S_2, s) \Rightarrow (S_1, s')}$$

Exercise: model [*S*] in SOS, where the intended meaning is that the program *S* cannot be interleaved. This is a basic form of synchronization.

(x:=1) `par` [x:=2; x:= x+2] may evaluate in x↦1, x↦4 only

Exercise: model $[S]$ in SOS, where the intended meaning is that the program S cannot be interleaved. This is a basic form of synchronization.

$$\frac{\langle S, s\rangle \Rightarrow s'}{\langle [S], s\rangle \Rightarrow s'}$$

What about $\dfrac{\langle S, s\rangle \Rightarrow \langle S', s'\rangle}{\langle [S], s\rangle \Rightarrow \langle [S'], s'\rangle}$?

With this rule, what about: $\langle x := 1 \,\texttt{par}\, [x := 2; x := x + 2], s\rangle \Rightarrow^* s[x/3]$?

Exercise: model $[S]$ in SOS, where the intended meaning is that the program S cannot be interleaved. This is a basic form of synchronization.

$$\frac{\langle S, s\rangle \Rightarrow s'}{\langle [S], s\rangle \Rightarrow s'}$$

What about $\dfrac{\langle S, s\rangle \Rightarrow \langle S', s'\rangle}{\langle [S], s\rangle \Rightarrow \langle [S'], s'\rangle}$?

With this rule, what about: $\langle x := 1 \texttt{ par } [x := 2; x := x + 2], s\rangle \Rightarrow^* s[x/3]$?

Doesn't work!

Exercise: model [*S*] in SOS, where the intended meaning is that the program *S* cannot be interleaved. This is a basic form of synchronization.

$$\frac{\langle S, s\rangle \Rightarrow s'}{\langle [S], s\rangle \Rightarrow s'}$$

$$\frac{\langle S, s\rangle \Rightarrow \langle S_0, s''\rangle \quad \langle [S_0], s''\rangle \Rightarrow s'}{\langle [S], s\rangle \Rightarrow s'}$$

Non-compositional and recursive rule

(x:=1) `par` [x:=2; x:= x+2] may evaluate in x↦1, x↦4 only