

Construction and Verification of Software

2018 - 2019

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

Lecture 2 - Specification and Verification

João Costa Seco (joao.seco@fct.unl.pt)

based on previous editions by **Luís Caires** (lcaires@fct.unl.pt)



FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Software Correctness

Relevance of Software Correctness

- Quality procedures must be enforced at all levels, in particular at the construction phase, where most of the issues are introduced and difficult to circumvent.
- **Questions for you now:**
 - What methods do you currently use to make sure your code is “bullet-proof” ?
 - How can you prove to yourself (and others) that your code is “bullet-proof” ?
 - What arguments do you use to convince yourself and others that your code works as expected and not goes wrong, with respect to functional correctness, security, or concurrency errors?
- You will **know better answers** at the end of this course.

Software Correctness: What and How

- **Key engineering concern:**
Make sure that the software developed and constructed is “correct”.
- What does this mean?
 - Is it crash-free? (“**runtime safety**”)
 - Gives the right results? (“**functional correctness**”)
 - Does it operate effectively? (“**resource conformance**”)
 - Does it violate user privacy? (“**security conformance**”)
 - ...
 - several process and methodological approaches to

Correctness is against a specification

- Then what does “correct software” mean?
 - Always relative to some given (**our**) specs
 - Correct means that software meets **our** specs
 - There is no such thing as the “right specification”
 - In practice, the spec is usually incomplete ...
 - But **the spec must not be wrong !**
 - It should be very easy to check what the spec states
 - The spec **must be simple, much simpler than code**
 - The spec should be **focused** (pick relevant cases)
 - e.g., buffers are not being overrun
 - e.g., never transfer money without logging the source

Checking Specs: Dynamic Verification

- By “dynamic verification” we mean that verification is **done at runtime**, while the program executes
- Some successful approaches:
 - **unit testing**
 - **coverage testing**
 - **regression testing**
 - **test generation**
 - **runtime monitoring**
- use runtime monitors to (continuously) check that code do not violate correctness properties
- violations causes exceptional behaviour or halt, so errors are detected after something wrong already occurred (think of a car crash, or a security leak)

Checking Specs: Dynamic Verification

- Some shortcomings of dynamic verification
 - always introduces a level of performance overhead
 - may show the existence of some errors, but does not ensure absence of errors (the code passed a test suite today, but may fail with some other clever test)
- **Challenge:** how do you make sure that you are defining the “right” tests and “enough” tests
- Will talk about testing methods later on in the course

Checking Specs: Static Verification

- “static verification” means verification at **compile time**
- relies on algorithmic reasoning about what programs do, by analysing the source code, not by running the code
- can ensure absence of all errors of a certain well defined kind (e.g., “no null dereferences”)
- can also tackle many complex correctness properties (e.g., functionality, absence of races, security, etc.)
- does not introduce in performance overhead at runtime (e.g. generics work by erasure)
- success stories:
 - **type checking**, as performed by the compiler
 - **extended checking**, static checking of assertions
 - **abstract interpretation**, simulates execution on a simpler decidable abstract model of runtime data

Checking Specs: Static vs. Dynamic

- Dynamic Verification
 - unsound
 - runtime
 - more flexible
 - execution overhead
 - based on ad-hoc testing
- Static Verification
 - sound
 - compile time
 - conservative
(may have false positives)
 - erasure of verification
 - each analysis targets a kind of property
 - some are more complex to design and use

Checking Specs: Static Verification

- Specifications are the essential tool for abstraction and decomposition.
- For each program we need to know
 - in what conditions it can be used (requires/pre-conditions)
 - what are its effects (effects/ensures/post-conditions)
- The post condition assertion can be assumed after the program's execution, provided that the pre-conditions were met at the beginning. That's the only assumptions that can be drawn from the post-condition.

Design by Contract & Defensive Programming

Design by Contract

- Design by contract
 - Eiffel language (Bertrand Meyer)
 - Formal specification of pre-, post-conditions and invariants
 - Assume that all preconditions are met when invoking an operation and that all postconditions will be satisfied after the operation is executed.
- Logic based verification
 - Axiomatic approaches based on Hoare Logic
 - If all components are verified, all contracts are fulfilled in all cases.



Defensive Programming

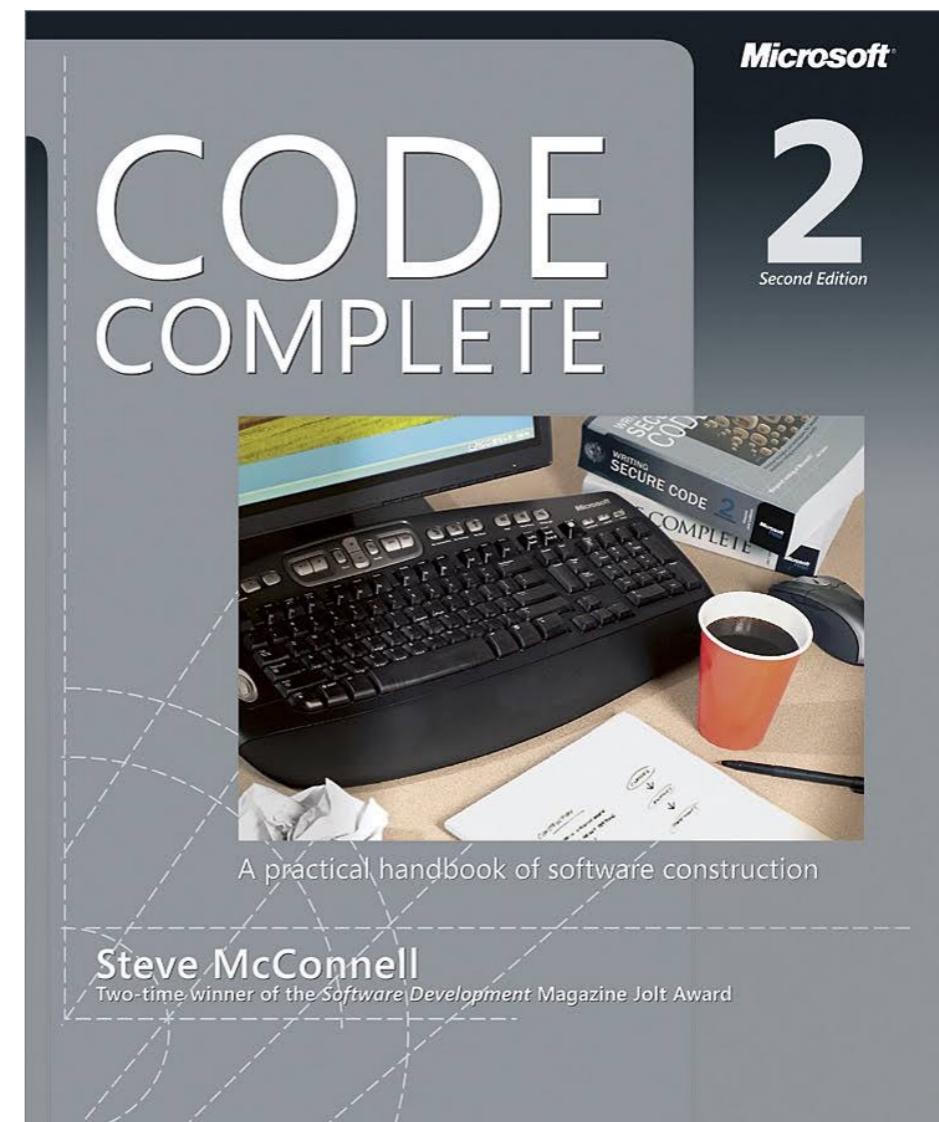
- Defensive programming
 - Prepare for all possible inputs and associated responses

Chapter 8

Defensive Programming

Contents

- 8.1 Protecting Your Program from Invalid Inputs: page 188
- 8.2 Assertions: page 189
- 8.3 Error-Handling Techniques: page 194
- 8.4 Exceptions: page 198
- 8.5 Barricade Your Program to Contain the Damage Caused by Errors: page 203
- 8.6 Debugging Aids: page 205
- 8.7 Determining How Much Defensive Programming to Leave in Production Code: page 209
- 8.8 Being Defensive About Defensive Programming: page 210



What may specs look like?

- A classical example is the use of “assertions”
 - You have used assertions before (IP, POO, AED)?
- A simple and fine grained spec is the “Hoare triple”:

$$\{ A \} \ P \ \{ B \}$$

- **A** and **B** are assertions (conditions on the program state)
- **P** is the piece of code we want to talk about
- The Hoare triple says:
 - If program P starts in a state satisfying A, then, if it terminates, the resulting state satisfies B.
 - A is called the “pre-condition”
 - B is called the “post-condition”

Interface contracts in ADT specs

- ADT specifications (we will detail this later) involve method contracts, expressed as assertions

```
method P(... parameters ...)  
requires pre-condition-assertion % PRE  
ensures post-condition-assertion % POST  
modifies global-state-changed % MOD  
{  
    ... method code  
}
```

- The method call $P(\dots)$, whenever started in a state that satisfies PRE, if it terminates, always ends in a state that satisfies POST, and only has effects on MOD

Invariants in ADT specs

- ADT specifications (we will detail this later) may involve representation invariants and abstraction mappings also expressed as assertions

```
class C {  
    invariant invariant-assertion REPINV  
    invariant abstraction-map-assertions ABSMAP  
    {  
        ... methods...  
    }
```

- ADT C implementation relies on a representation type T that satisfies the representation invariant REPINV and maps into the abstract type as specified by ABSMAP

Stack Example : A glimpse of dafny code

```
class Stack {  
  
    var elements:array<int>;  
    var count: int;  
    var MAX: int;  
  
    function StackInv(): bool  
    reads `count, `MAX, `elements  
{  
    0 < MAX && 0 <= count <= MAX && elements.Length == MAX  
}  
  
constructor()  
ensures StackInv()  
{  
    MAX := 10;  
    elements := new int[10];  
    count := 0;  
}
```

Stack Example : A glimpse of dafny code

```
class Stack {  
    ...  
    method push(x:int)  
        requires StackInv() && notFull()  
        ensures StackInv() && notEmpty()  
        modifies elements, `count  
{  
    elements[count] := x;  
    count := count + 1;  
}  
  
method pop() returns (x:int)  
    requires StackInv() && notEmpty()  
    ensures StackInv() && notFull()  
    modifies elements, `count  
{  
    count := count - 1;  
    x := elements[count];  
}  
  
function notFull():bool  
reads `count, `MAX  
{ count < MAX }  
  
function notEmpty():bool  
reads `count, `MAX  
{ count > 0 }
```

How are Specs verified?

- A logic is used to prove properties of programs
- What kinds of properties are we interested in?
 - safety properties (partial correctness)
 - state that if the program terminates (delivers an outcome), then the final state satisfies some property
 - liveness properties (total correctness)
 - say that the program terminates (at least under certain conditions)
- Hoare logic is the “mother of all program logics”: It provides a foundation for most program logics for imperative programming languages
- Reason of HL success: verification at the level of the programming languages (not of programs, cf. Floyd)

Dafny

“Dafny is an imperative object-based language with built-in specification constructs. The Dafny static program verifier can be used to verify the functional correctness of programs. The specifications include pre- and postconditions, frame specifications (read and write sets), and termination metrics”

Leino, Koenig, 2010

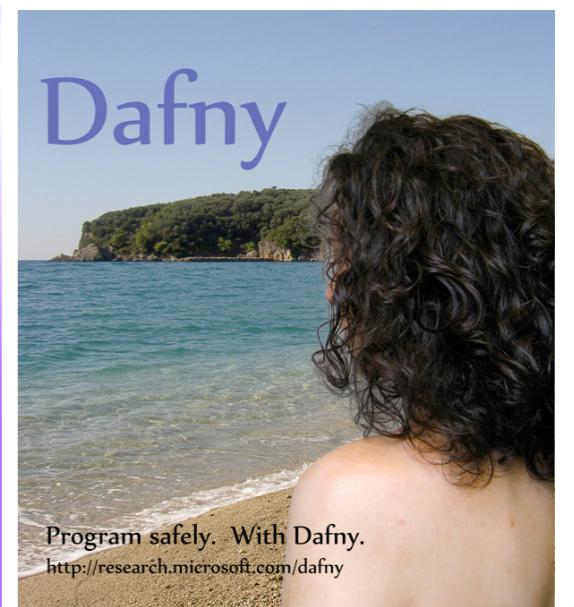
Dafny: An Automatic Program Verifier for Functional Correctness

K. Rustan M. Leino
Microsoft Research
leino@microsoft.com

Abstract

Traditionally, the full verification of a program’s functional correctness has been obtained with pen and paper or with interactive proof assistants, whereas only reduced verification tasks, such as extended static checking, have enjoyed the automation offered by satisfiability-modulo-theories (SMT) solvers. More recently, powerful SMT solvers and well-designed program verifiers are starting to break that tradition, thus reducing the effort involved in doing full verification.

This paper gives a tour of the language and verifier Dafny, which has been used to verify the functional correctness of a number of challenging pointer-based programs. The paper describes the features incorporated in Dafny, illustrating their use by small examples and giving a taste of how they are coded for an SMT solver. As a larger case study, the paper shows the full functional specification of the Schorr-Waite algorithm in Dafny.



rise4fun @ Microsoft Research (MSR)

The screenshot shows the rise4fun website interface. At the top, there's a browser-like header with tabs for 'rise4fun.com' and other links like 'http://www.apapnce.pdf', 'STM', 'ERC metrics', etc. Below the header, the main title 'rise4fun' is displayed with the subtitle '708285 programs analyzed'. A blue diagonal banner on the right says 'Your tool here'. The page features a grid of tool cards:

- dafny**: A language and program verifier for functional correctness.
- f***: A verification tool for higher-order stateful programs.
- visual c++**: The Visual C++ compiler.
- z3rcf**: Python interface for the Z3 Real Closed Fields package (and Theorem Prover).

Below this section, there's a heading 'microsoft' followed by a 4x4 grid of tool cards:

- agl**: Automatic Graph Layout.
- bek**: A domain specific language for writing and analyzing common string functions.
- boogie**: Intermediate Verification Language.
- chalice**: A language and program verifier for reasoning about concurrent programs.

- code contracts**: Language agnostic modular program verification and repair with abstract interpretation.
- counterdog**: Theorem-prover for Counterfactual Datalog.
- dafny**: A language and program verifier for functional correctness.
- dkal**: Distributed Knowledge Authorization Language.

- esm**: Empirical Software Engineering and Measurement Group.
- formula**: Formal Modeling Using Logic Programming and Analysis.
- try f#**: Programming language combining functional, object-oriented and scripting programming.
- f***: A verification tool for higher-order stateful programs.

- heapdbg**: Runtime heap abstraction.
- koka**: A function-oriented language with effect inference.
- pex**: Automatic test generation using Dynamic Symbolic Execution for .NET.
- poirot**: Poirot.

A glimpse of Dafny programming



Is this program correct?

```
1 class PSet
2 {
3     var s: set<int>;
4     var n: int;
5
6     function SetInv(): bool
7     reads this;
8     {
9         (forall x::x in s ==> x >= 0) && |s| == n
10    }
11
12 method initBag()
13 ensures SetInv();
14 modifies this;
15 {
16     s := {};
17     n := 0;
18 }
19
20 method add(x:int)
21 requires SetInv() && x >= 0;
22 modifies this;
```



[home](#) [video](#) [permalink](#)
'►' shortcut: Alt+B

Basic Program Specs (Hoare Logic)





C.A.R. HOARE
United Kingdom – **1980**

For his fundamental contributions to the definition and design
of programming languages.

Hoare Logic (1969)

An Axiomatic Basis for Computer Programming

C. A. R. HOARE

The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

KEY WORDS AND PHRASES: axiomatic method, theory of programming, proofs of programs, formal language definition, programming language design, machine-independent programming, program documentation

CR CATEGORY: 4.0, 4.21, 4.22, 5.20, 5.21, 5.23, 5.24

of axioms it is possible to deduce such simple theorems as:

$$x = x + y \times 0$$

$$y \leq r \supset r + y \times q = (r - y) + y \times (1 + q)$$

The proof of the second of these is:

$$\begin{aligned} A5 \quad & (r - y) + y \times (1 + q) \\ &= (r - y) + (y \times 1 + y \times q) \\ &= (r - y) + (y + y \times q) \\ A9 \quad &= ((r - y) + y) + y \times q \\ A3 \quad &= r + y \times q \\ A6 \quad &= r + y \times q \end{aligned}$$

The axioms A1 to A9 are, of course, true of the computational infinite set of integers in many cases. They are also true of the finite sets of integers manipulated by computers provided they are defined to *nonnegative* numbers. Their validity depends on the size of the set; furthermore, it is dependent on the choice of technique applied in the proof. "flow"; for example:

(1) Strict interpretation: the result of an overflow operation does not exist; when overflow occurs in a running program never completes its operations. In this case, the equalities of A1 to A9 are not valid since that both sides exist or fail to exist.



Simple Programming Language

$E ::=$	Expressions	
	num	Integer
	x	Variable
	$E + E \mid \dots$	Integer operators
	$E < E \mid \dots$	Relational operators
	$E \text{ and } E \dots$	Boolean operators

$P ::=$	Programs
	No op
skip	
	Assignment
$x := E$	
	Sequential Composition
$P; P$	
	Conditional
$\text{if } E \text{ then } P \text{ else } P$	
	Iteration
$\text{while } E \text{ do } P$	

States and State Transformers

- A program is a state transformer,
it transforms an initial state into a target state
- What is a program state? a state is an assignment of
values to state variables

$$\sigma = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$$

- An imperative program transforms states into states

$$P \triangleq x := y + x; z := z - x$$

- If P is executed in state σ it yields state σ' where

$$\sigma' = \{x \mapsto 3, y \mapsto 2, z \mapsto 0\}$$

- We may say that P transforms σ in σ'
- P is only defined on states σ where $\text{vars}(P) \subseteq \text{dom}(\sigma)$

States and Assertions

- A (safety) property is a set of (safe) states
- Essentially an assertion is a boolean expression that only depends on observing program (state) variables
- Thus, an assertion is just a pure observation, it is either true or false, its evaluation does not change the state
- In general, one may use all the expressiveness of (first order) logic in assertions (e.g. quantifiers, etc...)
- The assertion language is part of the specification language, not of the programming language
- But in some cases, assertions may be expressed in the programming language (Java / Dafny).

Assertions in Dafny



Is this program correct?

```
1 method strncpy(a:array<int>, n:int, b:array<int>)
2 requires a!=null && b!= null;
3 requires 0 <= n <= a.Length <= b.Length;
4 modifies b;
5 ensures forall j::(0<=j<n) ==> b[j] == a[j];
6 {
7 var i:int :=0;
8 while (i < n)
9 invariant 0 <= i <= n;
10 invariant forall j::(0<=j<i) ==> b[j] == a[j];
11 {
12   b[i] := a[i];
13   i := i + 1;
14 }
15 }
```

Some bits of history ... (extra)

Kick off:

- **Checking a large routine**

Turing

Kick off:

- “**Checking a large routine**”

“How can one check a routine in the sense of making sure that it is right? In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.”

Alan Turing, 24th June 1949



Friday, 24th June.

Checking a large routine. by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

Consider the analogy of checking an addition. If it is given as:

1374
596
6719
4337
7768

26104

one must check the whole at one sitting, because of the carries.

But if the totals for the various columns are given, as below:

1374
596
6719
4337
7768

26104

Assertions

Second boost:

– Floyd's Assertion Method

Robert Floyd's, "Assigning Meanings to Programs," opened the field of program verification. His basic idea was to attach so-called "tags" in the form of logical assertions to individual program statements or branches that would define the effects of the program based on a formal semantic definition of the programming language.

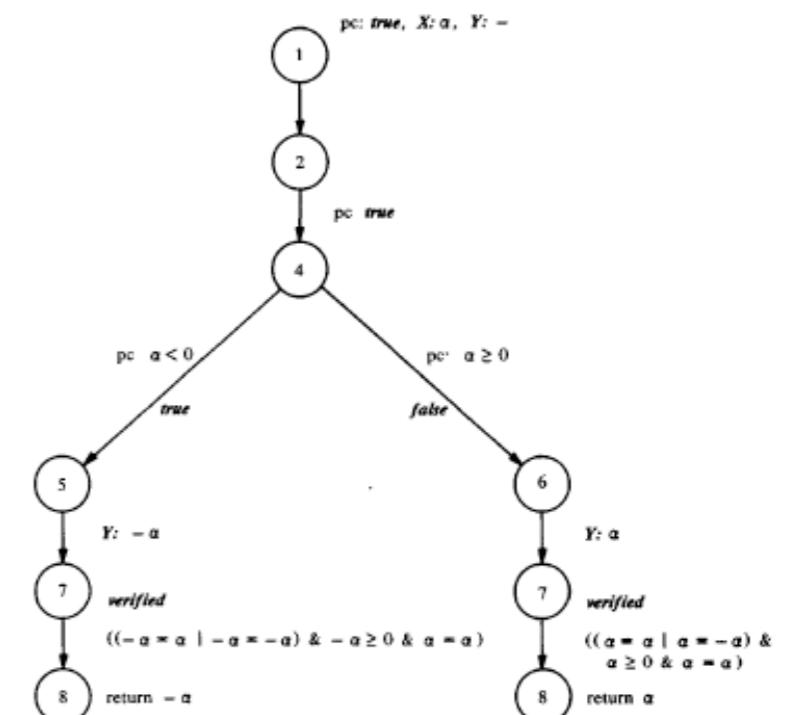


FIGURE 3. Symbolic execution tree for procedure ABSOLUTE.

R. Floyd, MFCS, June 1967

Assertions

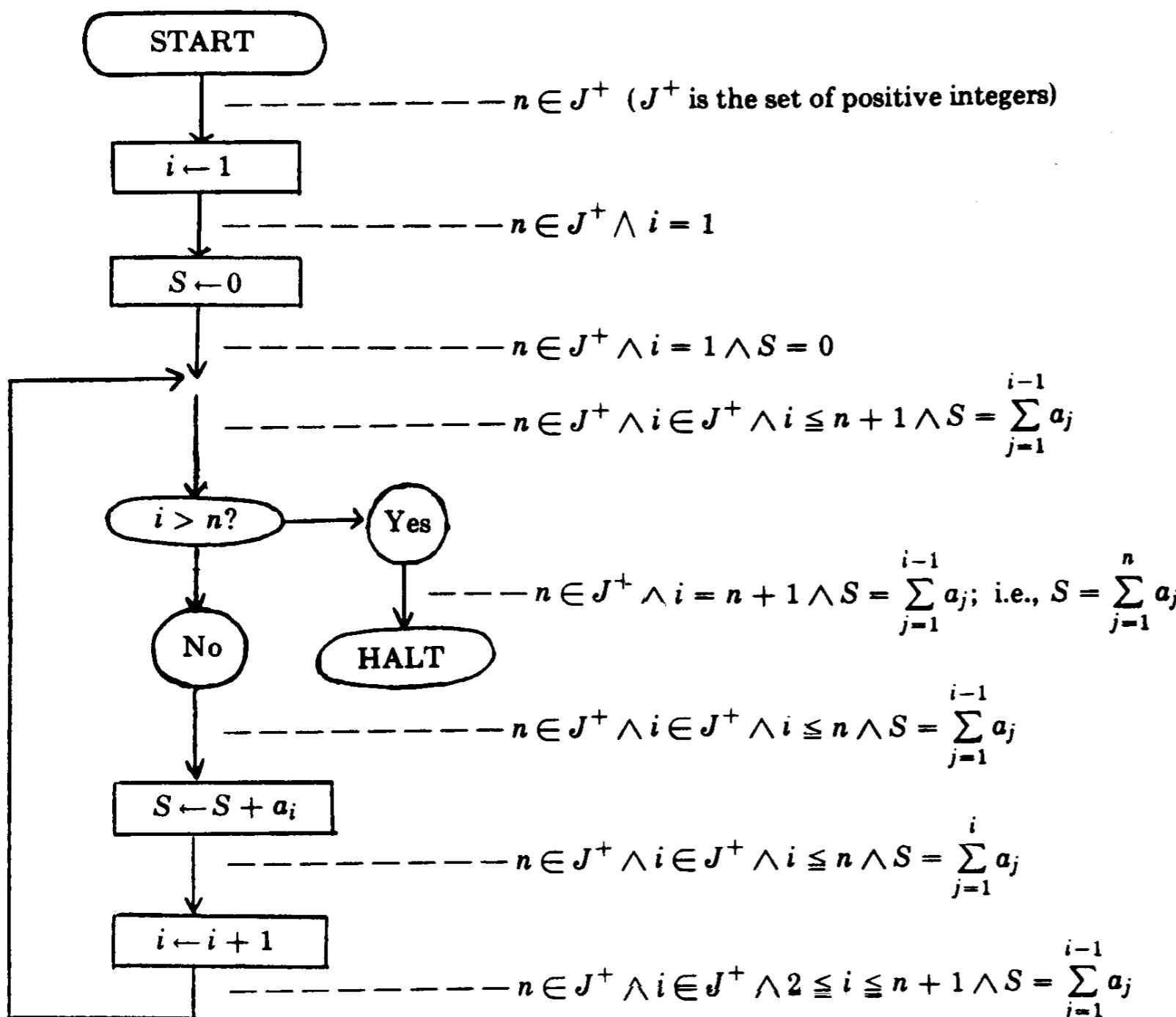


FIGURE 1. Flowchart of program to compute $S = \sum_{j=1}^n a_j$ ($n \geq 0$)

Language Based Program Specs

Lift Off:

– Hoare Logic

“Computer Programming is an exact science in that all the properties of a program and all consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning.”

Tony Hoare, CACM 1969



AXIOM 1: ASSIGNMENT AXIOM

$$\{p[t/x]\} \ x := t \ {p}.$$

RULE 2: COMPOSITION RULE

$$\frac{\{p\} S_1 \{r\}, \{r\} S_2 \{q\}}{\{p\} S_1; S_2 \{q\}}.$$

RULE 3: if-then-else RULE

$$\frac{\{p \wedge e\} S_1 \{q\}, \{p \wedge \neg e\} S_2 \{q\}}{\{p\} \text{ if } e \text{ then } S_1 \text{ else } S_2 \text{ fi } \{q\}}$$

RULE 4: while RULE

$$\frac{\{p \wedge e\} S \{p\}}{\{p\} \text{ while } e \text{ do } S \text{ od } \{p \wedge \neg e\}}$$

Hoare Logic Today

Still hot ...

– Hoare Logic

“The axiomatic method gives an objective criterion of the quality of a programming language, and the ease with which programmers could use it. The latest response comes from hardware designers, who are using axioms in anger to define the properties of modern multicore chips with weak memory consistency.”

Tony Hoare, CACM 2009

The screenshot shows a web browser displaying the Communications of the ACM website. The page title is "Retrospective: An Axiomatic Basis for Computer Programming | October 2009 | Communications of the ACM". The main content area features an article by C.A.R. Hoare titled "Retrospective: An Axiomatic Basis for Computer Programming". The article discusses the 40th anniversary of the publication of his first academic paper. It includes a sidebar with links to "this article" (Abstract, Full Text (HTML), Full Text (PDF), User Comments (0), In the Digital Edition, In the Digital Library) and "article contents" (Introduction, Retrospective (1969–1999), Progress (1999–2009), Prospective (2009–), The End, Author, Footnotes, Figures). To the right of the text is a black and white photograph of Tony Hoare attending a conference in 1969. Below the photo is a caption: "C.A.R. Hoare attending the NATO Software Engineering Techniques Conference in 1969. Credit: Robert M. McClure". At the bottom of the page is a section titled "Retrospective (1969–1999)" with a brief summary of his work on ALGOL 60.

Extended Static Checking

JML and Extended Static Checking for Java

ESC/Java2 is a programming tool that uses static analysis to verify the correctness of Java programs, using an extension of Hoare Logic called JML.

G.T. Leavens, 2000

Extended Static Checking for Java

Cormac Flanagan

Greg Nelson

K. Rustan M. Leino*

James B. Saxe

Mark Lillibridge

Raymie Stata

Compaq Systems Research Center 130 Lytton Ave. Palo Alto, CA 94301, USA

ABSTRACT

Software development and maintenance are costly endeavors. The cost can be reduced if more software defects are detected earlier in the development cycle. This paper introduces the Extended Static Checker for Java (ESC/Java), an experimental compile-time program checker that finds common programming errors. The checker is powered by verification-condition generation and automatic theorem-proving techniques. It provides programmers with a simple annotation language with which programmer design decisions can be expressed formally. ESC/Java examines the annotated software and warns of inconsistencies between the design decisions recorded in the annotations and the actual code, and also warns of potential runtime errors in the code. This paper gives an overview of the checker architecture and annotation language and describes our experience applying the checker to tens of thousands of lines of Java programs.

Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements/Specifications;
D.2.4 [Software Engineering]: Program Verification

General Terms

Design, Documentation, Verification

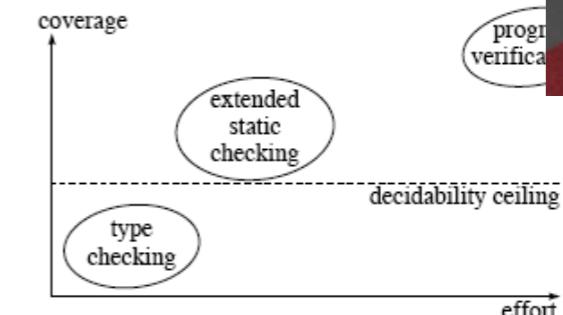


Figure 1: Static checkers plotted along the two dimensions of coverage and effort.



Extended Static Checking

Spec#

Spec# is an extension of the object-oriented language C#. It extends the type system to include non-null types and checked exceptions. It provides method contracts in the form of pre- and postconditions as well as object invariants.

Barnett, Leino, Schulte, 2004

The Spec# Programming System: An Overview

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte

Microsoft Research, Redmond, WA, USA

{mbarnett, leino, schulte}@microsoft.com

Manuscript KRML 136, 12 October 2004. To appear in CASSIS 2004 proceedings.

Abstract. The Spec# programming system is a new attempt at a more cost effective way to develop and maintain high-quality software. This paper describes the goals and architecture of the Spec# programming system, consisting of the object-oriented Spec# programming language, the Spec# compiler, and the Boogie static program verifier. The language includes constructs for writing specifications that capture programmer intentions about how methods and data are to be used, the compiler emits run-time checks to enforce these specifications, and the verifier can check the consistency between a program and its specifications.



Dafny

Dafny

Dafny is an imperative object-based language with built-in specification constructs. The Dafny static program verifier can be used to verify the functional correctness of programs. The specifications include pre- and postconditions, frame specifications (read and write sets), and termination metrics

Leino, Koenig, 2010

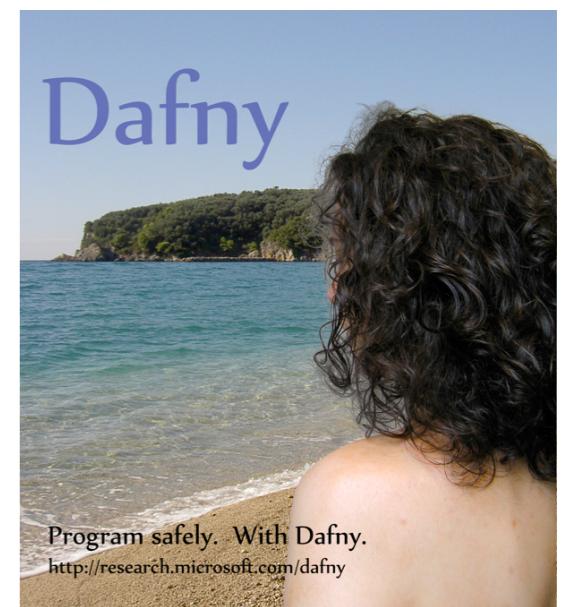
Dafny: An Automatic Program Verifier for Functional Correctness

K. Rustan M. Leino
Microsoft Research
leino@microsoft.com

Abstract

Traditionally, the full verification of a program's functional correctness has been obtained with pen and paper or with interactive proof assistants, whereas only reduced verification tasks, such as extended static checking, have enjoyed the automation offered by satisfiability-modulo-theories (SMT) solvers. More recently, powerful SMT solvers and well-designed program verifiers are starting to break that tradition, thus reducing the effort involved in doing full verification.

This paper gives a tour of the language and verifier Dafny, which has been used to verify the functional correctness of a number of challenging pointer-based programs. The paper describes the features incorporated in Dafny, illustrating their use by small examples and giving a taste of how they are coded for an SMT solver. As a larger case study, the paper shows the full functional specification of the Schorr-Waite algorithm in Dafny.



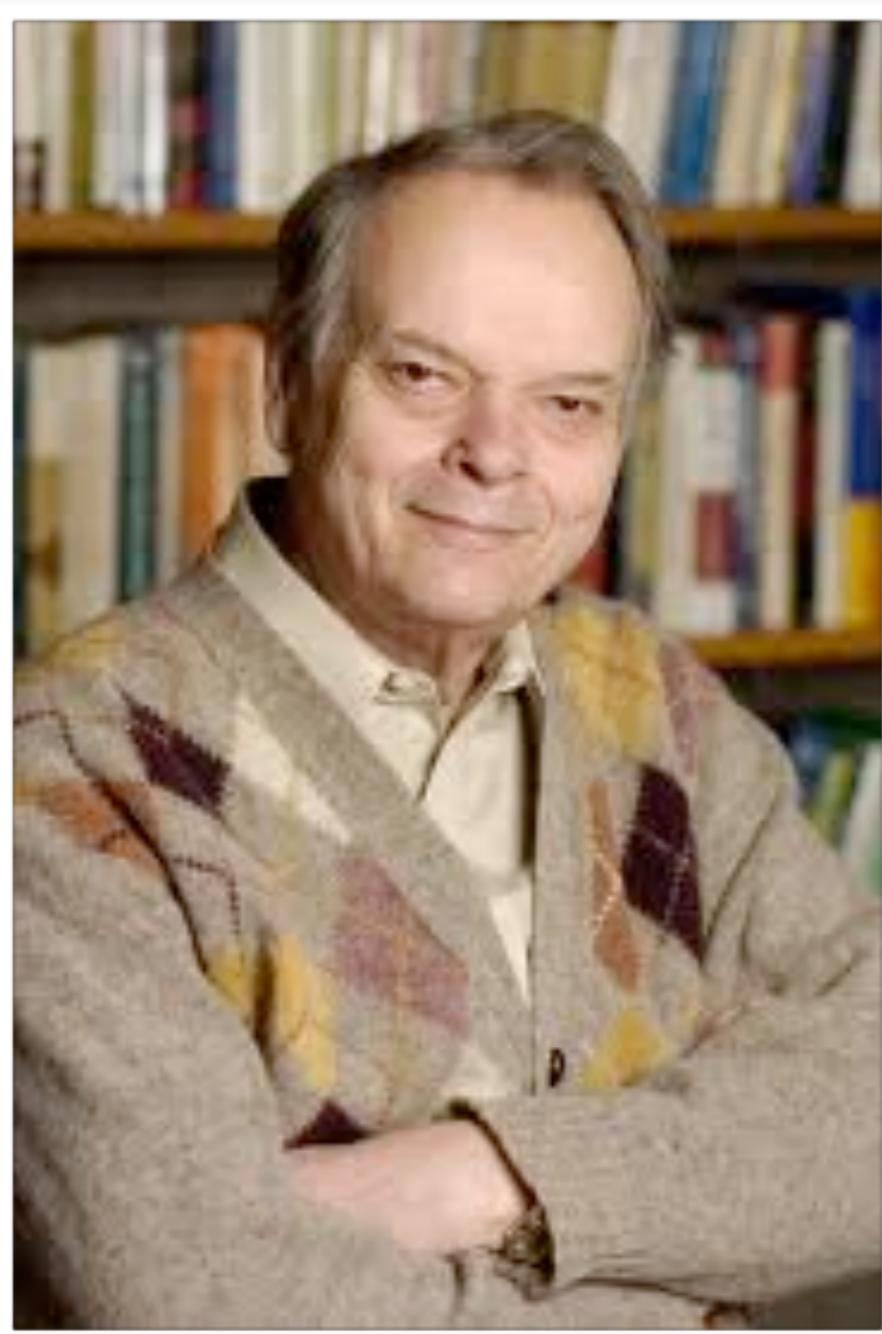
Program safely. With Dafny.
<http://research.microsoft.com/dafny>

rise4fun @ MSR

The screenshot shows the rise4fun website interface. At the top, there's a browser-like header with tabs for 'rise4fun.com' and other links like 'Reader', 'Download', and 'Conselho Científico'. Below the header, the main title 'rise4fun' is displayed with the subtitle '708285 programs analyzed'. A blue diagonal banner on the right says 'Your tool here'. The page features several sections of tool cards:

- new!**: Cards for **dafny** (A language and program verifier for functional correctness), **f*** (A verification tool for higher-order stateful programs), **visual c++** (The Visual C++ compiler), and **z3rcf** (Python interface for the Z3 Real Closed Fields package (and Theorem Prover)).
- microsoft**: Cards for **agl** (Automatic Graph Layout), **bek** (A domain specific language for writing and analyzing common string functions.), **boogie** (Intermediate Verification Language), and **chalice** (A language and program verifier for reasoning about concurrent programs).
- code contracts**: Language agnostic modular program verification and repair with abstract interpretation.
- counterdog**: Theorem-prover for Counterfactual Datalog.
- dafny**: A language and program verifier for functional correctness.
- dkal**: Distributed Knowledge Authorization Language.
- esm**: Empirical Software Engineering and Measurement Group.
- formula**: Formal Modeling Using Logic Programming and Analysis.
- try f#**: Programming language combining functional, object-oriented and scripting programming.
- f***: A verification tool for higher-order stateful programs.
- heapdbg**: Runtime heap abstraction.
- koka**: A function-oriented language with effect inference.
- pex**: Automatic test generation using Dynamic Symbolic Execution for .NET.
- poirot**: Poirot.

Separation Logic



John C. Reynolds



Peter O'Hearn

$$\frac{s, h \models P * (P -* Q)}{s, h \models Q} \quad \frac{\{P\} C \{Q\}}{\{P * R\} C \{Q * R\}} \text{ mod}(C) \cap \text{fv}(R) = \emptyset$$

Infer@Facebook

Infer

Docs Support Blog Twitter Facebook GitHub

A tool to detect bugs in Java and C/C++/Objective-C code before it ships

Infer is a static analysis tool - if you give Infer some Java or C/C++/Objective-C code it produces a list of potential bugs. Anyone can use Infer to intercept critical bugs before they have shipped to users, and help prevent crashes or poor performance.

GET STARTED

TRY INFER IN YOUR BROWSER

★ Star

Android and Java

Infer checks for null pointer exceptions, resource leaks, annotation reachability, missing lock guards, and concurrency race conditions in Android and Java code.

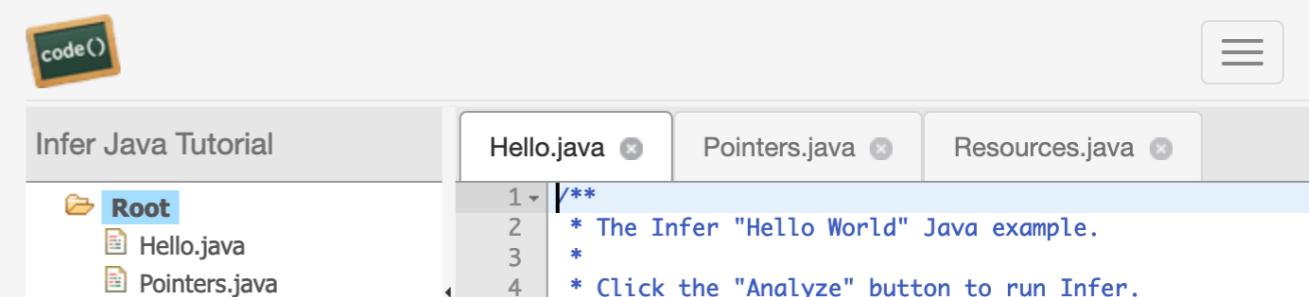
C, C++, and iOS/Objective-C

Infer checks for null pointer dereferences, memory leaks, coding conventions and unavailable API's.

Infer in Action

```
class Infer {  
    String mayReturnNull(int i) {  
        if (i > 0) {  
            return "Hello, Infer!";  
        }  
        return null;  
    }  
}
```

Try Infer



Verifast

Verifast

VeriFast is a verifier for single-threaded and multithreaded C and Java programs annotated with preconditions and postconditions written in separation logic.

*Jacobs, Smans, Piessens,
2010*

NB: separation logic is a spec language for talking about programs that allocate memory and use references

```
public void broadcast_message(String message) throws IOException
  //@ requires room(this) && message != null;
  //@ ensures room(this);
{
  //@ open room(this);
  //@ assert foreach(?members0,_);
  List membersList = this.members;
  Iterator iter = membersList.iterator();
  boolean hasNext = iter.hasNext();
  //@ length_nonnegative(members0);
  while (hasNext)
    /*@
     invariant
      foreach<Member>(?members, @member) && iter(iter, membersList, members, ?i)
        && hasNext == (i < length(members)) && 0 <= i && i <= length(members);
    */
    {
      Object o = iter.next();
      Member member = (Member)o;
      //@ mem_nth(i, members);
      //@ foreach_remove<Member>(member, members);
      //@ open member(member);
      Writer writer = member.writer;
      writer.write(message);
      writer.write("\r\n");
      writer.flush();
      //@ close member(member);
      //@ foreach_unremove<Member>(member, members);
      hasNext = iter.hasNext();
    }
  //@ iter_dispose(iter);
  //@ close room(this);
}
```



Rules of Hoare Logic

Program Proofs in Hoare Logic

- A program proof in Hoare logic adds assertions between program statements, making sure that all Hoare triples are satisfied.
- For example, consider the code snippet

```
if (x>y) {  
    z := x  
} else {  
    z := y  
}
```

Program Proofs in Hoare Logic

- A Hoare Logic “proof” may look like

```
{ true }

if ( $x > y$ ) {
    { ( $x > y$ ) }

     $z := x;$ 
    { ( $x > y$ )  $\&\&$  ( $z == x$ ) }

}

else {
    { ( $x \leq y$ ) }

     $z := y;$ 
    { ( $x \leq y$ )  $\&\&$  ( $z == y$ ) }

}

{ ( $x > y$ )  $\&\&$  ( $z == x$ )  $\|$  ( $x \leq y$ )  $\&\&$  ( $z == y$ )}
{  $z == \max(x, y)$  }
```

Using dafny

```
function max(x:int, y:int):int { if x > y then x else y }

method maxMethodZ(x:int, y:int) returns (z:int)
  ensures z == max(x,y)
{
  assert true;
  if (x>y) {
    assert x > y ;
    z := x;
    assert z > y && z == x;
    assert z >= y && z == x;
  } else {
    assert x <= y;
    z := y ;
    assert z >= x && z == y;
  }
  assert (z >= y && z == x) || (z >= x && z == y);
  assert z == max(x,y);
}
```

Example: Rule for Sequence

- A sequence defines a dependency on the effects of both program statements.

$$\frac{\{A\} P \{B\} \quad \{B\} Q \{C\}}{\{A\} P; Q \{C\}}$$

Rules of Hoare Logic (general form)

- The inference rules of Hoare logic are used to derive (valid) Hoare triples given some already derived Hoare triples

$$\frac{\{A_1\} P_1 \{B_1\} \dots \{A_n\} P_n \{B_n\}}{\{A\} C(P_1, \dots, P_n) \{B\}}$$

- What is nice here:
 - the program in the conclusion contains the subprograms P_1, \dots, P_n as components
 - we derive properties of the composite from the properties of its parts (compositionality)
 - pretty much the same as with a type system

“Structural” Proof Rules

- Basic logic proof systems operate on propositions, e.g.

$$\frac{A \quad A \Rightarrow B}{B} \quad \frac{A \quad B}{A \wedge B} \quad \frac{A}{A \vee B} \quad \frac{B}{A \vee B}$$

- Hoare logic proof system operates on Hoare triples, e.g.

$$\frac{\{A\} P \{B\} \quad \{B\} Q \{C\}}{\{A\} P; Q \{C\}}$$

One rule for each PL construct

AXIOM 1: ASSIGNMENT AXIOM

$$\{p[t/x]\} \ x := t \ {p}.$$

RULE 2: COMPOSITION RULE

$$\frac{\{p\} S_1 \{r\}, \{r\} S_2 \{q\}}{\{p\} S_1; S_2 \{q\}}.$$

RULE 3: if-then-else RULE

$$\frac{\{p \wedge e\} S_1 \{q\}, \{p \wedge \neg e\} S_2 \{q\}}{\{p\} \text{ if } e \text{ then } S_1 \text{ else } S_2 \text{ fi } \{q\}}$$

RULE 4: while RULE

$$\frac{\{p \wedge e\} S \{p\}}{\{p\} \text{ while } e \text{ do } S \text{ od } \{p \wedge \neg e\}}$$



- A really cool idea:
 - every programmer can use the Hoare rules informally to mentally check her code while coding
 - tools exist that automate most of the process
 - we now go through each rule, one by one

Simple Programming Language

$E ::=$	Expressions	
	num	Integer
	x	Variable
	$E + E \mid \dots$	Integer operators
	$E < E \mid \dots$	Relational operators
	$E \text{ and } E \dots$	Boolean operators
$P ::=$		Programs
	skip	No op
	$x := E$	Assignment
	$P; P$	Sequential Composition
	if E then P else P	Conditional
	while E do P	Iteration

Rule for Skip

$$\{A\} \text{ skip } \{A\}$$

Rule for Skip

$\{A\}$ skip $\{A\}$

```
if x < 0
{ x := -x; }
```

```
if x < 0
{ x := -x; }
else
skip
```

Example: Rule for Sequence

- A sequence defines a dependency on the effects of both program statements.

$$\frac{\{A\} P \{B\} \quad \{B\} Q \{C\}}{\{A\} P; Q \{C\}}$$

Rule for Conditional

$$\frac{\{A \wedge E\} P \{B\} \quad \{A \wedge \neg E\} Q \{B\}}{\{A\} \text{ if } E \text{ then } P \text{ else } Q \{B\}}$$

Rule for Deduction

$$\frac{A' \implies A \quad \{A\} P \{B\} \quad B \implies B'}{\{A'\} P \{B'\}}$$

- $A \Rightarrow B$ means “A logically implies B”
- We prove $A \Rightarrow B$ using the principles of first order logic, plus basic properties of the domain data types, e.g. properties of integers, arrays, etc.

Rule for Assignment

$$\{A[E/x]\} \ x := E \ \{A\}$$

- $A[E/x]$ means:
 - the result of replacing all free occurrences of variable x in assertion A by the expression E
 - For this rule to be sound, we require E to be an expression without side effects (a pure expression)

Rule for Assignment

$$\{A[x/E]\} \quad x := E \quad \{A\}$$

- We can think of A as a condition where “x” appears in some places. A is a condition dependent on “x”.
- The assignment $x := E$ changes the value of x to E, but leaves everything else unchanged
- So everything that could be said of E in the precondition, can be said of x in the postcondition, since the value of x after the assignment is E
- Example: $\{x + 1 > 0\} \quad x := x + 1 \quad \{x > 0\}$

Rule for Assignment

$$\{A[E/x]\} \ x := E \ {A}$$

- Example, let's check $\{x > -1\} x := x + 1 \{x > 0\}$

$\{ (x+1 > 0) \} x := x+1 \ {x > 0} \quad \text{by the } := \text{ Rule}$

that is, $\{ (x > 0)[x+1/x] \} \ x := (x+1) \ {x > 0}$

$\{x > -1\} x := x + 1 \ {x > 0} \quad \text{by deduction}$

Rule for Assignment

$$\{A[E/x]\} \ x := E \ \{A\}$$

- Trick: if x does not appear in E or A .

We can always write $\{ A \&& E == E \} x := E \{ x == E \}$

So, if x does not occur in E , A the triple

$$\{ A \} x := E \{ A \&& x == E \}$$

is always valid

Rule for Assignment

$$\{A[E/x]\} \ x := E \ \{A\}$$

- Exercises. Derive:
 - $\{y > 0\} x := y \{x > 0 \&\& y == x\}$
 - $\{x == y\} x := 2^*x \{y == x \text{ div } 2\}$
 - $\{P(y) \&\& Q(z)\}$ (here P and Q are any properties)
 $x := y ; y := z; z := x$
 $\{P(z) \&\& Q(y)\}$

Example

- Consider the program

$$P \triangleq \text{if } (x > y) \text{ then } z := x \text{ else } z := y$$

- We (mechanically) check the triple

$$\{ \text{true} \} P \{ z == \max(x,y) \}$$

Example

- Consider the program

$$P \triangleq \text{if } (x > y) \text{ then } z := x \text{ else } z := y$$

- We (mechanically) check the triple

$$\{ \text{true} \} P \{ z == \max(x,y) \}$$
$$\{ x == \max(x,y) \} \ z := x \{ z == \max(x,y) \}$$
$$\{ x > y \} \ z := x \{ z == \max(x,y) \}$$
$$\{ y == \max(x,y) \} z := y \{ z == \max(x,y) \}$$
$$\{ y >= x \} z := y \{ z == \max(x,y) \}$$