Deductive Program Verification

Maria João Frade

HASLab - INESC TEC
Departmento de Informática, Universidade do Minho

2018/2019

Roadmap

Deductive Program Verification

- One possible definition: *"an exhaustive, correct and complete form of static checking w.r.t. to a specification, based on a program logic"*
- Provides a global certification that the program behaves as it is specified to behave.
- Properties may include functional aspects; safety properties; security properties; ...
  - formal models using expressive logics
  - computer-assisted mathematical proof
  - requires deep expertise
- Typically confined to very specific application areas where their use and cost are justified.

Program annotations and contracts

- A *program annotation* is a formula placed together with the code of a program indicating the conditions that should be met.
- The rationalisation of the code annotation methodology gave rise to a software development paradigm based on the notion of *contract*.
  - pioneered in the Eiffel programming language (1986), which implements the notion of *runtime or dynamic verification of contracts* (design-by-contract).
- Nowadays every widespread programming language benefits from a contracts layer. Some only support the *static verification of contracts*.
  - Spec#
  - SPARK, ADA 2012
  - Esc/Java, KeY, Krakatoa (based on JML annotation language)
  - Frama-C, VCC (based on ACSL annotation language)
  - ...
- The logical formalisms underlying this approach are program logics like *Hoare logic*.

Hoare Logic

Hoare logic

- Hoare logic (also know as Floyd-Hoare logic) is a method of reasoning mathematically about imperative programs.
  - Robert Floyd, "Assigning meaning to programs", 1967.
  - Tony Hoare, "An axiomatic basis for computer programming", 1969.
- The logic deals with the notion of correction w.r.t. a *specification* that consists of
  - a *precondition* - an assertion that is assumed to hold when the execution of the program starts
  - and a *postcondition* - an assertion that is required to hold when execution stops.

A simple programming language - While$^{\text{int}}$

A While language whose commands are defined over a set of variables $x \in \mathbf{Var}$

$$\mathbf{Type} \ni \tau ::= \mathbf{bool} \mid \mathbf{int}$$

$$\begin{array}{lll} \mathbf{Exp_{int}} \ni e & ::= & \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid x \mid \\ & & -e \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e_1 \,\mathtt{div}\, e_2 \mid e_1 \,\mathtt{mod}\, e_2 \end{array}$$

$$\begin{array}{lll} \mathbf{Exp_{bool}} \ni b & ::= & \mathtt{true} \mid \mathtt{false} \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid e_1 = e_2 \mid e_1 \neq e_2 \mid \\ & & e_1 < e_2 \mid e_1 \leq e_2 \mid e_1 > e_2 \mid e_1 \geq e_2 \end{array}$$

$$\mathbf{Comm} \ni C ::= \mathbf{skip} \mid C \,;\, C \mid x := e \mid \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \mathbf{while}\ b\ \mathbf{do}\ C$$

Assertions about programs

- We need formulas that express properties of particular states of the program.
- Program assertions $\phi, \theta, \psi \in \mathbf{Assert}$ (preconditions and postconditions in particular) are *first-order formulas* of a language obtained as an expansion of $\mathbf{Exp_{bool}}$.
- Note that assertions may contain occurrences of functions and predicates provided by the user.

Semantics

- Will consider an *interpretation structure* $\mathcal{M} = (D, I)$ for the vocabulary describing the concrete syntax of program expressions.
- The interpretation of expressions depends on a *state*, which is a function that maps each variable into its value. $\Sigma = \mathbf{Var} \to D$
- In the While[int] the set of states is $\Sigma = \mathbf{Var} \to \mathbb{Z}$
- Expressions are interpreted as functions from states to the corresponding domain of interpretation.
- We are considering that expressions evaluation
  - are free of side-effects
  - does not go wrong

Semantics of expressions

- $[\![e]\!] : \Sigma \to \mathbb{Z}$ is defined inductively by:

$$
\begin{array}{rcl}
[\![n]\!](s) & = & n \\
[\![x]\!](s) & = & s(x) \\
[\![-e]\!](s) & = & -[\![e]\!](s) \\
[\![e_1 + e_2]\!](s) & = & [\![e_1]\!](s) + [\![e_2]\!](s) \\
[\![e_1 - e_2]\!](s) & = & [\![e_1]\!](s) - [\![e_2]\!](s) \\
[\![e_1 \times e_2]\!](s) & = & [\![e_1]\!](s) \times [\![e_2]\!](s) \\
[\![e_1 \,\texttt{div}\, e_2]\!](s) & = & \begin{cases} [\![e_1]\!](s) \div [\![e_2]\!](s) & \text{if } [\![e_2]\!](s) \neq 0 \\ 0 & \text{otherwise} \end{cases} \\
[\![e_1 \,\texttt{mod}\, e_2]\!](s) & = & \begin{cases} [\![e_1]\!](s) \,\texttt{mod}\, [\![e_2]\!](s) & \text{if } [\![e_2]\!](s) \neq 0 \\ 0 & \text{otherwise} \end{cases}
\end{array}
$$

Semantics of expressions

- $[\![b]\!] : \Sigma \to \{\mathbf{F}, \mathbf{T}\}$ is defined inductively by:

$$
\begin{array}{rcl}
[\![\texttt{true}]\!](s) & = & \mathbf{T} \\
[\![\texttt{false}]\!](s) & = & \mathbf{F} \\
[\![\neg e]\!](s) & = & \begin{cases} \mathbf{T} & \text{if } [\![e]\!](s) = \mathbf{F} \\ \mathbf{F} & \text{if } [\![e]\!](s) = \mathbf{T} \end{cases} \\
[\![e_1 \wedge e_2]\!](s) & = & \begin{cases} \mathbf{F} & \text{if } [\![e_1]\!](s) = \mathbf{F} \\ [\![e_2]\!](s) & \text{otherwise} \end{cases} \\
[\![e_1 \vee e_2]\!](s) & = & \begin{cases} \mathbf{T} & \text{if } [\![e_1]\!](s) = \mathbf{T} \\ [\![e_2]\!](s) & \text{otherwise} \end{cases} \\
[\![e_1 \odot e_2]\!](s) & = & [\![e_1]\!](s) \odot [\![e_2]\!](s), \ \text{where } \odot \in \{=, \neq, <, \leq, >, \geq\}
\end{array}
$$

Assertion semantics

- We take the usual interpretation of first-order formulas, noting two facts:
  - interpretation of assertions also depends on $\mathcal{M}$
  - states from Σ can be used as *variable assignments*
- The interpretation of the assertion $\phi \in \mathbf{Assert}$ is then given by $[\![\phi]\!] : \Sigma \to \{\mathbf{F}, \mathbf{T}\}$
- Since assertions may also contain occurrences of functions and predicates provided by the user, the semantics of those must also be given axiomatically by the user.
- We will be reasoning in the context of a first-order theory that is specified in part by the semantics of program expressions and in part by user-provided axioms.

Program semantics

A natural semantics based on a deterministic evaluation relation

1. $\langle \mathbf{skip}, s\rangle \rightsquigarrow s$
2. $\langle x := e, s\rangle \rightsquigarrow s[x \mapsto [\![e]\!](s)]$
3. if $\langle C_1, s\rangle \rightsquigarrow s'$ and $\langle C_2, s'\rangle \rightsquigarrow s''$, then $\langle C_1 ; C_2, s\rangle \rightsquigarrow s''$
4. if $[\![b]\!](s) = \mathbf{T}$ and $\langle C_t, s\rangle \rightsquigarrow s'$, then $\langle \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, s\rangle \rightsquigarrow s'$
5. if $[\![b]\!](s) = \mathbf{F}$ and $\langle C_f, s\rangle \rightsquigarrow s'$, then $\langle \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, s\rangle \rightsquigarrow s'$
6. if $[\![b]\!](s) = \mathbf{T}$, $\langle C, s\rangle \rightsquigarrow s'$ and $\langle \mathbf{while}\ b\ \mathbf{do}\ C, s'\rangle \rightsquigarrow s''$, then $\langle \mathbf{while}\ b\ \mathbf{do}\ C, s\rangle \rightsquigarrow s''$
7. if $[\![b]\!](s) = \mathbf{F}$, then $\langle \mathbf{while}\ b\ \mathbf{do}\ C, s\rangle \rightsquigarrow s$

There is no possible *runtime error*, but the program may *diverge*.

Validity

- We assume the existence of "external" means for checking the validity of assertions, in the presence of some *background theory*.
- These tools should additionally allow us to write axioms concerning the uninterpreted functions and predicates.
- Suppose that we wish to encode in the logic a description of what the *factorial* of a number is. The following axioms could be given

$$isfact(0,1)$$
$$\forall n, r.\, n > 0 \to isfact(n-1, r) \to isfact(n, n \times r)$$

$$\forall n.\, isfact(n, fact(n))$$
$$\forall n, r.\, isfact(n, r) \to r = fact(n)$$

Hoare triples (for partial correction)

- Notation: $\{\phi\}\, C\, \{\psi\}$
  - ϕ is the *precondition*
  - ψ is the *postcondition*
- Denote the *partial correctness* of program C relative to specification (ϕ, ψ)

Intended meaning of $\{\phi\}\, C\, \{\psi\}$

If ϕ holds in a given state and C is executed in that state, then either execution of C does not stop, or *if it does*, ψ will hold in the final state.

- Examples

$$\{x = y\}\, x := x + y\, ;\, x := 10 * x\, \{x = 20 * y\}$$
$$\{x = 5\}\, \mathbf{while}\ x > 0\ \mathbf{do\ skip}\, \{\mathtt{false}\}$$

Hoare triples (for total correction)

- Notation: $[\phi]\, C\, [\psi]$
- Denote the *total correctness* of program C relative to specification (ϕ, ψ)

Intended meaning of $[\phi]\, C\, [\psi]$

If ϕ holds in a given state and C is executed in that state, then execution of C *will stop*, and moreover ψ will hold in the final state of execution.

- Examples

$$[x = y]\, x := x + y\, ;\, x := 10 * x\, [x = 20 * y]$$
$$[x = 5]\, \mathbf{while}\ x > 0\ \mathbf{do}\ x := x - 1\, [x = 0]$$
$$[\exists a. x = 10 * a]\, x := x + 18\, [\exists v. x = 2 * v]$$

Semantics of Hoare triples

$\models \{\phi\}\, C\, \{\psi\}$

The Hoare triple $\{\phi\}\, C\, \{\psi\}$ is said to be *valid*, denoted $\models \{\phi\}\, C\, \{\psi\}$, whenever for all $s, s' \in \Sigma$,

$$\text{if } [\![\phi]\!](s) = \mathbf{T} \text{ and } \langle C, s\rangle \rightsquigarrow s', \text{ then } [\![\psi]\!](s') = \mathbf{T}.$$

$\models [\phi]\, C\, [\psi]$

The Hoare triple $[\phi]\, C\, [\psi]$ is said to be *valid*, denoted $\models [\phi]\, C\, [\psi]$, whenever for all $s \in \Sigma$,

$$\text{if } [\![\phi]\!](s) = \mathbf{T}, \text{ then } \exists s' \in \Sigma.\, \langle C, s\rangle \rightsquigarrow s' \text{ and } [\![\psi]\!](s') = \mathbf{T}.$$

Hoare logic as an Axiomatic Semantics (system H)

(skip) $\quad \dfrac{}{\{\phi\}\, \mathbf{skip}\, \{\phi\}}$

(assign) $\quad \dfrac{}{\{\psi[e/x]\}\, x := e\, \{\psi\}}$

(seq) $\quad \dfrac{\{\phi\}\, C_1\, \{\theta\} \qquad \{\theta\}\, C_2\, \{\psi\}}{\{\phi\}\, C_1\, ;\, C_2\, \{\psi\}}$

(if) $\quad \dfrac{\{\phi \wedge b\}\, C_t\, \{\psi\} \qquad \{\phi \wedge \neg b\}\, C_f\, \{\psi\}}{\{\phi\}\, \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\, \{\psi\}}$

(while) $\quad \dfrac{\{\theta \wedge b\}\, C\, \{\theta\}}{\{\theta\}\, \mathbf{while}\ b\ \mathbf{do}\ C\, \{\theta \wedge \neg b\}}$

(conseq) $\quad \dfrac{\{\phi\}\, C\, \{\psi\}}{\{\phi'\}\, C\, \{\psi'\}}$ if $\phi' \to \phi$ and $\psi \to \psi'$

Loop invariants

- We call *loop invariant* to any property whose validity is preserved by executions of the loop's body.
- Since these executions may only take place when the loop condition is true, an invariant of the loop $\mathbf{while}\ b\ \mathbf{do}\ C$ is any assertion θ such that $\{\theta \wedge b\}\, C\, \{\theta\}$ is valid, in which case of course it also holds that $\{\theta\}\, \mathbf{while}\ b\ \mathbf{do}\ C\, \{\theta \wedge \neg b\}$ is valid.

Warning

Find an adequate loop invariant may be a major difficulty!

Loop variants

- However the validity of $[\theta \wedge b]\, C\, [\theta]$ does not imply the validity of $[\theta]\, \mathbf{while}\ b\ \mathbf{do}\ C\, [\theta \wedge \neg b]$ *(why?)*
- The required notion here is a *loop variant*: any program expression (or more generally some function on the state) whose value strictly decreases with each iteration, with respect to some well-founded relation.
- The natural choice in our language is to use *non-negative integer* expressions with strictly decreasing values.

(while) $\quad \dfrac{[\theta \wedge b \wedge V = v_0]\, C\, [\theta \wedge V < v_0]}{[\theta]\, \mathbf{while}\ b\ \mathbf{do}\ C\, [\theta \wedge \neg b]}$ if $\theta \wedge b \to V \geq 0$

Soundness

- We will write $\vdash_H \{\phi\}\, C\, \{\psi\}$ to denote the fact that the triple is derivable in this system H.
- Note that the system H contains one rule whose application is guarded by first-order conditions.

$$\text{(conseq)} \quad \frac{\{\phi\}\, C\, \{\psi\}}{\{\phi'\}\, C\, \{\psi'\}} \text{ if } \phi' \to \phi \text{ and } \psi \to \psi'$$

- We will consider that reasoning in this system takes place in the context of the *complete theory* $\mathrm{Th}(\mathcal{M})$ of the implicit structure $\mathcal{M}$, so that when constructing derivations in H one simply checks, when applying the (conseq) rule, whether the side conditions are elements of $\mathrm{Th}(\mathcal{M})$.

System H is sound w.r.t. the semantics of Hoare triples

If $\vdash_H \{\phi\}\, C\, \{\psi\}$, then $\models \{\phi\}\, C\, \{\psi\}$.

Completeness

- Two major difficulties for proving a program:
  - guess the appropriate intermediate formulas (for sequence, for the loop invariant)
  - prove the logical premises of consequence rule
- System H is complete as long as the assertion language is *sufficiently expressive* to grant the existence of intermediate assertions for reasoning.

System H is complete w.r.t. the semantics of Hoare triples

With $\mathbf{Assert}$ expressive in the above sense, if $\models \{\phi\}\, C\, \{\psi\}$ then $\vdash_H \{\phi\}\, C\, \{\psi\}$.

- This is usually called *relative completeness* [Cook, 1978]

Auxiliary variables

- How to specify formally what the following program does?

$$a := x\, ;\, x := y\, ;\, y := a$$

- Employ *auxiliary variables*, forbidden to occur in the program, to record initial values of variables.

$$\{x = x_0 \wedge y = y_0\}\, a := x\, ;\, x := y\, ;\, y := a\, \{x = y_0 \wedge y = x_0\}$$

- In fact, auxiliary variables are required in every specification, to avoid trivial solutions.
  - For instance, an inappropriate specification of factorial would be $(n \geq 0, f = fact(n))$ (Give some solutions!)

Program verification SW uses a *state label mechanism* that allows to refer to the value of a variable in any state.

Exercises

- Prove the validity of the following Hoare triple

$$\{x = x_0 \wedge y = y_0\}\, a := x\, ;\, x := y\, ;\, y := a\, \{x = y_0 \wedge y = x_0\}$$

- How to specify formally what the following program does?

$$\mathbf{if}\ x < 0\ \mathbf{then}\ x := -x\ \mathbf{else\ skip}$$

  Prove its correction w.r.t. the specification proposed.

- Consider the following While$^{\mathrm{int}}$-program for calculating x^e

$$\begin{array}{l} r := 1\,; \\ \mathbf{while}\ e > 0\ \mathbf{do}\ \{ \\ \quad r := r \times x\,; \\ \quad e := e - 1 \\ \} \end{array}$$

  Specify formally what the following program does and prove its correction w.r.t. the specification proposed.

Annotated programs

- We are interested in automated verification
  - invariants are notoriously difficult to infer automatically
  - in practice loop invariants are typically given by the programmer as an input to the program verification process
- The syntactic class of *annotated programs*

  $$\mathbf{AComm} \ni C \ ::= \ \mathbf{skip} \mid C\,;C \mid x := e \mid \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C$$

- Annotations do not affect the operational semantics.
- The (while) rule

$$\frac{\{\theta \wedge b\}\ C\ \{\theta\}}{\{\theta\}\ \mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C\ \{\theta \wedge \neg b\}}$$

Annotated programs

- Whereas in the standard presentation a program can be proved correct with respect to a specification if there exists adequate invariants for proving it, with annotated loops a program can only be proved correct if it is *correctly annotated*.
- Soundness is preserved.
- Completeness does not hold, since the annotated invariants may be inadequate for deriving the triple.

The factorial example

The following is an example of a correctly annotated program w.r.t. the specification

$$(n \geq 0, f = fact(n))$$

Let **fact** be

$$\begin{array}{l} f := 1\,;\, i := 1\,; \\ \mathbf{while}\ i \leq n\ \mathbf{do}\ \{f = fact(i-1) \wedge i \leq n+1\}\ \{ \\ \quad f := f \times i\,; \\ \quad i := i + 1 \\ \} \end{array}$$

A proof of $\{n \geq 0\}\ \mathbf{fact}\ \{f = fact(n)\}$ will be given later.

Handling Arrays

Aliasing

- *Aliasing* in general is a phenomenon that occurs in programming whenever the same object can be accessed through more than one name.

What should be the H rule to deal with array assignment?

- If the standard rule for assignment is used naively, aliasing is handled inadequately.

$$\overline{\{\psi[e'/u[e]]\}\, u[e] := e'\, \{\psi\}}$$

- This axiom is **wrong!**

  It would derive the invalid triple (note that i and j may have equal values)

$$\{u[j] > 100\}\, u[i] := 8\, \{u[j] > 100\}$$

- This phenomenon is called *subscript aliasing*.

While$^{\text{array}}$

We extend the language with arrays as follows

$$\mathbf{Type} \ni \tau ::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{array}$$

$$\mathbf{Exp_{int}} \ni e ::= \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid x \mid -e \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e_1\, \mathtt{div}\, e_2 \mid e_1\, \mathtt{mod}\, e_2 \mid a[e]$$

$$\mathbf{Exp_{array}} \ni a ::= u \mid a[e \triangleright e']$$

$$\mathbf{Exp_{bool}} \ni b ::= \mathtt{true} \mid \mathtt{false} \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid e_1 = e_2 \mid e_1 \neq e_2 \mid e_1 < e_2 \mid e_1 \leq e_2 \mid e_1 > e_2 \mid e_1 \geq e_2$$

The command language is the same. And

$u[e] := e'$ is an abbreviation of $u := u[e \triangleright e']$

where an *array update operator* is used at the term level.

Semantics of expressions of While$^{\text{array}}$

The semantics of While$^{\text{array}}$ expressions is given by extending the semantics of While$^{\text{int}}$ expressions as follows

- $[\![\cdot]\!]$ maps every array $a \in \mathbf{Exp_{array}}$ to a function $[\![a]\!] : \Sigma \to (\mathbb{Z} \to \mathbb{Z})$ defined inductively by

$$[\![u]\!](s) = s(u)$$

$$[\![a[e \triangleright e']]\!](s) = [\![a]\!](s)[[\![e]\!](s) \mapsto [\![e']\!](s)]$$

- the definition of $[\![e]\!] : \Sigma \to \mathbb{Z}$ has the following additional case for integer expressions of the form $a[e]$:

$$[\![a[e]]\!](s) = [\![a]\!](s)([\![e]\!](s))$$

A rule for array assignment

- A correct axiom for array assignment

$$(array\ assign) \qquad \overline{\{\psi[u[e \triangleright e']/u]\}\, u[e] := e'\, \{\psi\}}$$

- This would derive the following valid triple

$$\{u[i \triangleright 8][j] > 100\}\, u[i] := 8\, \{u[j] > 100\}$$

  since the interpretation of $u[i \triangleright 8]$ correctly handles aliasing.

- Arrays are modeled in logic as applicative data structures. Recall the theory of arrays.

Generating Verification Conditions

Mechanising Hoare logic

- In H system two desirable properties for backward proof construction are missing:
  - sub-formula property
  - unambiguous choice of rule

$$\frac{\{\phi\}\, C_1\, \{\theta\} \qquad \{\theta\}\, C_2\, \{\psi\}}{\{\phi\}\, C_1\, ;\, C_2\, \{\psi\}} \qquad\qquad \frac{\{\phi\}\, C\, \{\psi\}}{\{\phi'\}\, C\, \{\psi'\}} \text{ if } \phi' \to \phi \text{ and } \psi \to \psi'$$

- The consequence rule causes ambiguity. Its presence is however necessary to make possible the application of rules for skip, assignment, and while, as well as reuse.
- An alternative is to distribute the side conditions among the different rules.

Hg a goal-directed system

(skip) $$\frac{}{\{\phi\}\, \textbf{skip}\, \{\psi\}} \text{ if } \phi \to \psi$$

(assign) $$\frac{}{\{\phi\}\, x := e\, \{\psi\}} \text{ if } \phi \to \psi[e/x]$$

(seq) $$\frac{\{\phi\}\, C_1\, \{\theta\} \qquad \{\theta\}\, C_2\, \{\psi\}}{\{\phi\}\, C_1\, ;\, C_2\, \{\psi\}}$$

(if) $$\frac{\{\phi \wedge b\}\, C_t\, \{\psi\} \qquad \{\phi \wedge \neg b\}\, C_f\, \{\psi\}}{\{\phi\}\, \textbf{if}\ b\ \textbf{then}\ C_t\ \textbf{else}\ C_f\, \{\psi\}}$$

(while) $$\frac{\{\theta \wedge b\}\, C\, \{\theta\}}{\{\phi\}\, \textbf{while}\ b\ \textbf{do}\, \{\theta\}\, C\, \{\psi\}} \text{ if } \phi \to \theta \text{ and } \theta \wedge \neg b \to \psi$$

Hg properties

Admissibility of the consequence rule in Hg

If $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi\}$, $\models \phi' \to \phi$, and $\models \psi \to \psi'$, then $\vdash_{\mathsf{Hg}} \{\phi'\}\, C\, \{\psi'\}$.

Let $\lfloor\cdot\rfloor : \mathbf{AComm} \to \mathbf{Comm}$ be a function that erases all annotations from a program (defined in the obvious way).

Soundness of Hg

If $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi\}$, then $\vdash_{\mathsf{H}} \{\phi\}\, \lfloor C \rfloor\, \{\psi\}$.

The converse implication does not hold, since the annotated invariants may be inadequate for deriving the triple.

Correctly-annotated program

We say that C is *correctly-annotated* w.r.t. (ϕ, ψ) if $\vdash_{\mathsf{H}} \{\phi\}\, \lfloor C \rfloor\, \{\psi\}$ implies $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi\}$.

A strategy for proofs

- Focus on the command and postcondition; guess an appropriate precondition that guarantees the given postcondition.
- In the rules for skip, assignment, and while, the precondition is determined by looking at the side condition and choosing the weakest condition that satisfies it.
- In the sequence rule, we obtain the intermediate condition by propagating the postcondition.

A strategy for proofs

- $\{\phi\}\, x := e_1 \,;\, y := e_2 \,;\, z := e_3 \,\{\psi\}$

  1. $\{\phi\}\, x := e_1 \,;\, y := e_2 \,\{\theta\}$
  2. $\{\theta\}\, z := e_3 \,\{\psi\}$

- Now the second sub-goal is an assignment, which means that the corresponding axiom can be applied by simply taking the precondition to be the one that trivially satisfies the side condition, i.e. $\theta = \psi[e_3/z]$. Now of course this can be substituted globally in the current proof construction
- $\{\phi\}\, x := e_1 \,;\, y := e_2 \,;\, z := e_3 \,\{\psi\}$

  1. $\{\phi\}\, x := e_1 \,;\, y := e_2 \,\{\psi[e_3/z]\}$
  2. $\{\psi[e_3/z]\}\, z := e_3 \,\{\psi\}$

A strategy for proofs

- $\{\phi\}\, x := e_1 \,;\, y := e_2 \,;\, z := e_3 \,\{\psi\}$

  1. $\{\phi\}\, x := e_1 \,;\, y := e_2 \,\{\psi[e_3/z]\}$
     - 1.1. $\{\phi\}\, x := e_1 \,\{\psi[e_3/z][e_2/y]\}$
     - 1.2. $\{\psi[e_3/z][e_2/y]\}\, y := e_2 \,\{\psi[e_3/z]\}$
  2. $\{\psi[e_3/z]\}\, z := e_3 \,\{\psi\}$

- $\{\phi\}\, x := e_1 \,;\, y := e_2 \,;\, z := e_3 \,\{\psi\}$

  1. $\{\phi\}\, x := e_1 \,;\, y := e_2 \,\{\psi[e_3/z]\}$
     - 1.1. $\{\phi\}\, x := e_1 \,\{\psi[e_3/z][e_2/y]\}$,
     - 1.2. $\{\psi[e_3/z][e_2/y]\}\, y := e_2 \,\{\psi[e_3/z]\}$
  2. $\{\psi[e_3/z]\}\, z := e_3 \,\{\psi\}$

- In step 1.1 we were not free to choose the precondition for the assignment since this is now the first command in the sequence. Thus the side condition $\phi \to \psi[e_3/z][e_2/y][e_1/x]$ is introduced.

Using the weakest precondition strategy to verify **fact**

$\{n \geq 0\}$ **fact** $\{f = fact(n)\}$

1. $\{n \geq 0\}\, f := 1 \,;\, i := 1 \,\{n \geq 0 \wedge f = 1 \wedge i = 1\}$
   - 1.1. $\{n \geq 0\}\, f := 1 \,\{n \geq 0 \wedge f = 1\}$
   - 1.2. $\{n \geq 0 \wedge f = 1\}\, i := 1 \,\{n \geq 0 \wedge f = 1 \wedge i = 1\}$
2. $\{n \geq 0 \wedge f = 1 \wedge i = 1\}$
   $\textbf{while}\ i \leq n\ \textbf{do}\ \{f = fact(i-1) \wedge i \leq n+1\}\, C_w$
   $\{f = fact(n)\}$
   - 2.1. $\{f = fact(i-1) \wedge i \leq n+1 \wedge i \leq n\}\, C_w \,\{f = fact(i-1) \wedge i \leq n+1\}$
     - 2.1.1. $\{f = fact(i-1) \wedge i \leq n+1 \wedge i \leq n\}\, f := f \times i \,\{f = fact(i-1) \times i \wedge i \leq n\}$
     - 2.1.2. $\{f = fact(i-1) \times i \wedge i \leq n\}\, i := i+1 \,\{f = fact(i-1) \wedge i \leq n+1\}$

where C_w represents the command $f := f \times i;\ i := i+1$.

Using the weakest precondition strategy to verify **fact**

- The following side conditions are required for each node of the tree:

  1.1 $n \geq 0 \rightarrow (n \geq 0 \wedge f = 1)[1/f]$
  1.2 $n \geq 0 \wedge f = 1 \rightarrow (n \geq 0 \wedge f = 1 \wedge i = 1)[1/i]$
  2. $n \geq 0 \wedge f = 1 \wedge i = 1 \rightarrow f = fact(i-1) \wedge i \leq n+1$ and
  $f = fact(i-1) \wedge i \leq n+1 \wedge \neg(i \leq n) \rightarrow f = fact(n)$
  2.1.1. $f = fact(i-1) \wedge i \leq n+1 \wedge i \leq n \rightarrow (f = fact(i-1) \times i \wedge i \leq n)[f \times i/f]$
  2.1.2. $f = fact(i-1) \times i \wedge i \leq n \rightarrow (f = fact(i-1) \wedge i \leq n+1)[i+1/i]$

- The validity of these conditions is fairly obvious in the current theory.

An architecture for program verification

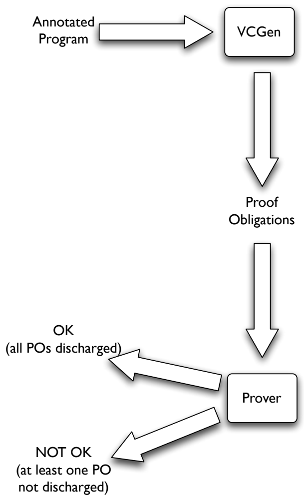
At this point we may outline a method for program verification as follows.

1. Mechanically produce a derivation with $\{\phi\}\, C\, \{\psi\}$ as conclusion, assuming that all the side conditions created in this process hold. The side conditions are called *Verification Conditions (VCs)* or *Proof Obligations (POs)*
2. Send the VCs generated in step 1 to some proof tool in order to be checked.
3. If all VCs are shown to be valid by a proof tool, then $\{\phi\}\, C\, \{\psi\}$ is valid.

Verification Conditions Generator

The mechanisation of the construction of the proof tree following the weakest precondition strategy does not even explicitly construct the proof tree; it just outputs the set of verification conditions.
This algorithm is called a *Verification Conditions Generator (VCGen)*.

An architecture for program verification



Discharging the VCs

- VCs are first-order formulas whose validity is to be checked w.r.t. a *background theory*.
- The VCs are discharged using proof tools.
- *Automated proof tools* (such as SMT-solvers) are usually the first choice.
  - It is possible to use a multi-prover approach (as we will see with Frama-C/Why3)
- If no conclusive answer is given (recall FOL is semi-decidable) one must use a *proof assistant*.
- If the automated prover find a counter-example (or if the interactive proof does not succeed), then we do not have a proof tree for the Hoare triple. That means the verification of the program has *failed!*

Warning

This may be due to errors in the *program*, *specification* or *annotations*!

Weakest liberal precondition

[Dijkstra, 1975]

Given a command C and a postcondition ψ, $\mathsf{wlp}(C,\psi)$ should return the minimal precondition ϕ that validates the triple $\{\phi\}\,C\,\{\psi\}$.

$$\begin{array}{rcl}
\mathsf{wlp}(\textbf{skip},\psi) & = & \psi \\
\mathsf{wlp}(x := e,\psi) & = & \psi[e/x] \\
\mathsf{wlp}(C_1;C_2,\psi) & = & \mathsf{wlp}(C_1,\mathsf{wlp}(C_2,\psi)) \\
\mathsf{wlp}(\textbf{if } b \textbf{ then } C_t \textbf{ else } C_f,\psi) & = & (b \to \mathsf{wlp}(C_t,\psi)) \wedge (\neg b \to \mathsf{wlp}(C_f,\psi)) \\
\mathsf{wlp}(\textbf{while } b \textbf{ do } \{\theta\}\,C,\psi) & = & \theta
\end{array}$$

VCGen algorithm

VC produces a set of verification conditions from a program and a postcondition

$$\begin{array}{rcl}
\mathsf{VC}(\textbf{skip},\psi) & = & \emptyset \\
\mathsf{VC}(x := e,\psi) & = & \emptyset \\
\mathsf{VC}(C_1;C_2,\psi) & = & \mathsf{VC}(C_1,\mathsf{wlp}(C_2,\psi)) \,\cup\, \mathsf{VC}(C_2,\psi) \\
\mathsf{VC}(\textbf{if } b \textbf{ then } C_t \textbf{ else } C_f,\psi) & = & \mathsf{VC}(C_t,\psi) \,\cup\, \mathsf{VC}(C_f,\psi) \\
\mathsf{VC}(\textbf{while } b \textbf{ do } \{\theta\}\,C,\psi) & = & \{(\theta \wedge b) \to \mathsf{wlp}(C,\theta), (\theta \wedge \neg b) \to \psi\} \\
 & & \cup\, \mathsf{VC}(C,\theta)
\end{array}$$

$$\mathsf{VCG}(\{\phi\}\,C\,\{\psi\}) \;=\; \{\phi \to \mathsf{wlp}(C,\psi)\} \,\cup\, \mathsf{VC}(C,\psi)$$

VCGen algorithm

Some observations:

- The function VC simply follows the structure of the rules of system Hg to collect the union of all sets of verification conditions.
- According to the weakest precondition strategy the side conditions generated are trivially satisfied (so we do not collect them).
- In fact, only the loop rule actually introduces verification conditions that need to be checked.
- To understand the clause for loops, it may help to observe that this clause is just an expansion of

$$\mathsf{VC}(\textbf{while } \theta \textbf{ do } \{b\}\,C,\psi) = \{(\theta \wedge \neg b) \to \psi\} \cup \mathsf{VCG}(\{\theta \wedge b\}\,C\,\{\theta\})$$

Properties of VCGen

Soundness

If $\vdash_{\mathsf{Hg}} \{\phi\}\,C\,\{\psi\}$, then

1. $\vdash_{\mathsf{Hg}} \{\mathsf{wlp}(C,\psi)\}\,C\,\{\psi\}$
2. $\models \phi \to \mathsf{wlp}(C,\psi)$

Adequacy of VCGen

$$\models \mathsf{VCG}(\{\phi\}\,C\,\{\psi\}) \quad \text{iff} \quad \vdash_{\mathsf{Hg}} \{\phi\}\,C\,\{\psi\}$$

Applying the VCGen algorithm to **fact**

- Start by calculating VC(**fact**, $f = fact(n)$).
- Then do the calculation of VCG($\{n \geq 0\}$ **fact** $\{f = fact(n)\}$).
- The end result should be the following set of proof obligations.
  1. $n \geq 0 \to 1 = fact(1-1) \wedge 1 \leq n+1$
  2. $f = fact(i-1) \wedge i \leq n+1 \wedge i \leq n \to f \times i = fact(i+1-1) \wedge i+1 \leq n+1$
  3. $f = fact(i-1) \wedge i \leq n+1 \wedge i > n \to f = fact(n)$

  The Frama-C call them
  1. loop invariant init
  2. loop invariant preservation
  3. postcondition

Exercise

Consider the program **maxarray** that determines the position of the largest element in an array between indexes 0 and $size - 1$, where $size \geq 1$. Let **maxarray** be

$$
\begin{array}{l}
max := 0\,; \\
i := 1\,; \\
\textbf{while } i < size \textbf{ do } \{1 \leq i \leq size \wedge 0 \leq max < i \wedge \\
\qquad\qquad\qquad\qquad \forall\, a.\, 0 \leq a < i \to u[a] \leq u[max]\} \\
\{ \\
\quad \textbf{if } u[i] > u[max] \textbf{ then } max := i \textbf{ else skip}\,; \\
\quad i := i + 1 \\
\}
\end{array}
$$

Show that this program indeed meets its specification, i.e.

$$\{size \geq 1\}\ \textbf{maxarray}\ \{0 \leq max < size \wedge \forall\, a.\, 0 \leq a < size \to u[a] \leq u[max]\}$$

Safety-sensitive Hoare Logic

Errors

- So far we have been considering that evaluation of an expression could "never go wrong", and neither could the execution of a command.
- What should the program logic state about:
  - Failing arithmetic operations (division by zero)?
  - The value of an out-of-bounds array position?
  - An assignment command to an out-of-bounds position?

Handling errors

- It is easy to adapt the language semantics to make it more realistic, and to deal with expressions and commands that "can go wrong", by
  - incorporating in the language semantics a special **error** value in the interpretation domains of expressions.
  - modifying the evaluation relation to admit evaluation of commands to a special **error** state.
- For instance, let s be a state such that $s(x) = 10$ and $s(y) = 0$.

$$[\![(x \,\mathtt{div}\, y) > 2]\!](s) = \textbf{error}, \quad \text{because } [\![y]\!](s) = 0$$

and

$$\langle \textbf{if } (x \,\mathtt{div}\, y) > 2 \textbf{ then } C_t \textbf{ else } C_f, s\rangle \rightsquigarrow \textbf{error}$$

no matter what C_t and C_f are.

Evaluation semantics with error state

1. $\langle \textbf{skip}, s\rangle \rightsquigarrow s$.
2. If $[\![e]\!](s) = \textbf{error}$, then $\langle x := e, s\rangle \rightsquigarrow \textbf{error}$.
3. If $[\![e]\!](s) \neq \textbf{error}$, then $\langle x := e, s\rangle \rightsquigarrow s[x \mapsto [\![e]\!](s)]$.
4. If $\langle C_1, s\rangle \rightsquigarrow \textbf{error}$, then $\langle C_1 \,;\, C_2, s\rangle \rightsquigarrow \textbf{error}$.
5. If $\langle C_1, s\rangle \rightsquigarrow s'$, $s' \neq \textbf{error}$, and $\langle C_2, s'\rangle \rightsquigarrow s''$, then $\langle C_1 \,;\, C_2, s\rangle \rightsquigarrow s''$.
6. If $[\![b]\!](s) = \textbf{error}$, then $\langle \textbf{if } b \textbf{ then } C_t \textbf{ else } C_f, s\rangle \rightsquigarrow \textbf{error}$.
7. If $[\![b]\!](s) = \mathbf{T}$ and $\langle C_t, s\rangle \rightsquigarrow s'$, then $\langle \textbf{if } b \textbf{ then } C_t \textbf{ else } C_f, s\rangle \rightsquigarrow s'$.
8. If $[\![b]\!](s) = \mathbf{F}$ and $\langle C_f, s\rangle \rightsquigarrow s'$, then $\langle \textbf{if } b \textbf{ then } C_t \textbf{ else } C_f, s\rangle \rightsquigarrow s'$.
9. If $[\![b]\!](s) = \textbf{error}$, then $\langle \textbf{while } \theta \textbf{ do } \{b\}\, C, s\rangle \rightsquigarrow \textbf{error}$.
10. If $[\![b]\!](s) = \mathbf{T}$ and $\langle C, s\rangle \rightsquigarrow \textbf{error}$, then $\langle \textbf{while } \theta \textbf{ do } \{b\}\, C, s\rangle \rightsquigarrow \textbf{error}$.
11. If $[\![b]\!](s) = \mathbf{T}$, $\langle C, s\rangle \rightsquigarrow s'$, $s' \neq \textbf{error}$, and $\langle \textbf{while } b \textbf{ do } \{\theta\}\, C, s'\rangle \rightsquigarrow s''$, then $\langle \textbf{while } b \textbf{ do } \{\theta\}\, C, s\rangle \rightsquigarrow s''$.
12. If $[\![b]\!](s) = \mathbf{F}$, then $\langle \textbf{while } b \textbf{ do } \{\theta\}\, C, s\rangle \rightsquigarrow s$.

Safety-sensitive semantics of Hoare triples

$\models \{\phi\}\, C\, \{\psi\}$

The Hoare triple $\{\phi\}\, C\, \{\psi\}$ is said to be *valid*, denoted $\models \{\phi\}\, C\, \{\psi\}$, whenever for all $s, s' \in \Sigma$,
if $[\![\phi]\!](s) = \mathbf{T}$ and $\langle C, s\rangle \rightsquigarrow s'$, then $s' \neq \textbf{error}$ and $[\![\psi]\!](s') = \mathbf{T}$.

$\models [\phi]\, C\, [\psi]$

The Hoare triple $[\phi]\, C\, [\psi]$ is said to be *valid*, denoted $\models [\phi]\, C\, [\psi]$, whenever for all $s \in \Sigma$,
if $[\![\phi]\!](s) = \mathbf{T}$, then $\exists s' \in \Sigma.\, \langle C, s\rangle \rightsquigarrow s'$, $s' \neq \textbf{error}$ and $[\![\psi]\!](s') = \mathbf{T}$.

Safety conditions

- We now need to adapt the inference system Hg to cope with this new notion of correctness.
- In order to be able to infer that a command executes without ever going wrong, we need to have the capacity to describe sufficient conditions guaranteeing that program expressions do not evaluate to **error**. These new side conditions will be called *safety conditions*.
- We introduce a function safe : $\bigcup_{\tau \in \mathbf{Type}} \mathbf{Exp}_\tau \to \mathbf{Assert}$.
- The idea is that the truth of the assertion safe(e^τ) in a given state implies that the evaluation of e^τ in that state will not produce an error – the evaluation is safe.
- We define the inference system Hs for safety-sensitive Hoare triples. Naturally its soundness depends on the safe property.

Safe While$^{\text{int}}$ programs

For the While$^{\text{int}}$ language, the function safe can be defined as follows

$$
\begin{array}{rcl}
\mathsf{safe} & : & (\mathbf{Exp_{int}} \cup \mathbf{Exp_{bool}}) \to \mathbf{Assert} \\
\mathsf{safe}(x) & = & \mathtt{true} \\
\mathsf{safe}(c) & = & \mathtt{true} \\
\mathsf{safe}(-e) & = & \mathsf{safe}(e) \\
\mathsf{safe}(e_1 \odot e_2) & = & \mathsf{safe}(e_1) \wedge \mathsf{safe}(e_2), \text{ where } \odot \in \{+,-,\times,=,<,\leq,>,\geq,\neq\} \\
\mathsf{safe}(e_1\,\mathtt{div}\,e_2) & = & \mathsf{safe}(e_1) \wedge \mathsf{safe}(e_2) \wedge e_2 \neq 0 \\
\mathsf{safe}(e_1\,\mathtt{mod}\,e_2) & = & \mathsf{safe}(e_1) \wedge \mathsf{safe}(e_2) \wedge e_2 \neq 0 \\
\mathsf{safe}(\neg b) & = & \mathsf{safe}(b) \\
\mathsf{safe}(b_1 \wedge b_2) & = & \mathsf{safe}(b_1) \wedge (b_1 \to \mathsf{safe}(b_2)) \\
\mathsf{safe}(b_1 \vee b_2) & = & \mathsf{safe}(b_1) \wedge (\neg b_1 \to \mathsf{safe}(b_2))
\end{array}
$$

Safety-sensitive Hoare calculus (system Hs)

(*skip*) $\dfrac{}{\{\!|\phi|\!\}\ \mathbf{skip}\ \{\!|\psi|\!\}}$ if $\phi \to \psi$

(*assign*) $\dfrac{}{\{\!|\phi|\!\}\ x := e\ \{\!|\psi|\!\}}$ if $\phi \to \mathsf{safe}(e)$ and $\phi \to \psi[e/x]$

(*seq*) $\dfrac{\{\!|\phi|\!\}\ C_1\ \{\!|\theta|\!\} \qquad \{\!|\theta|\!\}\ C_2\ \{\!|\psi|\!\}}{\{\!|\phi|\!\}\ C_1\,;\ C_2\ \{\!|\psi|\!\}}$

(*while*) $\dfrac{\{\!|\theta \wedge b|\!\}\ C\ \{\!|\theta|\!\}}{\{\!|\phi|\!\}\ \mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C\ \{\!|\psi|\!\}}$ if $\phi \to \theta$ and $\theta \to \mathsf{safe}(b)$ and $\theta \wedge \neg b \to \psi$

(*if*) $\dfrac{\{\!|\phi \wedge b|\!\}\ C_t\ \{\!|\psi|\!\} \qquad \{\!|\phi \wedge \neg b|\!\}\ C_f\ \{\!|\psi|\!\}}{\{\!|\phi|\!\}\ \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\ \{\!|\psi|\!\}}$ if $\phi \to \mathsf{safe}(b)$

Safety-sensitive VCGen

$$
\begin{array}{rcl}
\mathsf{wlp^s}\,(\mathbf{skip}, \psi) & = & \psi \\
\mathsf{wlp^s}\,(x := e, \psi) & = & \mathsf{safe}(e) \wedge \psi[e/x] \\
\mathsf{wlp^s}\,(C_1; C_2, \psi) & = & \mathsf{wlp^s}\,(C_1, \mathsf{wlp^s}\,(C_2, \psi)) \\
\mathsf{wlp^s}\,(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, \psi) & = & \mathsf{safe}(b) \wedge (b \to \mathsf{wlp^s}\,(C_t, \psi)) \wedge (\neg b \to \mathsf{wlp^s}\,(C_f, \psi)) \\
\mathsf{wlp^s}\,(\mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C, \psi) & = & \theta
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{VC^s}(\mathbf{skip}, \psi) & = & \emptyset \\
\mathsf{VC^s}(x := e, \psi) & = & \emptyset \\
\mathsf{VC^s}(C_1; C_2, \psi) & = & \mathsf{VC^s}(C_1, \mathsf{wlp^s}\,(C_2, \psi)) \ \cup\ \mathsf{VC^s}(C_2, \psi) \\
\mathsf{VC^s}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, \psi) & = & \mathsf{VC^s}(C_t, \psi) \ \cup\ \mathsf{VC^s}(C_f, \psi) \\
\mathsf{VC^s}(\mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C, \psi) & = & \{\theta \to \mathsf{safe}(b), (\theta \wedge b) \to \mathsf{wlp^s}\,(C, \theta), (\theta \wedge \neg b) \to \psi\} \\
 & & \cup\ \mathsf{VC^s}(C, \theta)
\end{array}
$$

$$\mathsf{VCG^s}(\{\!|\phi|\!\}\ C\ \{\!|\psi|\!\}) \ = \ \{\phi \to \mathsf{wlp^s}\,(C, \psi)\} \ \cup\ \mathsf{VC^s}(C, \psi)$$

Properties of Hs and the VCGen

Soundness of Hs

Let $[\![e]\!](s) \neq \mathbf{error}$ whenever $[\![\mathsf{safe}(e)]\!](s) = \mathbf{T}$. Then

$$\text{if } \vdash_{\mathsf{Hs}} \{\!|\phi|\!\}\ C\ \{\!|\psi|\!\}, \text{ then } \models \{\!|\phi|\!\}\ C\ \{\!|\psi|\!\}.$$

Adequacy of the safety-sensitive VCGen

$$\models \mathsf{VCG^s}(\{\!|\phi|\!\}\ C\ \{\!|\psi|\!\}) \quad \text{iff} \quad \vdash_{\mathsf{Hs}} \{\!|\phi|\!\}\ C\ \{\!|\psi|\!\}$$

Bounded arrays: the While$^{\text{array[N]}}$ language

Instead of having a single type **array**, we will have a family of array types $\{\mathbf{array}[N]\}_{N\in\mathbb{N}}$. Expressions of type $\mathbf{array}[N]$ are arrays of length N that admit as valid indexes non-negative integers below N.

$$\begin{array}{llll} \mathbf{Exp}_{\mathbf{array}[N]} & \ni a & ::= & u \mid a[e \triangleright e'] \\ \mathbf{Exp}_{\mathbf{int}} & \ni e & ::= & \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid x \mid \\ & & & -e \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e_1 \,\mathtt{div}\, e_2 \mid e_1 \,\mathtt{mod}\, e_2 \mid \\ & & & a[e] \mid \mathsf{len}(a) \\ \mathbf{Exp}_{\mathbf{bool}} & \ni b & ::= & \mathtt{true} \mid \mathtt{false} \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid e_1 = e_2 \mid e_1 \neq e_2 \\ & & & e_1 < e_2 \mid e_1 \leq e_2 \mid e_1 > e_2 \mid e_1 \geq e_2 \end{array}$$

Semantics of expressions of While$^{\text{array[N]}}$ with error

The semantics of While$^{\text{array[N]}}$ expressions is given by extending the semantics of While$^{\text{int}}$ expressions as follows:

- $[\![a]\!] : \Sigma \to ((\mathbb{Z} \to \mathbb{Z}) \cup \{\textbf{error}\})$ is defined inductively by

$$[\![u]\!](s) = s(u)$$

$$[\![a[e \triangleright e']]\!](s) = \begin{cases} [\![a]\!](s)[[\![e]\!](s) \mapsto [\![e']\!](s)] & \text{if } [\![a]\!](s) \neq \textbf{error} \\ & \text{and } [\![e]\!](s) \neq \textbf{error} \\ & \text{and } 0 \leq [\![e]\!](s) < [\![\mathsf{len}(a)]\!](s) \\ & \text{and } [\![e']\!](s) \neq \textbf{error} \\ \textbf{error} & \text{otherwise} \end{cases}$$

- For integer expressions the definition of $[\![e]\!] : \Sigma \to (\mathbb{Z} \cup \{\textbf{error}\})$ has the following additional cases:

$$[\![\mathsf{len}(a^{\mathbf{array}[N]})]\!](s) = N$$

$$[\![a[e]]\!](s) = \begin{cases} [\![a]\!](s)([\![e]\!](s)) & \text{if } [\![a]\!](s) \neq \textbf{error} \text{ and } [\![e]\!](s) \neq \textbf{error} \\ & \text{and } 0 \leq [\![e]\!](s) < \mathsf{len}(a) \\ \textbf{error} & \text{otherwise} \end{cases}$$

Safe While$^{\text{array[N]}}$ Programs

- For the While$^{\text{array[N]}}$, safe has the following additional cases:

$$\begin{array}{rcl} \mathsf{safe}(u) & = & \mathtt{true} \\ \mathsf{safe}(\mathsf{len}(a)) & = & \mathtt{true} \\ \mathsf{safe}(a[e]) & = & \mathsf{safe}(a) \wedge \mathsf{safe}(e) \wedge 0 \leq e < \mathsf{len}(a) \\ \mathsf{safe}(a[e \triangleright e']) & = & \mathsf{safe}(a) \wedge \mathsf{safe}(e) \wedge 0 \leq e < \mathsf{len}(a) \wedge \mathsf{safe}(e') \end{array}$$

- A rule of system Hs can be given for array assignment, as a special case of rule (*assign*), by expanding the syntactic sugar:

$$\frac{}{\{\!|\phi|\!\}\ u[e] := e'\ \{\!|\psi|\!\}} \quad \text{if} \quad \phi \to \mathsf{safe}(u[e \triangleright e']) \text{ and } \phi \to \psi[u[e \triangleright e']/u]$$

- Clauses of the safety-sensitive VCGen can also be obtained in the same way:

$$\begin{array}{rcl} \mathsf{wlp}^{\mathsf{s}}\,(u[e] := e', \psi) & = & \mathsf{safe}(u[e \triangleright e']) \wedge \psi[u[e \triangleright e']/u] \\ \mathsf{VC}^{\mathsf{s}}(u[e] := e', \psi) & = & \emptyset \end{array}$$

Exercise

- Consider again the program **maxarray**. Show that ehe verification conditions produced by the safety-sensitive VCGen cannot all be proved.
- Consider the following defined predicates concerning the safety of accesses to an individual array position or a contiguous set of positions.

$$\begin{array}{rcl} \mathit{valid\_index}(u, i) & \stackrel{\text{def}}{=} & 0 \leq i < \mathsf{len}(u) \\ \mathit{valid\_range}(u, i, j) & \stackrel{\text{def}}{=} & 0 \leq i \leq j < \mathsf{len}(u) \vee i > j \end{array}$$

Prove that

$$\begin{array}{l} \{\!|\mathit{size} \geq 1 \wedge \mathit{valid\_range}(u, 0, \mathit{size} - 1)|\!\} \\ \mathbf{maxarray} \\ \{\!|0 \leq \mathit{max} < \mathit{size} \wedge \forall\, a.\, 0 \leq a < \mathit{size} \to u[a] \leq u[\mathit{max}]|\!\} \end{array}$$

Continuous invariants

Let **factab** be

$k := 0\,;$
while $k < size$ **do** $\{\theta_2^0 \wedge \theta_2\}\,\{$
  $f := 1\,;\, i := 1\,;\, n := in[k]\,;$
  while $i \leq n$ **do** $\{\theta_1^0 \wedge \theta_1\}\,\{$
    $f := f \times i\,;$
    $i := i + 1$
  $\}$
  $out[k] := f\,;$
  $k := k + 1$
$\}$

where

θ_2^0 is $size \geq 0 \wedge \forall a.\, 0 \leq a < size \rightarrow in[a] \geq 0$
θ_2 is $0 \leq k \leq size \wedge \forall a.\, 0 \leq a < k \rightarrow out[a] = fact(in[a])$
θ_1^0 is $\theta_2^0 \wedge n = in[k] \wedge 0 \leq k < size \wedge\ \forall a.\, 0 \leq a < k \rightarrow out[a] = fact(in[a])$
θ_1 is $1 \leq i \leq n + 1 \wedge f = fact(i - 1)$

- The invariants of the loops have two components:
  - one concerns to the loop task itself (θ_2 and θ_1)
  - the other just transport information between the initial and final states of the loop execution. These are usually called *continuous invariants*.

Continuous invariants

- The need for continuous invariants comes from the verification condition that relates the loop invariant (together with the negated loop condition) and the calculated weakest precondition ψ of the subsequent command.
- The weakest precondition of the loop "forgets" ψ (the postcondition with respect to which it was calculated).
- The continuous invariant plays the role of transporting information between the initial and final states of the loop execution.
- Tools for realistic languages (like the VCGen of Frama-C) are capable of keeping this transported information in the context automatically; there is no need to explicitly include continuous invariants.

Frame conditions

- The following rule is admissible in Hoare logic

$$(\text{frame})\quad \frac{\{\phi\}\, C\, \{\psi\}}{\{\phi \wedge \theta\}\, C\, \{\psi \wedge \theta\}}\quad \text{if no free variable of } \theta \text{ is modified by } C$$

- This rule justifies that program verification tools usually take continuous invariants as implicit. So, they can be omitted in loop invariants. This substantially simplifies the annotated invariants.
- Related to this, its worth mention that annotation languages (like ACSL) usually provide an annotation *assigns* with the list of the variables assigned. These kind of annotations can be placed in routine contracts or in loops.
- Lists of assigned variables explicitly included in contracts are usually called *frame conditions*.
- This kind of annotations will cause specific VCs to be generated.
- A frame condition is an important part of a routine's contract when reasoning about calls to that routine, since it immediately implies the preservation of the values contained in all locations not mentioned.

Bibliography

[RSD 2011] *Rigorous Software Development: An Introduction to Program Verification.* J.B. Almeida & M.J. Frade & J.S. Pinto & S.M. de Sousa. Springer (2011)