FMSoft
Lecture 8 — beginning Hoare logic
(lecture version)

Tadeusz Litak

Dec 4, 2018

Informatik 8, FAU Erlangen-Nürnberg

Arithmetical expressions

The class of arithmetical expressions AExp is defined as

$a_1, a_2 ::= \texttt{z} \mid \texttt{X} \mid a_1 \texttt{ + } a_2 \mid a_1 \texttt{ - } a_2 \mid a_1 \texttt{ * } a_2$

where

- for each $z \in \mathbb{Z}$, z is a corresponding constant
- X belongs to the collection ProgVar of program variables

  The term has its disadvantages. Some references, including the Winskel book, are using the term locations, but this also can be confusing

More on AExp

- In practice, ProgVar would be indexed by natural numbers
- The two primitive types of AExp would then be obtained by constructors `const z` and `var n`, respectively
- As a trivial exercise, you can implement them as a datatype in your favourite programming language.

  This, of course, is a starting point for building a parser for IMP
- But using definitions from GLoIn, we could also notice in passing that elements of AExp are exactly what one would obtain as terms for signature

  $\{\ \texttt{+}/2,\ \texttt{-}/2,\ \texttt{*}/2,\ \texttt{z}/0 \mid z \in \mathbb{Z}\}$

  We will see later what to do with ProgVar: are they really "variables" in the same sense as those in GLoIn?

The class of Boolean expressions BExp is defined as

$$b_1, b_2 ::= \texttt{true} \mid \texttt{false} \mid a_1 \texttt{ == } a_2 \mid a_1 \texttt{ <= } a_2 \mid a_1 \texttt{ >= } a_2 \mid \texttt{not } b_1 \mid b_1 \texttt{ and } b_2 \mid b_1 \texttt{ or } b_2 .$$

Finally, the set of commands Com is defined as

$$c_1, c_2 ::= \texttt{SKIP} \mid \texttt{X := } a \mid c1 \texttt{ ; } c2 \mid \texttt{IF } b \texttt{ THEN } c_1 \texttt{ ELSE } c_2 \texttt{ ENDIF} \mid \texttt{WHILE } b \texttt{ DO } c_1 \texttt{ END}$$

You can proceed with implementing these syntactic sets as datatypes; and indeed, if you're ambitious, with implementing a full-blown parser

We know now how expressions of IMP look like ...But what do they mean?

Semantics by syntax: annotating code

- Our first glimpse at the axiomatic approach
- The meaning of the program is what it is supposed to achieve
- ...expressed syntactically via its correctness assertions
- Ideally, one builds a program together with its specification and verification conditions
- But we can also annotate it post factum

- In other words, we want a calculus/a proof system ...
- ...which derives *compositionally* judgements

$$\vdash \{A_1\}c\{A_2\}$$

- *if* A_1 *holds before execution of* c*, then* A_2 *holds afterwards*
- A_1 (precondition) and A_2 (postcondition) are called assertions
- The whole thing is called Hoare triple
- Note ambiguity: what if c does not terminate?
- One of reasons why you need good semantic semantics ...
- Normally, we understand such a triple as a partial correctness assertion (PCA)

- The language of assertions should contain BExp

- The language of assertions should contain BExp
- It is natural to extend AExp with integer or logical variables LogVar like i two slides ago

- The language of assertions should contain BExp
- It is natural to extend AExp with integer or logical variables LogVar like i two slides ago
- Furthermore, assertions should be able to quantify over those variables

- The language of assertions should contain BExp
- It is natural to extend AExp with integer or logical variables LogVar like i two slides ago
- Furthermore, assertions should be able to quantify over those variables
- Actually, we allow ourselves more than that: infinite conjunctions ...!

- The class of extended arithmetical expressions EAExp

  $a_1, a_2 ::= i \mid \texttt{z} \mid \texttt{X} \mid a_1 \texttt{ + } a_2 \mid a_1 \texttt{ - } a_2 \mid a_1 \texttt{ * } a_2$

  where $i \in$ LogVar and the rest as in AExp

- The class of extended arithmetical expressions EAExp

  $a_1, a_2 ::= i \mid \mathtt{Z} \mid \mathtt{X} \mid a_1 \ \mathtt{+} \ a_2 \mid a_1 \ \mathtt{-} \ a_2 \mid a_1 \ \mathtt{*} \ a_2$

  where $i \in$ LogVar and the rest as in AExp
- The class of assertions of Hoare logic AssertHo

  $b_1, b_2 ::= \top \mid a_1 \ \mathtt{==} \ a_2 \mid a_1 \ \mathtt{<} \ a_2 \mid \neg b_1 \mid \bigwedge_{z \in \mathbb{Z}} b_z,$

  where $a_1, a_2 \in$ EAExp.

- The class of extended arithmetical expressions EAExp

  $a_1, a_2 ::= i \mid \texttt{z} \mid \texttt{X} \mid a_1 \texttt{ + } a_2 \mid a_1 \texttt{ - } a_2 \mid a_1 \texttt{ * } a_2$

  where $i \in$ LogVar and the rest as in AExp
- The class of assertions of Hoare logic AssertHo

  $b_1, b_2 ::= \top \mid a_1 \texttt{ == } a_2 \mid a_1 \texttt{ < } a_2 \mid \neg b_1 \mid \bigwedge_{z \in \mathbb{Z}} b_z,$

  where $a_1, a_2 \in$ EAExp.
- Note 1: lots of things definable

  $\texttt{false}, \quad b_1 \texttt{ and } b_2, \quad b_1 \texttt{ or } b_2, \quad b_1 \to b_2,$

  $a_1 \texttt{ <= } a_2, \quad a_1 \texttt{ >= } a_2,$

  $\forall i.b(i), \quad \exists i.b(i), \quad \bigwedge_{n \in \mathbb{N}} b_n, \quad \bigvee_{n \in \mathbb{N}} b_n \ldots$

  Re quantification, it is so-called substitutional interpretation

- The class of extended arithmetical expressions EAExp

  $a_1, a_2 ::= i \mid \texttt{z} \mid \texttt{X} \mid a_1 \texttt{ + } a_2 \mid a_1 \texttt{ - } a_2 \mid a_1 \texttt{ * } a_2$

  where $i \in$ LogVar and the rest as in AExp
- The class of assertions of Hoare logic AssertHo

  $b_1, b_2 ::= \top \mid a_1 \texttt{ == } a_2 \mid a_1 \texttt{ < } a_2 \mid \neg b_1 \mid \bigwedge_{z \in \mathbb{Z}} b_z,$

  where $a_1, a_2 \in$ EAExp.
- Note 1: lots of things definable

  $\texttt{false}, \quad b_1 \texttt{ and } b_2, \quad b_1 \texttt{ or } b_2, \quad b_1 \to b_2,$

  $a_1 \texttt{ <= } a_2, \quad a_1 \texttt{ >= } a_2,$

  $\forall i.b(i), \quad \exists i.b(i), \quad \bigwedge_{n \in \mathbb{N}} b_n, \quad \bigvee_{n \in \mathbb{N}} b_n \ldots$

  Re quantification, it is so-called substitutional interpretation
- Note 2: use of notation interchangeable, esp. in AssertHo ($\top$ and `true`, $\bot$ and `false`, $\wedge$ and `and`, $\vee$ and `or` ...)

Why those infinitary connectives?

1. Much less bureaucracy in relative completeness proof

   otw, tricks with Gödel numbering etc. needed

Why those infinitary connectives?

1. Much less bureaucracy in relative completeness proof

   otw, tricks with Gödel numbering etc. needed

2. Easy to encode in dependent, higher-order proof assistants

   Just for those suprema/infima indexed by functions the proof assistant "believes to exist", of course

Why those infinitary connectives?

1. Much less bureaucracy in relative completeness proof

   otw, tricks with Gödel numbering etc. needed

2. Easy to encode in dependent, higher-order proof assistants

   Just for those suprema/infima indexed by functions the proof assistant "believes to exist", of course

3. equalities like $(t_1!)$ `==` t_2 directly expressible:

   $$\bigvee_{n \in \mathbb{N}} (\ (\texttt{n == } t_1\texttt{) and (1 * 2 * ... * n == } t_2\texttt{)})$$

- Returning to the word *compositionality* ...

- Returning to the word *compositionality* ...
- We want to derive

$$\vdash \{A_1\}c\{A_2\}$$

for a more complex c from

$$\vdash \{B_1^1\}c_1\{B_2^1\}, \ldots, \vdash \{B_1^n\}c_n\{B_2^n\},$$

where c is built inductively from c_1, ..., c_n

- Returning to the word *compositionality* ...
- We want to derive

$$\vdash \{A_1\}c\{A_2\}$$

  for a more complex c from

$$\vdash \{B_1^1\}c_1\{B_2^1\}, \ldots, \vdash \{B_1^n\}c_n\{B_2^n\},$$

  where c is built inductively from c_1, ..., c_n
- In other words, want to set up a rule system ...

- Returning to the word *compositionality* ...
- We want to derive

$$\vdash \{A_1\}c\{A_2\}$$

  for a more complex c from

$$\vdash \{B_1^1\}c_1\{B_2^1\}, \ldots, \vdash \{B_1^n\}c_n\{B_2^n\},$$

  where c is built inductively from c_1, ..., c_n
- In other words, want to set up a rule system ...
- Easy example: $$\frac{\vdash \{A_1\}C_1\{A_3\} \qquad \vdash \{A_3\}C_2\{A_2\}}{\vdash \{A_1\}C_1 \,;\, C_2\{A_2\}}$$

- An easy axiom: $\vdash \{A\}$ SKIP $\{A\}$

- An easy axiom: $\vdash \{A\}\ \texttt{SKIP}\ \{A\}$
- Clearly sound, clearly no scope for improvement

- An easy axiom: $\vdash \{A\}$ SKIP $\{A\}$
- Clearly sound, clearly no scope for improvement
- But how about assignment ...?

- An easy axiom: $\vdash \{A\}$ `SKIP` $\{A\}$
- Clearly sound, clearly no scope for improvement
- But how about assignment ...?
- First take: $\{\top\}$ `X := a{ X ==` $a\}$?

- An easy axiom: $\vdash \{A\}$ `SKIP` $\{A\}$
- Clearly sound, clearly no scope for improvement
- But how about assignment ...?
- First take: $\{\top\}$ `X := a{ X == a}` ?
- What if we take a to be `X + 1` ?

- An easy axiom: $\vdash \{A\}$ `SKIP` $\{A\}$
- Clearly sound, clearly no scope for improvement
- But how about assignment ...?
- First take: $\{\top\}$ `X := a{ X == a}` ?
- What if we take a to be `X + 1` ?
- Seems that $\not\vdash \{\top\}$ `X := X + 1 { X == X + 1 }`

  To be completely precise, we'd need formal semantics

- An easy axiom: $\vdash \{A\}$ `SKIP` $\{A\}$
- Clearly sound, clearly no scope for improvement
- But how about assignment …?
- First take: $\{\top\}$ `X := a{ X ==  a}` ?
- What if we take a to be `X + 1` ?
- Seems that $\not\vdash \{\top\}$ `X := X + 1 { X == X + 1 }`

  To be completely precise, we'd need formal semantics
- Second take: $\{A\}$ `X := a{`A`[a/ X ]}` ?

  A`[a/ X ]` is read as *A with a replacing* `X`

- An easy axiom: $\vdash \{A\}$ `SKIP` $\{A\}$
- Clearly sound, clearly no scope for improvement
- But how about assignment ...?
- First take: $\{\top\}$ `X := a{ X == a}` ?
- What if we take a to be `X + 1` ?
- Seems that $\not\vdash \{\top\}$ `X := X + 1 { X == X + 1 }`

  To be completely precise, we'd need formal semantics
- Second take: $\{A\}$ `X := a{`A`[a/ X ]}` ?

  A`[a/ X ]` is read as *A with a replacing* `X`
- What if we take A to be `Y == X` and a to be `X + 1` again?

- An easy axiom: $\vdash \{A\}$ `SKIP` $\{A\}$
- Clearly sound, clearly no scope for improvement
- But how about assignment ...?
- First take: $\{\top\}$ `X := a{ X == a}` ?
- What if we take a to be `X + 1` ?
- Seems that $\not\vdash \{\top\}$ `X := X + 1 { X == X + 1 }`

  To be completely precise, we'd need formal semantics
- Second take: $\{A\}$ `X := a{`A`[a/ X ]}` ?

  A`[a/ X ]` is read as *A with a replacing* `X`
- What if we take A to be `Y == X` and a to be `X + 1` again?
- Seems that $\not\vdash$ `{ Y == X } X := X + 1 { Y == X + 1 }`

  Again, to be completely precise we'd need formal semantics

- Maybe the "backwards" one: $\{A[a/\,\mathsf{X}\,]\}\,\mathsf{X}\;\;\mathsf{:=}\;a\{A\}$?

- Maybe the "backwards" one: $\{A[a/\,\mathsf{X}\,]\}\,\mathsf{X} \;:=\; a\{A\}$?
- Actually, this is the right idea!

  ...but after those false takes we better have a semantic proof it holds!

- Maybe the "backwards" one: $\{A[a/\,\mathsf{X}\,]\}\,\mathsf{X}\ \mathsf{:=}\ a\{A\}$?
- Actually, this is the right idea!

  ...but after those false takes we better have a semantic proof it holds!
- Moreover, how do we know it's "the most general possible" axiom?

- Maybe the "backwards" one: $\{A[a/\ \mathtt{X}\ ]\}\ \mathtt{X}\ \mathtt{:=}\ a\{A\}$?
- Actually, this is the right idea!

  ...but after those false takes we better have a semantic proof it holds!
- Moreover, how do we know it's "the most general possible" axiom?
- For example, we could also propose:

  ~~$\{A\}\ \mathtt{X}\ \mathtt{:=}\ a\{\bigvee_{z\in\mathbb{Z}}(A[\mathtt{z}/\ \mathtt{X}\ ]\ \mathtt{and}\ \mathtt{z}\ \mathtt{==}\ a)\}$~~

- Maybe the "backwards" one: $\{A[a/\,\mathtt{X}\,]\}\ \mathtt{X}\ \mathtt{:=}\ a\{A\}$?
- Actually, this is the right idea!

  ...but after those false takes we better have a semantic proof it holds!
- Moreover, how do we know it's "the most general possible" axiom?
- For example, we could also propose:

  ~~$\{A\}\ \mathtt{X}\ \mathtt{:=}\ a\{\bigvee_{z\in\mathbb{Z}}(A[\mathtt{z}/\,\mathtt{X}\,]\ \mathtt{and}\ \mathtt{z}\ \mathtt{==}\ a)\}$~~
- If you have a concrete proof system and a concrete candidate for a replacement axiom, you can always try to to derive this candidate from previously postulated ones

- Maybe the "backwards" one: $\{A[a/\,\mathtt{X}\,]\}\ \mathtt{X}\ \mathtt{:=}\ a\{A\}$?
- Actually, this is the right idea!

  ...but after those false takes we better have a semantic proof it holds!
- Moreover, how do we know it's "the most general possible" axiom?
- For example, we could also propose:

  ~~$\{A\}\ \mathtt{X}\ \mathtt{:=}\ a\{\bigvee_{z\in\mathbb{Z}}(A[\mathtt{z}/\,\mathtt{X}\,]\ \mathtt{and}\ \mathtt{z}\ \mathtt{==}\ a)\}$~~
- If you have a concrete proof system and a concrete candidate for a replacement axiom, you can always try to to derive this candidate from previously postulated ones
- But in general, it is a question of completeness (perhaps relatively to some *oracle*)

- Maybe the "backwards" one: $\{A[a/\,\mathtt{X}\,]\}\ \mathtt{X}\ \mathtt{:=}\ a\{A\}$?
- Actually, this is the right idea!

  ...but after those false takes we better have a semantic proof it holds!
- Moreover, how do we know it's "the most general possible" axiom?
- For example, we could also propose:
  ~~$\{A\}\ \mathtt{X}\ \mathtt{:=}\ a\{\bigvee_{z\in\mathbb{Z}}(A[\mathtt{z}/\,\mathtt{X}\,]\ \mathtt{and}\ \mathtt{z}\ \mathtt{==}\ a)\}$~~
- If you have a concrete proof system and a concrete candidate for a replacement axiom, you can always try to to derive this candidate from previously postulated ones
- But in general, it is a question of completeness (perhaps relatively to some *oracle*)
- Such questions can be settled only against specific semantics

- For conditionals, we could think of a trivial rule:

- For conditionals, we could think of a trivial rule:
- whenever $\{A_1\}c_1\{A_2\}$ and $\{A_1\}c_2\{A_2\}$ hold, it's true that $\{A_1\}$`IF` b `THEN` c_1 `ELSE` c_2 `ENDIF`$\{A_2\}$ holds too

- For conditionals, we could think of a trivial rule:
- whenever $\{A_1\}c_1\{A_2\}$ and $\{A_1\}c_2\{A_2\}$ hold, it's true that $\{A_1\}$`IF` b `THEN` c_1 `ELSE` c_2 `ENDIF`$\{A_2\}$ holds too
- However, it seems too restrictive not to mention b!

- For conditionals, we could think of a trivial rule:
- whenever $\{A_1\}c_1\{A_2\}$ and $\{A_1\}c_2\{A_2\}$ hold, it's true that $\{A_1\}\texttt{IF}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{ENDIF}\{A_2\}$ holds too
- However, it seems too restrictive not to mention b!
- So we propose instead

$$\frac{\vdash \{A_1 \wedge b\}c_1\{A_2\} \qquad \vdash \{A_1 \wedge \neg b\}c_2\{A_2\}}{\vdash \{A_1\}\ \texttt{IF}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{ENDIF}\ \{A_2\}}$$

Note how we promote here b to an assertion, using the fact that BExp is a subset of AssertHo!

- For conditionals, we could think of a trivial rule:
- whenever $\{A_1\}c_1\{A_2\}$ and $\{A_1\}c_2\{A_2\}$ hold, it's true that $\{A_1\}\texttt{IF}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{ENDIF}\{A_2\}$ holds too
- However, it seems too restrictive not to mention b!
- So we propose instead

$$\frac{\vdash \{A_1 \wedge b\}c_1\{A_2\} \qquad \vdash \{A_1 \wedge \neg b\}c_2\{A_2\}}{\vdash \{A_1\}\ \texttt{IF}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{ENDIF}\ \{A_2\}}$$

  Note how we promote here b to an assertion, using the fact that BExp is a subset of AssertHo!
- We would like to be able to show that we can derive the trivial rule from this

- For conditionals, we could think of a trivial rule:
- whenever $\{A_1\}c_1\{A_2\}$ and $\{A_1\}c_2\{A_2\}$ hold, it's true that $\{A_1\}\texttt{IF}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{ENDIF}\{A_2\}$ holds too
- However, it seems too restrictive not to mention b!
- So we propose instead

$$\frac{\vdash \{A_1 \wedge b\}c_1\{A_2\} \qquad \vdash \{A_1 \wedge \neg b\}c_2\{A_2\}}{\vdash \{A_1\}\ \texttt{IF}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{ENDIF}\ \{A_2\}}$$

  Note how we promote here b to an assertion, using the fact that BExp is a subset of AssertHo!
- We would like to be able to show that we can derive the trivial rule from this
- We need more inference rules

- Similarly, we can find suboptimal rules for `WHILE` ...

- Similarly, we can find suboptimal rules for WHILE ...
- e.g., whenever $\{A\}c\{A\}$ holds, it's true that $\{A\}$WHILE b DO c END$\{A\}$ holds too

- Similarly, we can find suboptimal rules for `WHILE` ...
- e.g., whenever $\{A\}c\{A\}$ holds, it's true that $\{A\}$`WHILE` b `DO` c `END`$\{A\}$ holds too
- However, it *again* seems too restrictive not to mention b!

- Similarly, we can find suboptimal rules for WHILE ...
- e.g., whenever $\{A\}c\{A\}$ holds, it's true that $\{A\}$WHILE b DO c END$\{A\}$ holds too
- However, it *again* seems too restrictive not to mention b!
- So we propose instead $$\frac{\vdash \{A \wedge b\}\, c\, \{A\}}{\vdash \{A\}\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}\ \{A \wedge \neg b\}}$$

  Note again we promote here b to an assertion, using the fact that BExp is a subset of AssertHo

- Similarly, we can find suboptimal rules for WHILE ...
- e.g., whenever $\{A\}c\{A\}$ holds, it's true that $\{A\}$WHILE b DO c END$\{A\}$ holds too
- However, it *again* seems too restrictive not to mention b!
- So we propose instead $$\frac{\vdash \{A \wedge b\}c\{A\}}{\vdash \{A\}\ \mathtt{WHILE}\ b\ \mathtt{DO}\ c\ \mathtt{END}\ \{A \wedge \neg b\}}$$
  Note again we promote here b to an assertion, using the fact that BExp is a subset of AssertHo
- As you know, such an A is called invariant of the loop

- Similarly, we can find suboptimal rules for WHILE ...
- e.g., whenever $\{A\}c\{A\}$ holds, it's true that $\{A\}$WHILE b DO c END$\{A\}$ holds too
- However, it *again* seems too restrictive not to mention b!
- So we propose instead $$\frac{\vdash \{A \wedge b\}c\{A\}}{\vdash \{A\}\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}\ \{A \wedge \neg b\}}$$
  Note again we promote here b to an assertion, using the fact that BExp is a subset of AssertHo
- As you know, such an A is called invariant of the loop
- Same questions as above: Is this rule sound? Is it most general?

- Recall all our problems of relating various possible rules for assignment, conditionals, loops ...

- Recall all our problems of relating various possible rules for assignment, conditionals, loops ...
- We need a (set of) rule(s) allowing to make inferences between PCA's involving the same commands but different pre/post-conditions

- Recall all our problems of relating various possible rules for assignment, conditionals, loops ...
- We need a (set of) rule(s) allowing to make inferences between PCA's involving the same commands but different pre/post-conditions
- Different spirit than those presented so far, which were inductive in c ...

- Assume

- Assume
  - $\{A\}c\{B\}$ holds ...

- Assume
  - $\{A\}c\{B\}$ holds ...
  - A' implies A ...

    what does it mean, by the way?

- Assume
  - $\{A\}c\{B\}$ holds ...
  - A' implies A ...

    what does it mean, by the way?
  - ...and A' holds before the execution

- Assume
  - $\{A\}c\{B\}$ holds ...
  - A' implies A ...

    what does it mean, by the way?
  - ...and A' holds before the execution
- Is it the case that B holds afterwards?

- Assume
  - $\{A\}c\{B\}$ holds ...
  - A' implies A ...

    what does it mean, by the way?
  - ...and A' holds before the execution
- Is it the case that B holds afterwards?
- We can always strengthen the precondition

  The more consequences you can derive, the stronger the precondition is

- Assume
  - $\{A\}c\{B\}$ holds ...
  - A' implies A ...

    what does it mean, by the way?
  - ...and A' holds before the execution
- Is it the case that B holds afterwards?
- We can always strengthen the precondition

  The more consequences you can derive, the stronger the precondition is
- Is the question of finding the strongest precondition interesting?

- Assume
  - $\{A\}c\{B\}$ holds ...
  - A' implies A ...

    what does it mean, by the way?
  - ...and A' holds before the execution
- Is it the case that B holds afterwards?
- We can always strengthen the precondition

  The more consequences you can derive, the stronger the precondition is
- Is the question of finding the strongest precondition interesting?
- $\{\texttt{false}\}c\{B\}$ always holds!

- Assume
  - $\{A\}c\{B\}$ holds ...
  - A' implies A ...

    what does it mean, by the way?
  - ...and A' holds before the execution
- Is it the case that B holds afterwards?
- We can always strengthen the precondition

  The more consequences you can derive, the stronger the precondition is
- Is the question of finding the strongest precondition interesting?
- $\{\texttt{false}\}c\{B\}$ always holds!
- In order to find best possible axiomatic specification ("semantics") for a command, we need to find weakest preconditions (relatively to each postcondition)!

- Assume
  - $\{A\}c\{B\}$ holds ...
  - A' implies A ...

    what does it mean, by the way?
  - ...and A' holds before the execution
- Is it the case that B holds afterwards?
- We can always strengthen the precondition

  The more consequences you can derive, the stronger the precondition is
- Is the question of finding the strongest precondition interesting?
- $\{\texttt{false}\}c\{B\}$ always holds!
- In order to find best possible axiomatic specification ("semantics") for a command, we need to find weakest preconditions (relatively to each postcondition)!
- For now, in order to clarify the notion of consequence and write a formal rule, we need to introduce some semantics into the picture

Semantics of arithmetical and boolean expressions

- Let us return for a second to the observation that elements of EAExp are just terms for signature $\{$ +/2, -/2, */2, z/0, X/0 $\mid z \in \mathbb{Z}$, X $\in$ ProgVar$\}$

  Note that I'm more firmly stating elements of ProgVar are constants

- Let us return for a second to the observation that elements of EAExp are just terms for signature $\{$ +/2, -/2, */2, z/0, X/0 $\mid z \in \mathbb{Z}$, X $\in$ ProgVar$\}$

  Note that I'm more firmly stating elements of ProgVar are constants
- What did GLoIn say we need in order to learn the meaning of such beasts?

- Let us return for a second to the observation that elements of EAExp are just terms for signature $\{$ +/2, -/2, */2, z/0, X/0 $\mid z \in \mathbb{Z}$, X $\in$ ProgVar$\}$

  Note that I'm more firmly stating elements of ProgVar are constants
- What did GLoIn say we need in order to learn the meaning of such beasts?
- If you recall, one had to provide first a model interpreting constants and function symbols ...

- Let us return for a second to the observation that elements of EAExp are just terms for signature $\{$ +/2, -/2, */2, z/0, X/0 $\mid z \in \mathbb{Z}$, X $\in$ ProgVar$\}$

  Note that I'm more firmly stating elements of ProgVar are constants
- What did GLoIn say we need in order to learn the meaning of such beasts?
- If you recall, one had to provide first a model interpreting constants and function symbols ...
- ...but in our case we are not interested in the whole variety of possible interpretations ...

- Let us return for a second to the observation that elements of EAExp are just terms for signature $\{$ +/2, -/2, */2, z/0, X/0 $\mid z \in \mathbb{Z}$, X $\in$ ProgVar$\}$

  Note that I'm more firmly stating elements of ProgVar are constants
- What did GLoIn say we need in order to learn the meaning of such beasts?
- If you recall, one had to provide first a model interpreting constants and function symbols ...
- ...but in our case we are not interested in the whole variety of possible interpretations ...
- ...we care about a single, fixed, intended and concrete model: integer numbers $\mathbb{Z}$ where + gets interpreted as integer addition, * as integer multiplication etc.

- Let us return for a second to the observation that elements of EAExp are just terms for signature $\{$ +/2, -/2, */2, z/0, X/0 $\mid z \in \mathbb{Z}$, X $\in$ ProgVar$\}$

  Note that I'm more firmly stating elements of ProgVar are constants
- What did GLoIn say we need in order to learn the meaning of such beasts?
- If you recall, one had to provide first a model interpreting constants and function symbols ...
- ...but in our case we are not interested in the whole variety of possible interpretations ...
- ...we care about a single, fixed, intended and concrete model: integer numbers $\mathbb{Z}$ where + gets interpreted as integer addition, * as integer multiplication etc.
- So this fixed model does not even need to be mentioned in our notation ...

- Let us return for a second to the observation that elements of EAExp are just terms for signature $\{$ +/2, -/2, */2, z/0, X/0 $\mid z \in \mathbb{Z}$, X $\in$ ProgVar$\}$

  Note that I'm more firmly stating elements of ProgVar are constants
- What did GLoIn say we need in order to learn the meaning of such beasts?
- If you recall, one had to provide first a model interpreting constants and function symbols ...
- ...but in our case we are not interested in the whole variety of possible interpretations ...
- ...we care about a single, fixed, intended and concrete model: integer numbers $\mathbb{Z}$ where + gets interpreted as integer addition, * as integer multiplication etc.
- So this fixed model does not even need to be mentioned in our notation ...
- We still have elements of ProgVar to take care of though!

- We can replace models with functions assigning integers to elements of ProgVar

- We can replace models with functions assigning integers to elements of ProgVar
- But there is a perfectly computational interpretations for these entities now. They are states!

Definition of state

- *In any computation, the present set of values being operated upon*

  Dictionary of Computer Science, Engineering and Technology, edited by Philip A. Laplante

Definition of state

- *In any computation, the present set of values being operated upon*

  Dictionary of Computer Science, Engineering and Technology, edited by Philip A. Laplante

- While we are at defining things, IMP is a minimalistic imperative language. What does this mean?

Definition of state

- *In any computation, the present set of values being operated upon*

  Dictionary of Computer Science, Engineering and Technology, edited by Philip A. Laplante
- While we are at defining things, IMP is a minimalistic imperative language. What does this mean?
- In the same source, we read this is a *programming language which achieves its primary effect by changing the state of variables by assignment*

Definition of state

- *In any computation, the present set of values being operated upon*

  Dictionary of Computer Science, Engineering and Technology, edited by Philip A. Laplante
- While we are at defining things, IMP is a minimalistic imperative language. What does this mean?
- In the same source, we read this is a *programming language which achieves its primary effect by changing the state of variables by assignment*
- It is thus no wonder states figure prominently in the denotational semantics of IMP

Definition of state

- *In any computation, the present set of values being operated upon*

  Dictionary of Computer Science, Engineering and Technology, edited by Philip A. Laplante
- While we are at defining things, IMP is a minimalistic imperative language. What does this mean?
- In the same source, we read this is a *programming language which achieves its primary effect by changing the state of variables by assignment*
- It is thus no wonder states figure prominently in the denotational semantics of IMP
- Those of you who attended SemProg will recall that also the operational semantics of IMPuses states in an essential way

Shared mutable data structures

- Before we go on further, ponder one thing ...

Shared mutable data structures

- Before we go on further, ponder one thing ...
- ... how much bigger the states would become if we allowed pointers, heaps, direct access to memory locations, shared mutable data structures?

  John Reynolds: *structures where an updatable field can be referenced from more than one point*

Shared mutable data structures

- Before we go on further, ponder one thing ...
- ... how much bigger the states would become if we allowed pointers, heaps, direct access to memory locations, shared mutable data structures?

  John Reynolds: *structures where an updatable field can be referenced from more than one point*
- Basically, each state would have to contain a snapshot of the whole heap, not just the program variables

Shared mutable data structures

- Before we go on further, ponder one thing ...
- ... how much bigger the states would become if we allowed pointers, heaps, direct access to memory locations, shared mutable data structures?

  John Reynolds: *structures where an updatable field can be referenced from more than one point*
- Basically, each state would have to contain a snapshot of the whole heap, not just the program variables
- Reasoning becomes difficult both theoretically and practically

Shared mutable data structures

- Before we go on further, ponder one thing ...
- ... how much bigger the states would become if we allowed pointers, heaps, direct access to memory locations, shared mutable data structures?

  John Reynolds: *structures where an updatable field can be referenced from more than one point*
- Basically, each state would have to contain a snapshot of the whole heap, not just the program variables
- Reasoning becomes difficult both theoretically and practically
- It does not mean *impossible* though. We'll see it later

States

- For the time being, back to our simple states

States

- For the time being, back to our simple states
- In fact, these simplest states should be called stores

States

- For the time being, back to our simple states
- In fact, these simplest states should be called stores
- Notation: $\mathsf{States}_{\mathsf{IMP}}$ is the set of all states σ, i.e., $\mathsf{States}_{\mathsf{IMP}} := \{\sigma \mid \sigma : \mathsf{ProgVar} \to \mathbb{Z}\}$

States

- For the time being, back to our simple states
- In fact, these simplest states should be called stores
- Notation: $\mathsf{States}_{\mathsf{IMP}}$ is the set of all states σ, i.e., $\mathsf{States}_{\mathsf{IMP}} := \{\sigma \mid \sigma : \mathsf{ProgVar} \to \mathbb{Z}\}$
- As you probably know, mathematicians would simply write $\mathsf{States}_{\mathsf{IMP}} = \mathsf{ProgVar} \to \mathbb{Z}$ or $\mathsf{States}_{\mathsf{IMP}} = \mathbb{Z}^{\mathsf{ProgVar}}$

States

- For the time being, back to our simple states
- In fact, these simplest states should be called stores
- Notation: $\mathsf{States}_{\mathsf{IMP}}$ is the set of all states σ, i.e., $\mathsf{States}_{\mathsf{IMP}} := \{\sigma \mid \sigma : \mathsf{ProgVar} \to \mathbb{Z}\}$
- As you probably know, mathematicians would simply write $\mathsf{States}_{\mathsf{IMP}} = \mathsf{ProgVar} \to \mathbb{Z}$ or $\mathsf{States}_{\mathsf{IMP}} = \mathbb{Z}^{\mathsf{ProgVar}}$
- In blackboard proofs, I will often write simply Σ or Σ_{IMP}

- Next, we need to extend our semantics to EAExp

- Next, we need to extend our semantics to EAExp
- These involve additional type of variables LogVar, so semantics has to involve interpretations (or interpretation functions) $I : \mathsf{LogVar} \to \mathbb{Z}$

- Next, we need to extend our semantics to EAExp
- These involve additional type of variables LogVar, so semantics has to involve interpretations (or interpretation functions) $I : \mathsf{LogVar} \to \mathbb{Z}$
- These in turn can be identified with $\hat{I} : \mathsf{LogVar} \to \mathsf{AExp}$: if $I(i) = z$, then $\hat{I}(i) := \mathsf{z}$

  We rely on having enough constants to name all the elements

- Next, we need to extend our semantics to EAExp
- These involve additional type of variables LogVar, so semantics has to involve interpretations (or interpretation functions) $I : \mathsf{LogVar} \to \mathbb{Z}$
- These in turn can be identified with $\hat{I} : \mathsf{LogVar} \to \mathsf{AExp}$: if $I(i) = z$, then $\hat{I}(i) := \mathsf{z}$

  We rely on having enough constants to name all the elements
- We can extend $\hat{I}$ to a map $\mathsf{EAExp} \to \mathsf{AExp}$: $\hat{I}(a) := a[\hat{I}(i)/i]$

- Next, we need to extend our semantics to EAExp
- These involve additional type of variables LogVar, so semantics has to involve interpretations (or interpretation functions) $I : \mathsf{LogVar} \to \mathbb{Z}$
- These in turn can be identified with $\hat{I} : \mathsf{LogVar} \to \mathsf{AExp}$: if $I(i) = z$, then $\hat{I}(i) := \mathsf{z}$

  We rely on having enough constants to name all the elements
- We can extend $\hat{I}$ to a map $\mathsf{EAExp} \to \mathsf{AExp}$: $\hat{I}(a) := a[\hat{I}(i)/i]$
- And now we can officially cannibalize semantics for AExp: $[\![\mathsf{a}]\!]\sigma I := [\![\hat{I}(\mathsf{a})]\!]\sigma$

- Given $\mathsf{b} \in \mathsf{BExp}$ and $\sigma \in \mathsf{States}_{\mathsf{IMP}}$, the denotation $[\![\ \mathsf{b}\ ]\!]\sigma$ is an element of the obvious object of truth values $2 := \{\mathsf{T}, \bot\}$

- Given $\mathsf{b} \in \mathsf{BExp}$ and $\sigma \in \mathsf{States}_{\mathsf{IMP}}$, the denotation $[\![\ \mathsf{b}\ ]\!]\sigma$ is an element of the obvious object of truth values $2 := \{\mathsf{T}, \bot\}$
- Similarly, $[\![B]\!]\sigma I \in 2$. Denote the obvious boolean operations on 2 as $\wedge$, $\vee$, $\neg$ ...

- Given $\mathsf{b} \in \mathsf{BExp}$ and $\sigma \in \mathsf{States}_{\mathsf{IMP}}$, the denotation $[\![\ \mathsf{b}\ ]\!]\sigma$ is an element of the obvious object of truth values $2 := \{\top, \bot\}$
- Similarly, $[\![B]\!]\sigma I \in 2$. Denote the obvious boolean operations on 2 as $\wedge$, $\vee$, $\neg$ ...
- $[\![\ \texttt{true}\ ]\!]\sigma I := \top$

- Given $\mathsf{b} \in \mathsf{BExp}$ and $\sigma \in \mathsf{States}_{\mathsf{IMP}}$, the denotation $[\![\ \mathsf{b}\ ]\!]\sigma$ is an element of the obvious object of truth values $2 := \{\top, \bot\}$
- Similarly, $[\![B]\!]\sigma I \in 2$. Denote the obvious boolean operations on 2 as $\wedge$, $\vee$, $\neg$ ...
- $[\![\ \texttt{true}\ ]\!]\sigma I := \top$
- $[\![a_1 \texttt{ == } a_2]\!]\sigma I :=$

- Given $\mathsf{b} \in \mathsf{BExp}$ and $\sigma \in \mathsf{States}_{\mathsf{IMP}}$, the denotation $[\![\ \mathsf{b}\ ]\!]\sigma$ is an element of the obvious object of truth values $2 := \{\top, \bot\}$
- Similarly, $[\![B]\!]\sigma I \in 2$. Denote the obvious boolean operations on 2 as $\wedge$, $\vee$, $\neg$ ...
- $[\![\ \texttt{true}\ ]\!]\sigma I := \top$
- $[\![a_1 \texttt{ == } a_2]\!]\sigma I :=$

- Given $\mathsf{b} \in \mathsf{BExp}$ and $\sigma \in \mathsf{States}_{\mathsf{IMP}}$, the denotation $[\![\ \mathsf{b}\ ]\!]\sigma$ is an element of the obvious object of truth values $2 := \{\top, \bot\}$
- Similarly, $[\![B]\!]\sigma I \in 2$. Denote the obvious boolean operations on 2 as $\wedge$, $\vee$, $\neg$ ...
- $[\![\ \texttt{true}\ ]\!]\sigma I := \top$
- $$[\![a_1 \texttt{ == } a_2]\!]\sigma I := \begin{cases} \top & \text{if } [\![a_1]\!]\sigma I = [\![a_2]\!]\sigma I \\ \bot & \text{otherwise} \end{cases}$$

- Given $\mathsf{b} \in \mathsf{BExp}$ and $\sigma \in \mathsf{States}_{\mathsf{IMP}}$, the denotation $[\![\ \mathsf{b}\ ]\!]\sigma$ is an element of the obvious object of truth values $2 := \{\top, \bot\}$
- Similarly, $[\![B]\!]\sigma I \in 2$. Denote the obvious boolean operations on 2 as $\wedge$, $\vee$, $\neg$ ...
- $[\![\ \texttt{true}\ ]\!]\sigma I := \top$
- $$[\![a_1 \texttt{ == } a_2]\!]\sigma I := \begin{cases} \top & \text{if } [\![a_1]\!]\sigma I = [\![a_2]\!]\sigma I \\ \bot & \text{otherwise} \end{cases}$$
- $[\![a_1 \texttt{ < } a_2]\!]\sigma I :=$

- Given $\mathsf{b} \in \mathsf{BExp}$ and $\sigma \in \mathsf{States}_{\mathsf{IMP}}$, the denotation $[\![\ \mathsf{b}\ ]\!]\sigma$ is an element of the obvious object of truth values $2 := \{\top, \bot\}$
- Similarly, $[\![B]\!]\sigma I \in 2$. Denote the obvious boolean operations on 2 as $\wedge$, $\vee$, $\neg$ ...
- $[\![\ \texttt{true}\ ]\!]\sigma I := \top$
- $$[\![a_1 \texttt{ == } a_2]\!]\sigma I := \begin{cases} \top & \text{if } [\![a_1]\!]\sigma I = [\![a_2]\!]\sigma I \\ \bot & \text{otherwise} \end{cases}$$
- $[\![a_1 \texttt{ < } a_2]\!]\sigma I :=$

- Given $\mathsf{b} \in \mathsf{BExp}$ and $\sigma \in \mathsf{States}_{\mathsf{IMP}}$, the denotation $[\![\ \mathsf{b}\ ]\!]\sigma$ is an element of the obvious object of truth values $2 := \{\top, \bot\}$
- Similarly, $[\![B]\!]\sigma I \in 2$. Denote the obvious boolean operations on 2 as $\wedge$, $\vee$, $\neg$ ...
- $[\![\ \texttt{true}\ ]\!]\sigma I := \top$
- $[\![a_1 \texttt{ == } a_2]\!]\sigma I := \begin{cases} \top & \text{if } [\![a_1]\!]\sigma I = [\![a_2]\!]\sigma I \\ \bot & \text{otherwise} \end{cases}$
- $[\![a_1 \texttt{ < } a_2]\!]\sigma I := \begin{cases} \top & \text{if } [\![a_1]\!]\sigma I < [\![a_2]\!]\sigma I \\ \bot & \text{otherwise} \end{cases}$
- $[\![\ \texttt{not } B]\!]\sigma I := \neg [\![B]\!]\sigma I$

- Given $\mathsf{b} \in \mathsf{BExp}$ and $\sigma \in \mathsf{States}_{\mathsf{IMP}}$, the denotation $[\![\ \mathsf{b}\ ]\!]\sigma$ is an element of the obvious object of truth values $2 := \{\top, \bot\}$
- Similarly, $[\![B]\!]\sigma I \in 2$. Denote the obvious boolean operations on 2 as $\wedge$, $\vee$, $\neg$ ...
- $[\![\ \texttt{true}\ ]\!]\sigma I := \top$
- $[\![a_1 \texttt{ == } a_2]\!]\sigma I := \begin{cases} \top & \text{if } [\![a_1]\!]\sigma I = [\![a_2]\!]\sigma I \\ \bot & \text{otherwise} \end{cases}$
- $[\![a_1 \texttt{ < } a_2]\!]\sigma I := \begin{cases} \top & \text{if } [\![a_1]\!]\sigma I < [\![a_2]\!]\sigma I \\ \bot & \text{otherwise} \end{cases}$
- $[\![\ \texttt{not}\ B]\!]\sigma I := \neg [\![B]\!]\sigma I$
- $[\![\bigwedge_{z \in \mathbb{Z}} B_z]\!]\sigma I = \begin{cases} \top & \text{if } [\![B_z]\!]\sigma I = \top \text{ for all } z \\ \bot & \text{otherwise} \end{cases}$

- Let us introduce $\sigma \models^I B$ as another notation for $[\![B]\!]\sigma I = \top$

  Exercise: rewrite the definition of semantics in this notation

- Let us introduce $\sigma \vDash^I B$ as another notation for $[\![B]\!]\sigma I = \top$

  Exercise: rewrite the definition of semantics in this notation

- Exercise: Show that $[\![B[\mathsf{z}/i]]\!]\sigma I = [\![B]\!]\sigma I[z/i]$ and use this to show that forcing for defined quantifiers agrees with the standard definition

- Let us introduce $\sigma \vDash^I B$ as another notation for $[\![B]\!]\sigma I = \top$

  Exercise: rewrite the definition of semantics in this notation
- Exercise: Show that $[\![B[\mathsf{z}/i]]\!]\sigma I = [\![B]\!]\sigma I[z/i]$ and use this to show that forcing for defined quantifiers agrees with the standard definition
- Furthermore, write $\vDash B$ for "for all σ and I, $\sigma \vDash^I B$"

Consequence rule: finally formally

- Recall we could always strengthen the precondition

Consequence rule: finally formally

- Recall we could always strengthen the precondition
- Dually, we can always weaken the postcondition

Consequence rule: finally formally

- Recall we could always strengthen the precondition
- Dually, we can always weaken the postcondition
- We can formalize it as $$\frac{\vDash A' \to A, \quad \vdash \{A\}c\{B\}, \quad \vDash B \to B'}{\vdash \{A'\}c\{B'\}}$$

Consequence rule: finally formally

- Recall we could always strengthen the precondition
- Dually, we can always weaken the postcondition
- We can formalize it as $$\frac{\vDash A' \to A, \quad \vdash \{A\}c\{B\}, \quad \vDash B \to B'}{\vdash \{A'\}c\{B'\}}$$
- Proof of soundness: trivial.

Consequence rule: finally formally

- Recall we could always strengthen the precondition
- Dually, we can always weaken the postcondition
- We can formalize it as $$\frac{\vDash A' \to A, \quad \vdash \{A\}c\{B\}, \quad \vDash B \to B'}{\vdash \{A'\}c\{B'\}}$$
- Proof of soundness: trivial.
- But the rule has semantic judgements hardwired into it!

Consequence rule: finally formally

- Recall we could always strengthen the precondition
- Dually, we can always weaken the postcondition
- We can formalize it as $$\frac{\vDash A' \to A, \quad \vdash \{A\}c\{B\}, \quad \vDash B \to B'}{\vdash \{A'\}c\{B'\}}$$
- Proof of soundness: trivial.
- But the rule has semantic judgements hardwired into it!
- We will later discuss the difficulties it leads to

$$\vdash \{A\}\ \texttt{SKIP}\ \{A\}$$

$$\vdash \{B[a/\ \texttt{X}\ ]\}\ \texttt{X}\ \texttt{:=}\ a\{B\}$$

$$\frac{\vdash \{A_1\}c_1\{A_2\} \qquad \vdash \{A_2\}c_2\{A_3\}}{\vdash \{A_1\}c_1\ \texttt{;}\ c_2\{A_3\}}$$

$$\frac{\vdash \{A_1 \wedge b\}c_1\{A_2\} \qquad \vdash \{A_1 \wedge \neg b\}c_2\{A_2\}}{\vdash \{A_1\}\ \texttt{IF}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{ENDIF}\ \{A_2\}}$$

$$\frac{\vdash \{A \wedge b\}c\{A\}}{\vdash \{A\}\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}\ \{A \wedge \neg b\}}$$

$$\frac{\vDash A' \to A, \quad \vdash \{A\}c\{B\}, \quad \vDash B \to B'}{\vdash \{A'\}c\{B'\}}$$

$$\vdash \{A\}\ \texttt{SKIP}\ \{A\}$$

$$\vdash \{B[a/\ \texttt{X}\ ]\}\ \texttt{X}\ \texttt{:=}\ a\{B\}$$

$$\frac{\vdash \{A_1\}c_1\{A_2\} \qquad \vdash \{A_2\}c_2\{A_3\}}{\vdash \{A_1\}c_1\ \texttt{;}\ c_2\{A_3\}}$$

$$\frac{\vdash \{A_1 \wedge b\}c_1\{A_2\} \qquad \vdash \{A_1 \wedge \neg b\}c_2\{A_2\}}{\vdash \{A_1\}\ \texttt{IF}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{ENDIF}\ \{A_2\}}$$

$$\frac{\vdash \{A \wedge b\}c\{A\}}{\vdash \{A\}\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}\ \{A \wedge \neg b\}}$$

$$\frac{\vDash A' \to A, \quad \vdash \{A\}c\{B\}, \quad \vDash B \to B'}{\vdash \{A'\}c\{B'\}}$$

$$\vdash \{A\}\ \texttt{SKIP}\ \{A\}$$

$$\vdash \{B[a/\ \texttt{X}\ ]\}\ \texttt{X}\ \texttt{:=}\ a\{B\}$$

$$\frac{\vdash \{A_1\}c_1\{A_2\} \qquad \vdash \{A_2\}c_2\{A_3\}}{\vdash \{A_1\}c_1\ \texttt{;}\ c_2\{A_3\}}$$

$$\frac{\vdash \{A_1 \wedge b\}c_1\{A_2\} \qquad \vdash \{A_1 \wedge \neg b\}c_2\{A_2\}}{\vdash \{A_1\}\ \texttt{IF}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{ENDIF}\ \{A_2\}}$$

$$\frac{\vdash \{A \wedge b\}c\{A\}}{\vdash \{A\}\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}\ \{A \wedge \neg b\}}$$

$$\frac{\vDash A' \to A, \quad \vdash \{A\}c\{B\}, \quad \vDash B \to B'}{\vdash \{A'\}c\{B'\}}$$

$$\vdash \{A\}\ \texttt{SKIP}\ \{A\}$$

$$\vdash \{B[a/\ \texttt{X}\ ]\}\ \texttt{X}\ \texttt{:=}\ a\{B\}$$

$$\frac{\vdash \{A_1\}c_1\{A_2\} \qquad \vdash \{A_2\}c_2\{A_3\}}{\vdash \{A_1\}c_1\ \texttt{;}\ c_2\{A_3\}}$$

$$\frac{\vdash \{A_1 \wedge b\}c_1\{A_2\} \qquad \vdash \{A_1 \wedge \neg b\}c_2\{A_2\}}{\vdash \{A_1\}\ \texttt{IF}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{ENDIF}\ \{A_2\}}$$

$$\frac{\vdash \{A \wedge b\}c\{A\}}{\vdash \{A\}\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}\ \{A \wedge \neg b\}}$$

$$\frac{\vDash A' \to A, \quad \vdash \{A\}c\{B\}, \quad \vDash B \to B'}{\vdash \{A'\}c\{B'\}}$$

Let us put them to an example use $\Rightarrow$ annotated programs!

$\{$X == i and i >= 0$\}$

```
Y := 1;
WHILE not (X == 0) DO
     Y := Y * X;
     X := X - 1
END
```

$\{$Y == $i!\}$

Recall that $(t_1!)$ == t_2 encoded in AssertHo as:

$\bigvee_{n \in \mathbb{N}}$((n == t_1) and (1 * 2 * ... * n == t_2))

$\{$X == i and i >= 0$\}$ $\rightarrow$
$\{$X == i and i >= 0 and 1 == 1$\}$

```
Y := 1;
```

$\{$X == i and i >= 0 and Y == 1$\}$

```
WHILE not (X == 0) DO
    Y := Y * X;
    X := X - 1
END
```

$\{$Y == $i!\}$

Recall that $(t_1!)$ == t_2 encoded in AssertHo as:

$\bigvee_{n \in \mathbb{N}}$((n == t_1) and (1 * 2 * ... * n == t_2))

$\{$X == i and i >= 0$\}\rightarrow$
$\{$X == i and i >= 0 and 1 == 1$\}$

```
Y := 1;
```
$\{$X == i and i >= 0 and Y == 1$\}\rightarrow$
$\{$Y * X! == i! and X >= 0 $\}$ $\Leftarrow$only here some creativity needed
```
WHILE not (X == 0) DO
    Y := Y * X;
    X := X - 1
END
```

$\{$Y == i!$\}$

Recall that (t_1!) == t_2 encoded in AssertHo as:

$\bigvee_{n\in\mathbb{N}}$((n == t_1) and (1 * 2 * ... * n == t_2))

```
{X == i and i >= 0} →
{X == i and i >= 0 and 1 == 1}

Y := 1;
{X == i and i >= 0 and Y == 1} →
{Y * X! == i! and X >= 0 }
WHILE not (X == 0) DO
      {Y * X! == i! and X >= 0 and not (X == 0) } →
      { Y * X * (X-1)! == i! and X > 0 }
          ⇑ derived from the encoding of (t1!) == t2 below
    Y := Y * X;
    X := X - 1
END
```

```
{Y == i!}
```

Recall that $(t_1!)$ == t_2 encoded in AssertHo as:

$$\bigvee_{n \in \mathbb{N}} (\ (\texttt{n} == t_1)\ \texttt{and}\ (1 * 2 * \ldots * \texttt{n} == t_2))$$

```
{X == i and i >= 0} →
{X == i and i >= 0 and 1 == 1}

Y := 1;
{X == i and i >= 0 and Y == 1} →
{Y * X! == i! and X >= 0 }
WHILE not (X == 0) DO
      {Y * X! == i! and X >= 0 and not (X == 0) } →
      {Y * X * (X-1)! == i! and X > 0 }
    Y := Y * X;
      {Y * (X-1)! == i! and X > 0 }
    X := X - 1
END




{Y == i!}
```

Recall that $(t_1!)$ == t_2 encoded in AssertHo as:

$$\bigvee_{n \in \mathbb{N}} (\ (\texttt{n} == t_1)\ \texttt{and}\ (\texttt{1} * \texttt{2} * \ldots * \texttt{n} == t_2))$$

```
{X == i and i >= 0} →
{X == i and i >= 0 and 1 == 1}

Y := 1;
{X == i and i >= 0 and Y == 1} →
{Y * X! == i! and X >= 0 }
WHILE not (X == 0) DO
      {Y * X! == i! and X >= 0 and not (X == 0) } →
      {Y * X * (X-1)! == i! and X > 0 }
    Y := Y * X;
      {Y * (X-1)! == i! and X > 0 } →
      {Y * (X − 1)! == i! and X − 1 >= 0 } →
    X := X - 1
      {Y * X! == i! and X >=0 }
END


{Y == i!}
```

Recall that $(t_1!)$ == t_2 encoded in AssertHo as:

$$\bigvee_{n \in \mathbb{N}} (\ (\texttt{n} == t_1)\ \texttt{and}\ (1 * 2 * \ldots * \texttt{n} == t_2))$$

```
{X == i and i >= 0} →
{X == i and i >= 0 and 1 == 1}

Y := 1;
{X == i and i >= 0 and Y == 1} →
{Y * X! == i! and X >= 0 }
WHILE not (X == 0) DO
      {Y * X! == i! and X >= 0 and not (X == 0) } →
      {Y * X * (X-1)! == i! and X > 0 }
    Y := Y * X;
      {Y * (X-1)! == i! and X > 0 } →
      {Y * (X-1)! == i! and X-1 >= 0 } →
    X := X - 1
      {Y * X! == i! and X >=0 }
END
{Y * X! == i! and X >=0 and X == 0 } →

{Y == i!}
```

Recall that $(t_1!)$ == t_2 encoded in AssertHo as:

$$\bigvee_{n \in \mathbb{N}} (\texttt{n} == t_1) \texttt{ and } (1 * 2 * \ldots * \texttt{n} == t_2))$$

- We used rules for assignment, sequencing, and of course consequence and while loops

- We used rules for assignment, sequencing, and of course consequence and while loops
- The applications of consequence rule are very easy to verify to whichever tractable subsystem of arithmetic we pick

  the only possible exception being this infinitary definition of factorial

- We used rules for assignment, sequencing, and of course consequence and while loops
- The applications of consequence rule are very easy to verify to whichever tractable subsystem of arithmetic we pick

  the only possible exception being this infinitary definition of factorial
- We see that we have a sound system which can be used for annotating programs

- We used rules for assignment, sequencing, and of course consequence and while loops
- The applications of consequence rule are very easy to verify to whichever tractable subsystem of arithmetic we pick

  the only possible exception being this infinitary definition of factorial
- We see that we have a sound system which can be used for annotating programs
- The question of (relative) completeness: are our rules as general as possible?

  again, assuming somebody gives us an oracle for arithmetic ...

- We used rules for assignment, sequencing, and of course consequence and while loops
- The applications of consequence rule are very easy to verify to whichever tractable subsystem of arithmetic we pick

  the only possible exception being this infinitary definition of factorial
- We see that we have a sound system which can be used for annotating programs
- The question of (relative) completeness: are our rules as general as possible?

  again, assuming somebody gives us an oracle for arithmetic ...
- In order to discuss it, we would need *denotational semantics*