Modern trends in discrete mathematics and computer science

Formal correctness proofs for sequential programs

Lecturer:
Vladislav Podymov

E-mail:
valdus@yandex.ru

CMC MSU, 2019, spring semester

Introduction

Consider the following fragment of a program in C syntax:

```
int * a;
int n;
...
int s = 0;
for(int i = 0; i < n; ++i) s = s + a[i];
```

What does this fragment do?

The most obvious answer is:

it computes the sum of the elements of `a` *and stores it to* `s`

A less obvious better answer is:

if the data is okay at the beginning of the loop,
then *it computes the sum ...*

Is it the most correct answer?

Not so simple: it implicitly assumes that the meaning of the program complies with the C standard

Introduction

Consider the following fragment of a program in C syntax:

```
int * a;
int n;
...
int s = 0;
for(int i = 0; i < n; ++i) s = s + a[i];
```

How to **prove** that a program behaves the way it should?
(*in general, maybe even non-standard, or non-C-like*)

To do it, we need a lot of mathematics:

- What is a program? (syntax)
- What is a behavior of a program? (semantics)
- How to define a desired behavior?
  (borderline between semantics and proof systems)
- What is a legitimate proof? (proof systems)
- Can we even prove anything useful? (properties of proof systems)

Introduction

Consider the following fragment of a program in C syntax:

```
int * a;
int n;
...
int s = 0;
for(int i = 0; i < n; ++i) s = s + a[i];
```

The main focus is on while programs: a simplistic mathematical model intended to capture core features of all sequential programs

Though "while programs" is a certain specific model,
note that it provides a general mathematical machinery which allows to (*at least*) describe and (*at most*) analyze all kinds of programs

General and specific pieces of this machinery are mixed together in the talk, and separated and pointed out whenever possible

Syntax: types

Program data are usually **typed** (*whatever it means*)

A type is a **name** of a set of legitimate data values

While programs use two sorts of types:

- a basic type is a type of a primitive chunk of data
  - *whatever "primitive" means:*
    bounded or unbounded numbers, plain pointers, structures, ...
  - `Boolean` is a always assumed to be a basic type of Boolean values
  - `integer`, when mentioned, is assumed to be
    a basic type of all integer numbers
- a higher type is used to denote arrays and functions:
  - $T_1 \times \cdots \times T_n \to T$, where $T_1, \ldots, T_n, T$ are basic types, and $n \geq 1$
  - n is the arity of the type

Syntax: variables, constants

A variable is a **name** (**symbol**) which denotes a chunk of data of a certain **type**: the data can be accessed to and modified by a program via its name

A simple variable is a variable of a basic type

An array variable is a variable of a higher type

Var is the set of all variables

A constant is a **name** which denotes a certain value of a certain type

A simple constant is a constant of a basic type

A functional symbol is a constant of a higher type

A relational symbol is a functional constant of a type

$$T_1 \times \cdots \times T_n \to Boolean$$

Const is the set of all constants

Syntax: typical constants

true, **false** are simple constants of the Boolean type

$0, 1, -1, 2, -2, \dots$ are simple constants of the integer type

$<, \leq, >, \geq, =, \neq, \dots$ are relational symbols of the type

$$\mathit{integer} \times \mathit{integer} \to \mathit{Boolean}$$

$+, -, *, /, \dots$ are functional symbols of the type

$$\mathit{integer} \times \mathit{integer} \to \mathit{integer}$$

$\&, \vee, \neg, \to, \dots$ are relational symbols of the type

$$\mathit{Boolean} \times \mathit{Boolean} \to \mathit{Boolean}$$

Syntax: expressions

Example: $x + 1 < y$

An **expression** of a type T:

- is a string constructed from variables and constants with respect to their types and arities
- intuitively, for certain current data values provides a value of the type T
  - *not a definition: what do "a **value**" and "**provides**" mean?*

Backus-Naur form (BNF) for an expression (ε):

$$\varepsilon \ ::= \ c \ \mid \ x \ \mid \ a[\varepsilon_1, \ldots, \varepsilon_n] \ \mid \ f(\varepsilon_1, \ldots, \varepsilon_n)$$

- c is a simple constant
- x is a simple variable
- a is an array variable of a type $T_1 \times \cdots \times T_n \to T$
- f is a functional symbol of a type $T_1 \times \cdots \times T_n \to T$
- ε_i is an expression of the type T_i
- $a[\varepsilon_1, \ldots, \varepsilon_n]$ and $f(\varepsilon_1, \ldots, \varepsilon_n)$ are expressions of the type T

Infix notation for binary functional symbols:

$\oplus(\varepsilon_1, \varepsilon_2)$ equals to $(\varepsilon_1 \oplus \varepsilon_2)$

Syntax: subscripted variables, programs

An expression of the form $a[\varepsilon_1, \ldots, \varepsilon_n]$ is a subscripted variable:

- it is not a variable "in the full sense"
- it refers to a primitive chunk of data
  (just like a *nonsubscripted* variable)

BNF for a program (π):

$$
\begin{array}{lcl}
\pi & ::= & \mathit{stmt} \mid \mathit{stmt}\ \pi \\
\mathit{stmt} & ::= & \mathit{skip}; \mid x := \varepsilon_x; \mid \\
 & & \textbf{if } \varepsilon_b \textbf{ then } \pi \textbf{ else } \pi \textbf{ fi}; \mid \textbf{while } \varepsilon_b \textbf{ do } \pi \textbf{ od};
\end{array}
$$

- stmt is a statement
  - *a program is a nonempty sequence of statements*
  - *any statement is a program*
- x is a subscripted variable
- ε_x is an expression of the same type as x
- ε_b is a Boolean expression

Π is the set of all programs

Syntax: programs

π_1: $abcd$

π_2: $skip$;

π_3: **if** $b \vee c[3]$ **then** $x[y[z], 1] := z; z := 2;$ **else** $skip$; **fi**;

π_1 is **not** a program

π_2 is a program

π_3 is a program iff

1. b is a simple Boolean variable
2. c is a variable of the type $integer \rightarrow Boolean$
3. z is a simple integer variable
4. y is a variable of a type $integer \rightarrow T$
5. x is a variable of the type $T \times integer \rightarrow integer$

Semantics

$$\textbf{if } b \vee c[3] \textbf{ then } x[y[z],1] := z; z := 2; \textbf{ else } \mathit{skip}; \textbf{ fi};$$

At this point we are able to distinguish programs from non-programs, but know nothing about their meaning (semantics):

- What "values" a program works with, and how these values are related to types
- What value is "provided" by an expression
- What does each statement of a program mean
- How the meanings of statements are combined into the meaning of the whole program

Semantics: values, domains

A domain D_T of a type T is a set of values of this type

To specify a certain domain, we should *at least*

- pick a certain programming language
- determine a goal of a program analysis
  - *for instance, if we do not care about extreme overflow cases usual for "real" modular arithmetic, then we may use a "simplified" unbounded arithmetic instead*

For while programs, the following domains are fixed:

- $D_{Boolean} = \{\mathbf{true}, \mathbf{false}\}$
- $D_{integer} = \{0, 1, -1, 2, -2, \dots\}$
- $D_{T_1 \times \cdots \times T_n \to T}$ is the set of all functions from the Cartesian product $D_{T_1} \times \cdots \times D_{T_n}$ into the set D_T

A semantic domain D is a **disjoint** union of domains of all types

Semantics: interpretations

Given a set of constants, a set of types, and type domains, an interpretation $\mathcal{I}$ is a mapping of every constant of every type T to an element of D_T

An interpretation is a natural mathematical way
to define a "static" semantical part of a programming language:
the meaning of constants, operations, predefined functions, ...

For instance, a typical (*but not the only*) interpretation $\mathcal{I}$ of Boolean-related and integer-related constants is defined as follows:

- each simple constant, Boolean or integer, is mapped into itself:
  - $\mathcal{I}(\mathbf{true}) = \mathbf{true}$, $\mathcal{I}(2) = 2$, ...
- each typical functional symbol is mapped into a function in a natural way:
  - $\mathcal{I}(\vee)(\mathbf{true}, \mathbf{false}) = \mathbf{true}$
  - $\mathcal{I}(+)(2, 3) = 5$
  - $\mathcal{I}(<)(5, 2) = \mathbf{false}$
  - ...

Semantics: data states

A data state is

- (*informally*) a collection of values stored at any given time in all data chunks managed by a program (and accessed via variable names)
- (*formally*) a mapping $\sigma : Var \to D$,
  such that for each variable of a type T, $\sigma(x)$ is a value of the domain D_T

A data state is a "dynamic" part of a programming language: an execution of a sequential program is a stepwise modification of a current data state

Σ is the set of all data states

$\{x_1/val_1, \ldots, x_n/val_n\}$ is a state σ such that $Var = \{x_1, \ldots, x_n\}$, and $\sigma(x_i) = val_i$ for each i, $1 \leq i \leq n$

Semantics: expressions

$$x + 3$$

Now, having a huge spectre of definitions, we finally can answer the (*apparently, not so simple*) question

"What does an expression mean?"

First of all, we pick a certain programming language, and fix its "static" part

- mathematically, we assume an interpretation $\mathcal{I}$ to be given

A value provided by an expression is fully defined by a data state obtained at a given execution time

- mathematically, the semantics of an expression ε of a type T is the following mapping $\mathcal{I}\,[\![\varepsilon]\!] : \Sigma \to D_T$:
  - for each simple constant c, $\mathcal{I}\,[\![c]\!](\sigma) = \mathcal{I}(c)$
  - for each simple variable x, $\mathcal{I}\,[\![x]\!](\sigma) = \sigma(x)$
  - for each expression ε of the form $a[\varepsilon_1, \ldots, \varepsilon_n]$,
    $$\mathcal{I}\,[\![\varepsilon]\!](\sigma) = \sigma(a)(\mathcal{I}\,[\![\varepsilon_1]\!]\,(\sigma), \ldots, \mathcal{I}\,[\![\varepsilon_n]\!]\,(\sigma))$$
  - for each expression ε of the form $f(\varepsilon_1, \ldots, \varepsilon_n)$,
    $$\mathcal{I}\,[\![\varepsilon]\!](\sigma) = \mathcal{I}(f)(\mathcal{I}\,[\![\varepsilon_1]\!]\,(\sigma), \ldots, \mathcal{I}\,[\![\varepsilon_n]\!]\,(\sigma))$$

Semantics: variety of definition approaches

π: **if** b **then** $x := x + 1$; **else** *skip*; **fi**;

The next question is:

"**What does a program mean?**"

First of all, the main purpose of a (sequential) program is to

- take some initial data values (input data state)
- process these values
- provide some final data values
  depending on the initial ones (output data state)

"To define the meaning of a program" basically means
"to define a relation between input and output data states"

- mathematically, the semantics of a program π is a relation
  $\mathcal{I}\,[\![\pi]\!] \subseteq \Sigma \times \Sigma$ between input and output data states
- for some programming languages this relation is a total function,
  for some — a partial function,
  for some — a multivalued function (i.e. relation in a full sense)

Semantics: variety of definition approaches

π: **if** b **then** $x := x + 1$; **else** *skip*; **fi**;

Even when a programming language is picked, a lot of approaches exist on how to define a semantics of a program

The most popular ones are:

- operational approach
  - a data state is modified step by step
    during a statement execution in the following way: ...
  - the next statement to be executed after the current statement is: ...
  - if a stepwise statement execution is finished, then the output state is: ...

Semantics: variety of definition approaches

π: **if** b **then** $x := x + 1$; **else** *skip*; **fi**;

Even when a programming language is picked, a lot of approaches exist on how to define a semantics of a program

The most popular ones are:

- denotational approach
  - semantics of a primitive statement
    is the following binary relation over data states: ...
    - *the relation is represented as a formula of a language designed specifically for declarative description of computable relations*
  - semantics of a complex statement
    is the following composition of relations: ...
    - *the composition is a simple syntactic modification of given formulae which declaratively describes some nontrivial transformations over relations*
    - *for instance, a minimization operator for μ-recursive functions is syntactically simple, but semantically rather nontrivial*

Semantics: variety of definition approaches

π: **if** b **then** $x := x + 1$; **else** *skip*; **fi**;

Even when a programming language is picked, a lot of approaches exist on how to define a semantics of a program

The most popular ones are:

- axiomatic approach:
  - an *assertion* is a formula (of a special purely-logical language) which represents a set of data states
  - a *rule* for a primitive statement is a set of pairs of assertions
  - a *rule* for a complex statement says how pairs of assertions obtained for substatements are transformed into ones for the statement
    - *just like in Hoare logic, if we speak about while programs*

Operational semantics: big-step and small-step

π: **if** b **then** $x := x + 1$; **else** *skip*; **fi**;

The most popular and well-known variations of an operational approach to define program semantics are:

- natural (big-step) semantics
  - each statement defines an input-output relation which says how the data is modified when the statement is fully executed
  - *for instance, "if the input for the statement* π *is* $\{b/\textbf{true}, x/2\}$*, then the output is* $\{b/\textbf{true}, x/3\}$*"*
- structural (small-step) semantics
  - each complex statement defines how the data is modified by the next most primitive execution step, and explicitly — what statement describes the rest of the execution
  - *for instance, "if the input for* π *is* $\{b/\textbf{true}, x/2\}$*, and "to pick a branch" is a primitive execution step, then then next data state is still* $\{b/\textbf{true}, x/2\}$*, and the rest statement is* $x := x + 1;$*"*

Small-step semantics: state update

For a data state σ, a subscripted variable x of a type T, and a value val of the same type T, $\sigma[x \leftarrow val]$ is a data state which differs from σ as follows:

- if x is a simple variable, then $\sigma[x \leftarrow val](x) = val$, and all other variables are mapped to the same values as by σ
- if x is a subscripted variable (equals to $a[\varepsilon_1, \ldots, \varepsilon_n]$ for clarity), then
  - $\sigma[x \leftarrow val](a)(\sigma(\varepsilon_1), \ldots, \sigma(\varepsilon_n)) = val$,
  - images of all other arguments of the function $\sigma[x \leftarrow val](a)$ equal to the corresponding images of $\sigma(a)$ (i.e. the rest of the array remains unchanged)
  - all variables, except a, are mapped to the same values as by σ

Small-step semantics of while programs

$\rightarrow_{\mathcal{I}}$ is a binary relation over $\Pi \times \Sigma$ which defines a small-step semantics of while programs operating in context of an interpretation $\mathcal{I}$: $\langle \pi, \sigma \rangle \rightarrow_{\mathcal{I}} \langle \pi', \sigma' \rangle$ means that a primitive execution step of the statement π on the input data σ leads to the output data σ', and the "rest" statement is π'

- $\langle x := \varepsilon;, \sigma \rangle \rightarrow_{\mathcal{I}} \langle skip;, \sigma[x \leftarrow \mathcal{I}\llbracket \varepsilon \rrbracket(\sigma)] \rangle$
- if $\langle \pi_1, \sigma \rangle \rightarrow_{\mathcal{I}} \langle \pi_1', \sigma' \rangle$, then $\langle \pi_1 \pi_2, \sigma \rangle \rightarrow_{\mathcal{I}} \langle \pi_1' \pi_2, \sigma' \rangle$
- $\langle skip; \pi, \sigma \rangle \rightarrow_{\mathcal{I}} \langle \pi, \sigma \rangle$
- if $\mathcal{I}\llbracket \varepsilon \rrbracket(\sigma) = \textbf{true}$,
  then $\langle \textbf{if } \varepsilon \textbf{ then } \pi_1 \textbf{ else } \pi_2 \textbf{ fi};, \sigma \rangle \rightarrow_{\mathcal{I}} \langle \pi_1, \sigma \rangle$,
  otherwise $\langle \textbf{if } \varepsilon \textbf{ then } \pi_1 \textbf{ else } \pi_2 \textbf{ fi}; \rangle \rightarrow_{\mathcal{I}} \langle \pi_2, \sigma \rangle$
- if $\mathcal{I}\llbracket \varepsilon \rrbracket(\sigma) = \textbf{false}$,
  then $\langle \textbf{while } \varepsilon \textbf{ do } \pi \textbf{ od};, \sigma \rangle \rightarrow_{\mathcal{I}} \langle skip;, \sigma \rangle$,
  otherwise $\langle \textbf{while } \varepsilon \textbf{ do } \pi \textbf{ od}; \rangle \rightarrow_{\mathcal{I}} \langle \pi \textbf{ while } \varepsilon \textbf{ do } \pi \textbf{ od};, \sigma \rangle$

$(\sigma_i, \sigma_o) \in \mathcal{I}(\pi)$ iff there exists a sequence of states

$$\langle \pi, \sigma_i \rangle \rightarrow_{\mathcal{I}} \cdots \rightarrow_{\mathcal{I}} \langle skip;, \sigma_o \rangle$$

Small-step semantics: example

Let $\mathcal{I}$ be a typical interpretation, and

π: **while** $x < 3$ **do if** $x > 1$ **then** $x := x + 2$; **else** $x := x + 1$; **fi**; **od**;

Then

$$\begin{array}{l}
\langle \pi, \{x/2\} \rangle \rightarrow_{\mathcal{I}} \\
\langle \textbf{if } x > 1 \textbf{ then } x := x + 2; \textbf{ else } x := x + 1; \textbf{ fi}; \pi, \{x/2\} \rangle \rightarrow_{\mathcal{I}} \\
\langle x := x + 2; \pi, \{x/2\} \rangle \rightarrow_{\mathcal{I}} \\
\langle \mathit{skip}; \pi, \{x/4\} \rangle \rightarrow_{\mathcal{I}} \\
\langle \pi, \{x/4\} \rangle \rightarrow_{\mathcal{I}} \\
\langle \mathit{skip};, \{x/4\} \rangle
\end{array}$$

Thus, $(\{x/2\}, \{x/4\}) \in \mathcal{I}(\pi)$

Other sequential programs

Now (*at last!*) we have mathematical means to describe **any** sequential program and its behavior:

- pick any programming language
- formalize a type system and type domains of the language
- write down syntactic rules of the language
  (*all modern languages have those*)
- carefully define semanitcs of all "static" components of the language, and then — its "dynamic" part: a small-step semantics

But why do we need it?

(*here go standard phrases about the critical importance of error-free programs, and about the rigorousness of mathematics*)

Now the big question is:
How can we prove anything about the absence of program errors?

Proof systems

How can we prove anything about the absence of program errors?

The question is much bigger than it seems:

Suppose someone gave you a random sequence of words similar to what is usually written in "Proof" sections of mathematical papers, and said
"**done, this proves that the program is error-free**" —
what means should you use to check mathematical consistency of a proof?

Exercise: *take any (old enough) scientific paper on correctness of distributed algorithms, and find an implicit assumption or an inconsistency which makes the main result "not as complete and valid as it seemed to be"* ☺

Proof systems

How can we prove anything about the absence of program errors?

One of the ways to lower the necessity of proof-checking is to formalize a proof as another mathematical object

Still, you need to prove that the mathematical definition of a proof is consistent (and introduce and solve several other problems), but once it is done, all well-formed formalized proofs become inherently proof-checked

The most famous collection of mathematical notions of a proof is known by many names, including:

proof systems, formal systems, deductive systems, and logical calculi

Proof systems

A proof (*recall: a random collection of words in a "Proof" section*) contains a sequence of *propositions*, such as:

- *(... thus,)* ***the sequence*** s ***is convergent****. (...)*
- *(... by definition of a field and Lemma 5,)*
  the ring $\mathcal{R}$ ***is a field****. (Q.E.D.)*
- *(... assuming that* $P \neq NP$*,)*
  the considered problem is hard*(, which implies ...)*

The first step to formalize a proof is to introduce a formal language of considered propositions

Proof systems: formulae

A proof system starts with the notion of a formula:

- an alphabet is a set of symbols,
  and each formula is a finite sequence of these symbols
- syntactic rules define which sequences of symbols are formulae,
  and which are not
  - typically, a collection of syntactic rules is a *BNF*
- intuitively, each formula corresponds to a certain *proposition* of a proof
- but the only strict meaning of a formula is the formula itself,
  if no additional definitions are provided

Several well-known examples of formulae:

- Boolean formulae: $x \mathbin{\&} y \to z$
- first-order formulae: $\forall x (\exists y\ (x > y) \vee Q(x)) \to \exists x\ R(x)$
- temporal formulae: $\mathbf{G}(\mathit{request} \to \mathbf{F}\mathit{response})$
- Hoare triples: $\{x > y\}\, x := x + y;\, \{x < y\}$

Proof systems: axioms

Some of the formulae correspond to propositions which require no proof (or proved a priori), for instance:

- addition is commutative: $\forall x \forall y (x + y = y + x)$
- a sequence is convergent iff <*here goes the definition*>:
  $$\forall s(convergent(s) \leftrightarrow \forall \varepsilon(real(\varepsilon)\,\&\,\varepsilon > 0 \rightarrow \exists N(\ldots)))$$
- every cow is an animal: $\forall c(cow(c) \rightarrow animal(c))$
- a program π computes $(x + y)$ and stores it to z:
  $$\{x = x_0\,\&\,y = y_0\}\,\pi\,\{z = x_0 + y_0\}$$
  (*if it is agreed to be okay to leave such a proposition unproved*)

Such formulae are called axioms

Proof systems: inference rules

A proof is usually not just "some random sequence" of propositions: the truth of each proposition "rationally" follows from the truth of previous propositions

An inference rule is a finite description of a relation between formulae:
a tuple $(f_1, \ldots, f_n, f)$ is an element of the relation iff f follows from $f_1, \ldots, f_n$
("if propositions $f_1, \ldots, f_n$ are proved, then f is also proved")

Inference rules are often (*but not always*) presented in the following form:

$$\frac{\varphi_1, \ldots, \varphi_n}{\varphi}$$

- φ_i and φ are formula schemata:
  formulae, some parts of which are replaced by *parameter names*
- all tuples $(f_1, \ldots, f_n, f)$ of the corresponding relation
  are obtained from the schemata $(\varphi_1, \ldots, \varphi_n, \varphi)$
  by replacement of parameter names with certain strings

Proof systems: inference rules

Several examples of inference rules:

- modus ponens:
  to prove B, it is sufficient to prove *a)* A, and *b)* that A implies B
  $$\frac{A, A \rightarrow B}{B}$$
- Small-step inference rules for while programs:
  $$\frac{\langle \pi_1, \sigma \rangle \rightarrow_{\mathcal{I}} \langle \pi_1', \sigma' \rangle}{\langle \pi_1 \pi_2, \sigma \rangle \rightarrow_{\mathcal{I}} \langle \pi_1' \pi_2, \sigma' \rangle}$$
- Hoare inference rules: ...
- Any reliable rules you want to use:
  $$\frac{cow(x)}{animal(x)}$$

Proof system: derivation

Now we can tell how to formally prove anything

First of all,

- define the notion of a *formula*:
  say what propositions can be used in a proof
- define a set of *axioms*: say what propositions are absolutely true
- define a set of *inference rules*: say what proof methods are rational

A derivation is a sequence $f_1, f_2, \ldots, f_k$ of formulae
such that for each i, $1 \leq i \leq k$,

- either f_i is an axiom
- or $(f_{j_1}, \ldots, f_{j_n}, f_i)$ is an element of a relation corresponding to any inference rule, and $j_1, \ldots, j_n < i$

A formula f is provable iff there exists a derivation $f_1, \ldots, f_k$ such that $f_k = f$

Hoare proof system

How can we prove anything about the absence of program errors?

A formula of a Hoare proof system (a Hoare triple) has the following form: $\{\varphi\}\,\pi\,\{\psi\}$, where φ and ψ are first-order formulae (*which have a signature compliant with the signature of* π), and π is a program

Intuitively, the triple corresponds to the following proposition:

- *partial correctness*: for any input data state satisfying φ, if π has an output data σ, then σ satisfies ψ
- *total correctness*: for any input data state satisfying φ, π has an output data σ, and σ satisfies ψ

Hoare proof system

Hoare axioms and inference rules
for partial correctness of while programs in an interpretation $\mathcal{I}$
(*recall all the courses in which Hoare logic was mentioned*):

axioms: all first-order formulae valid in $\mathcal{I}$

axioms: $\{\varphi\}\ \mathit{skip};\ \{\varphi\}$

axioms: $\{\varphi\{x/\varepsilon\}\}\ x := \varepsilon;\ \{\varphi\}$
(*the expression [term]* ε *should be "good enough"*)

rule: $$\frac{\varphi \to \varphi', \{\varphi'\}\,\pi\,\{\psi'\}, \psi' \to \psi}{\{\varphi\}\,\pi\,\{\psi\}}$$

rule: $$\frac{\{\varphi\}\,\pi_1\,\{\chi\}, \{\chi\}\,\pi_2\,\{\psi\}}{\{\varphi\}\,\pi_1\pi_2\,\{\psi\}}$$

rule: $$\frac{\{\varphi \,\&\, C\}\,\pi_1\,\{\psi\}, \{\varphi \,\&\, \neg C\}\,\pi_2\,\{\psi\}}{\{\varphi\}\ \mathbf{if}\ C\ \mathbf{then}\ \pi_1\ \mathbf{else}\ \pi_2\ \mathbf{fi};\ \{\psi\}}$$

rule: $$\frac{\{\varphi \,\&\, C\}\,\pi\,\{\varphi\}}{\{\varphi\}\ \mathbf{while}\ C\ \mathbf{do}\ \pi\ \mathbf{od};\ \{\varphi \,\&\, \neg C\}}$$

Hoare proof system

Hoare axioms and inference rules
for **total** correctness of while programs in an interpretation $\mathcal{I}$
(*that is something new*):

axioms: all first-order formulae valid in $\mathcal{I}$

axioms: $\{\varphi\}\, \mathit{skip};\, \{\varphi\}$

axioms: $\{\varphi\, \{x/\varepsilon\}\}\, x := \varepsilon;\, \{\varphi\}$
(*the expression [term]* ε *should be "good enough"*)

rule: $\dfrac{\varphi \to \varphi',\, \{\varphi'\}\, \pi\, \{\psi'\}\, ,\, \psi' \to \psi}{\{\varphi\}\, \pi\, \{\psi\}}$

rule: $\dfrac{\{\varphi\}\, \pi_1\, \{\chi\}\, ,\, \{\chi\}\, \pi_2\, \{\psi\}}{\{\varphi\}\, \pi_1 \pi_2\, \{\psi\}}$

rule: $\dfrac{\{\varphi \,\&\, C\}\, \pi_1\, \{\psi\}\, ,\, \{\varphi \,\&\, \neg C\}\, \pi_2\, \{\psi\}}{\{\varphi\}\, \textbf{if } C \textbf{ then } \pi_1 \textbf{ else } \pi_2 \textbf{ fi};\, \{\psi\}}$

rule: $\dfrac{\{\varphi \,\&\, C\}\, \pi\, \{\varphi\}\, ,\, \{\varphi \,\&\, C \,\&\, \varepsilon = z\}\, \pi\, \{\varepsilon < z\}\, ,\, \varphi \to \varepsilon \geq 0}{\{\varphi\}\, \textbf{while } C \textbf{ do } \pi \textbf{ od};\, \{\varphi \,\&\, \neg C\}}$

(*z is an integer variable not present in π*)

Hoare proof system

Another big question is: do such proof systems actually work?

For instance,

- the following axiom schema breaks everything: $\{\varphi\}\,\pi\,\{\psi\}$
- the following inference rule breaks everything:
$$\frac{\{\varphi\}\,\pi\,\{\psi\}}{\{\varphi'\}\,\pi\,\{\psi'\}}$$
- the absence of any axiom or any inference rule breaks a lot

Provable formulae are defined syntactically, but we need semantics to check that what we formally prove is exactly what we intuitively want

Proof systems: soundness and completeness

To measure the quality of a proof system (*in general*), all formulae are divided into valid and invalid: intuitively, a valid formula is a formula which corresponds to a true proposition

You *probably* know what is a validiy:

- "a first-order formula is valid in an interpretation $\mathcal{I}$": *true for the meaning of constants defined by $\mathcal{I}$, and for all valuations of free variables*
- "a Hoare triple is valid in an interpretation $\mathcal{I}$": *this validity slightly differs in case of partial and total correctness proofs, and complies with the corresponding intuitive meaning*

Proof systems: soundness and completeness

A proof system (*in general*) is sound iff all provable formulae are valid

A proof system is complete iff all valid formulae are provable

Magically (*by the results of a long research and a couple of heavy theorems*), Hoare proof systems for partial and total correctness are both **sound**, and moreover,

- the system for **partial** correctness is **complete**
- if integer expressions are *expressive enough* (for instance, to represent all computable functions), then the system for **total** correctness is alse **complete**

Too good to be effectively true:
the problem is hidden in the axioms, and in the (un)decidability

That's all. Questions?