Why

A Software Verification Tool

Ming-Hsien Tsai

SVVRL
Dept. of Information Management
National Taiwan University

11/25/2009

Outline

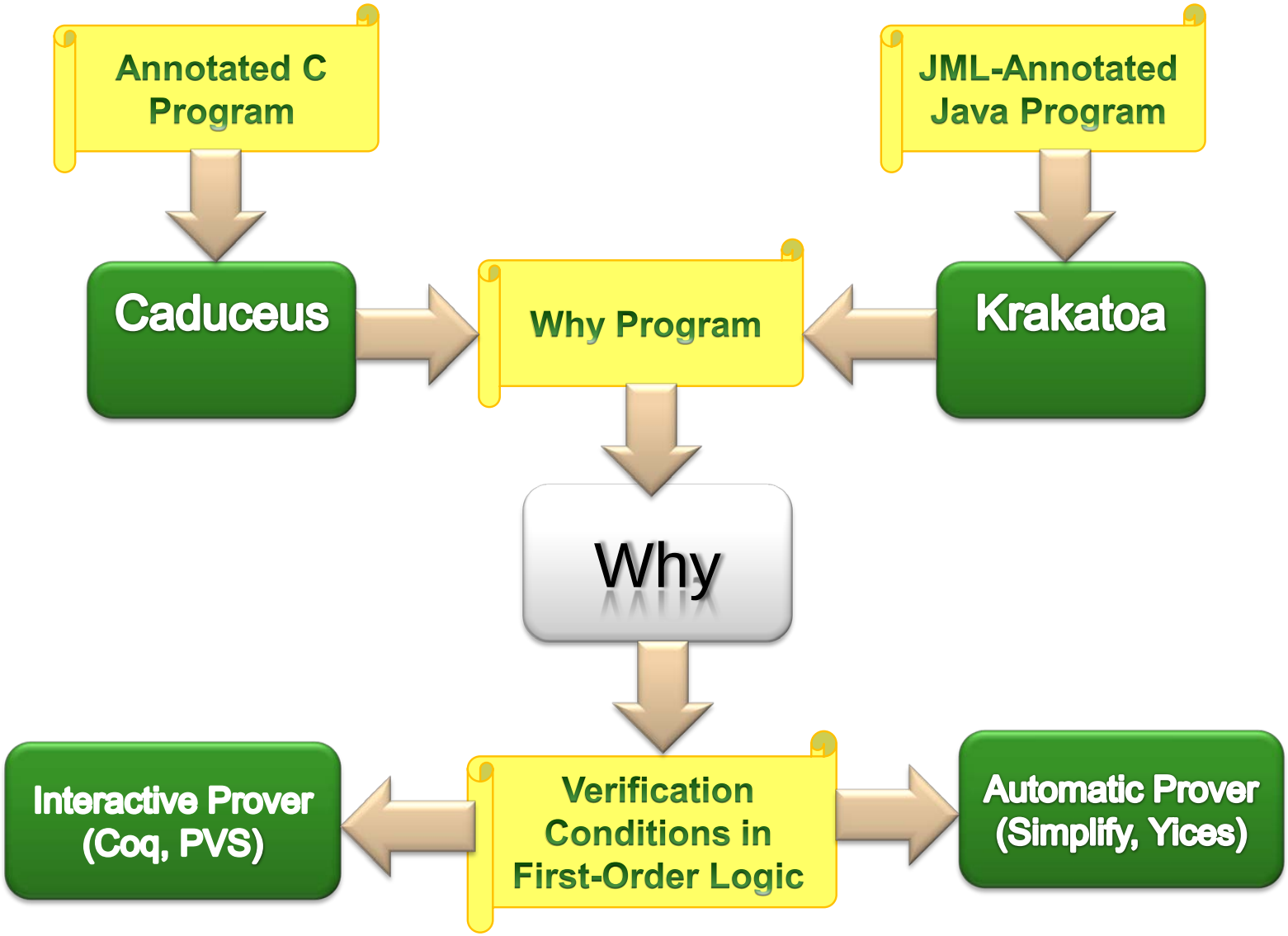
What is Why

- A verification condition generator (VCG)
  - Input: an annotated program in Why language
  - Output: proof obligations for various provers
- An implementation of Dijkstra's weakest preconditions

Features

- Generate verification conditions for Why language.
- Why language is a typed language.
- Allow declaration of types, functions, predicates and axioms.
- Several external provers are supported.
  - Coq, PVS, Isabelle/HOL, HOL4, Hol Light, Mizar
  - Simplify, CVC Lite, haRVey, Zenon, SMTlib
- Tools can be used to verify C (Caduceus,Frama-C) and Java (Krakatoa) programs in combination with Why.
- OCaml code can be generated from Why progrms.

Verification with Why



Verification Condition Generation

Annotated Program

$\{P_1\}$

S_1

$\vdots$

S_i
$\{P_2\}$

S_{i+1}

$\vdots$

S_n
$\{P_3\}$

Verification Conditions

Verification Condition Generation

Annotated Program

$\{P_1\}$

S_1

$\vdots$

S_i

$\{P_2\}$

S_{i+1}

$\vdots$

$\{Q_1\}$[2]

S_n

$\{P_3\}$

Verification Conditions

[2] Q_1 is the weakest precondition for S_n with postcondition P_3.

Verification Condition Generation

Annotated Program	Verification Conditions
$\{P_1\}$	
S_1	
$\vdots$	
S_i	
$\{P_2\}$	
S_{i+1}	
$\{Q_2\}$	
$\vdots$	
$\{Q_1\}$[2]	
S_n	
$\{P_3\}$	

[1] Q_4 is the weakest precondition for S_i with postcondition P_2.

[2] Q_1 is the weakest precondition for S_n with postcondition P_3.

Verification Condition Generation

Annotated Program	Verification Conditions
$\{P_1\}$	
S_1	
$\vdots$	
S_i	
$\{P_2\}$	
$\{Q_3\}$	
S_{i+1}	
$\{Q_2\}$	
$\vdots$	
$\{Q_1\}$[2]	
S_n	
$\{P_3\}$	

[1] Q_4 is the weakest precondition for S_i with postcondition P_2.

[2] Q_1 is the weakest precondition for S_n with postcondition P_3.

Verification Condition Generation

Annotated Program	Verification Conditions
$\{P_1\}$	$P_2 \Rightarrow Q_3$
S_1	
$\vdots$	
S_i	
$\{P_2\}$	
$\{Q_3\}$	
S_{i+1}	
$\{Q_2\}$	
$\vdots$	
$\{Q_1\}$[2]	
S_n	
$\{P_3\}$	

[1] Q_4 is the weakest precondition for S_i with postcondition P_2.

[2] Q_1 is the weakest precondition for S_n with postcondition P_3.

Verification Condition Generation

Annotated Program	Verification Conditions
$\{P_1\}$	$P_2 \Rightarrow Q_3$
S_1	
$\vdots$	
$\{Q_4\}$[1]	
S_i	
$\{P_2\}$	
$\{Q_3\}$	
S_{i+1}	
$\{Q_2\}$	
$\vdots$	
$\{Q_1\}$[2]	
S_n	
$\{P_3\}$	

[1] Q_4 is the weakest precondition for S_i with postcondition P_2.

[2] Q_1 is the weakest precondition for S_n with postcondition P_3.

Verification Condition Generation

Annotated Program	Verification Conditions
$\{P_1\}$	$P_2 \Rightarrow Q_3$
S_1	
$\{Q_5\}$	
$\vdots$	
$\{Q_4\}$[1]	
S_i	
$\{P_2\}$	
$\{Q_3\}$	
S_{i+1}	
$\{Q_2\}$	
$\vdots$	
$\{Q_1\}$[2]	
S_n	
$\{P_3\}$	

[1] Q_4 is the weakest precondition for S_i with postcondition P_2.

[2] Q_1 is the weakest precondition for S_n with postcondition P_3.

Verification Condition Generation

Annotated Program	Verification Conditions
$\{P_1\}$	$P_2 \Rightarrow Q_3$
$\{Q_6\}$	
S_1	
$\{Q_5\}$	
$\vdots$	
$\{Q_4\}$[1]	
S_i	
$\{P_2\}$	
$\{Q_3\}$	
S_{i+1}	
$\{Q_2\}$	
$\vdots$	
$\{Q_1\}$[2]	
S_n	
$\{P_3\}$	

[1] Q_4 is the weakest precondition for S_i with postcondition P_2.

[2] Q_1 is the weakest precondition for S_n with postcondition P_3.

Verification Condition Generation

Annotated Program	Verification Conditions
$\{P_1\}$	$P_2 \Rightarrow Q_3$
$\{Q_6\}$	$P_1 \Rightarrow Q_6$
S_1	
$\{Q_5\}$	
$\vdots$	
$\{Q_4\}$[1]	
S_i	
$\{P_2\}$	
$\{Q_3\}$	
S_{i+1}	
$\{Q_2\}$	
$\vdots$	
$\{Q_1\}$[2]	
S_n	
$\{P_3\}$	

[1] Q_4 is the weakest precondition for S_i with postcondition P_2.

[2] Q_1 is the weakest precondition for S_n with postcondition P_3.

Outline

Why Language

- Why language is a typed language.
- Why language can be used to write
  - logic formulae, and
  - programs.
- Why's syntax is close to OCaml's syntax.

Example

```
let f1 (x : int) =
  {x > 0}
  x - 1
  {x >= 0}
```

Outline

Logic Syntax

$$
\begin{array}{rcl}
\langle term \rangle & ::= & \langle constant \rangle \\
& | & \langle term \rangle\ \langle arith\_op \rangle\ \langle term \rangle \\
& | & \texttt{-}\ \langle term \rangle \\
& | & \langle lab\_identifier \rangle \\
& | & \langle identifier \rangle\ (\ \langle term \rangle^{+}_{,}\ ) \\
& | & \langle lab\_identifier \rangle\ \texttt{[}\ \langle term \rangle\ \texttt{]} \\
& | & \textbf{if}\ \langle term \rangle\ \textbf{then}\ \langle term \rangle\ \textbf{else}\ \langle term \rangle \\
& | & \textbf{let}\ \langle identifier \rangle\ \texttt{=}\ \langle term \rangle\ \textbf{in}\ \langle term \rangle \\
& | & \texttt{(}\ \langle term \rangle\ \texttt{)}
\end{array}
$$

$$
\begin{array}{rcl}
\langle constant \rangle & ::= & \langle integer\text{-}constant \rangle \\
& | & \langle real\text{-}constant \rangle \\
& | & \textbf{true} \\
& | & \textbf{false} \\
& | & \textbf{void}
\end{array}
$$

$$
\langle arith\_op \rangle \quad ::= \quad \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%}
$$

Logic Syntax (cont'd)

$$
\begin{array}{rcl}
\langle predicate \rangle & ::= & \textbf{true} \\
& | & \textbf{false} \\
& | & \langle identifier \rangle \\
& | & \langle identifier \rangle \ \textbf{(}\ \langle term \rangle^{+}_{,}\ \textbf{)} \\
& | & \langle term \rangle\ \langle relation \rangle\ \langle term \rangle\ [\ \langle relation \rangle\ \langle term \rangle\ ] \\
& | & \langle predicate \rangle\ \texttt{->}\ \langle predicate \rangle \\
& | & \langle predicate \rangle\ \texttt{<->}\ \langle predicate \rangle \\
& | & \langle predicate \rangle\ \textbf{or}\ \langle predicate \rangle \\
& | & \langle predicate \rangle\ \textbf{and}\ \langle predicate \rangle \\
& | & \textbf{not}\ \langle predicate \rangle \\
& | & \textbf{if}\ \langle term \rangle\ \textbf{then}\ \langle predicate \rangle\ \textbf{else}\ \langle predicate \rangle \\
& | & \textbf{let}\ \langle identifier \rangle\ \texttt{=}\ \langle term \rangle\ \textbf{in}\ \langle predicate \rangle \\
& | & \textbf{forall}\ \langle identifier \rangle^{+}_{,}\ \textbf{:}\ \langle primitive\_type \rangle\ [\ \langle triggers \rangle\ ]\ \textbf{.}\ \langle predicate \rangle \\
& | & \textbf{exists}\ \langle identifier \rangle^{+}_{,}\ \textbf{:}\ \langle primitive\_type \rangle\ [\ \langle triggers \rangle\ ]\ \textbf{.}\ \langle predicate \rangle \\
& | & \textbf{(}\ \langle predicate \rangle\ \textbf{)} \\
& | & (\langle identifier \rangle \mid \langle string \rangle)\ \textbf{:}\ \langle predicate \rangle \\
\\
\langle triggers \rangle & ::= & [\ \langle trigger \rangle^{+}_{|}\ ] \\
\langle trigger \rangle & ::= & \langle term \rangle^{+}_{,} \\
\\
\langle primitive\_type \rangle & ::= & \textbf{int} \mid \textbf{bool} \mid \textbf{real} \mid \textbf{unit} \mid \langle identifier \rangle \mid \texttt{'}\langle identifier \rangle \\
& | & \langle primitive\_type \rangle\ \langle identifier \rangle \mid \textbf{(}\ \langle primitive\_type \rangle^{*}_{,}\ \textbf{)}\ \langle identifier \rangle \\
\\
\langle relation \rangle & ::= & \texttt{=} \mid \texttt{<>} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=}
\end{array}
$$

Logic Syntax (cont'd)

$$
\begin{array}{rcl}
\langle \mathit{l\_declaration} \rangle & ::= & [\ \textbf{external}\ ]\ \textbf{type}\ [\ \langle \mathit{type\_parameters} \rangle\ ]\ \langle \mathit{identifier} \rangle \\
& | & [\ \textbf{external}\ ]\ \textbf{logic}\ \langle \mathit{identifier} \rangle^{+}_{,}\ \textbf{:}\ \langle \mathit{logic\_type} \rangle \\
& | & \textbf{function}\ \langle \mathit{identifier} \rangle\ \textbf{(}\ \langle \mathit{logic\_binder} \rangle^{*}_{,}\ \textbf{)}\ \textbf{:}\ \langle \mathit{primitive\_type} \rangle \\
& & \textbf{=}\ \langle \mathit{term} \rangle \\
& | & \textbf{predicate}\ \langle \mathit{identifier} \rangle\ \textbf{(}\ \langle \mathit{logic\_binder} \rangle^{*}_{,}\ \textbf{)}\ \textbf{=}\ \langle \mathit{predicate} \rangle \\
& | & \textbf{inductive}\ \langle \mathit{identifier} \rangle\ \textbf{:}\ \langle \mathit{logic\_type} \rangle\ \textbf{=} \\
& & (\textbf{|}\ \langle \mathit{identifier} \rangle\ \textbf{:}\ \langle \mathit{predicate} \rangle)^{*} \\
& | & \textbf{axiom}\ \langle \mathit{identifier} \rangle\ \textbf{:}\ \langle \mathit{predicate} \rangle \\
& | & \textbf{goal}\ \langle \mathit{identifier} \rangle\ \textbf{:}\ \langle \mathit{predicate} \rangle \\
\\
\langle \mathit{logic\_type} \rangle & ::= & \langle \mathit{logic\_arg\_type} \rangle^{*}_{,}\ \textbf{->}\ \textbf{prop} \\
& | & \langle \mathit{logic\_arg\_type} \rangle^{*}_{,}\ \textbf{->}\ \langle \mathit{primitive\_type} \rangle \\
\\
\langle \mathit{logic\_arg\_type} \rangle & ::= & \langle \mathit{primitive\_type} \rangle \mid \langle \mathit{primitive\_type} \rangle\ \textbf{array} \\
\\
\langle \mathit{logic\_binder} \rangle & ::= & \langle \mathit{identifier} \rangle\ \textbf{:}\ \langle \mathit{primitive\_type} \rangle \\
\\
\langle \mathit{type\_parameters} \rangle & ::= & \textbf{'}\langle \mathit{identifier} \rangle \mid \textbf{(}\ (\textbf{'}\langle \mathit{identifier} \rangle)^{+}_{,}\ \textbf{)}
\end{array}
$$

Examples

Type Declaration

```
type color
type 'a list
```

Logic Declaration

```
logic white : color
logic black : color
predicate is_color (c : color) = c = white or c = black

logic x : int list
logic length : 'a list -> int
logic insert : 'a, 'a list -> 'a list
axiom insert_length :
  forall x : 'a. forall l : 'a list.
    length(insert(x, l)) = length(l) + 1
```

Logic Semantics

- Types, terms, and predicates

$$
\begin{array}{rcl}
\tau & ::= & \alpha \mid (\tau, \ldots, \tau)\ s \\
t & ::= & x \mid f(t, \ldots, t) \\
P & ::= & p(t, \ldots, t) \\
 & \mid & \top \mid \bot \mid P \wedge P \mid P \vee P \mid \neg P \mid P \Rightarrow P \\
 & \mid & \forall x : \tau.\ P \mid \exists x : \tau.\ P
\end{array}
$$

- A theory Σ is a finite list of declarations δ where

$$
\begin{array}{rcl}
\delta & ::= & \textbf{type}\ \vec{\alpha}\ s \\
 & \mid & x : \tau \\
 & \mid & \textbf{logic}\ f : \forall \vec{\alpha}.\tau, \ldots, \tau \rightarrow \tau \\
 & \mid & \textbf{logic}\ p : \forall \vec{\alpha}.\tau, \ldots, \tau \rightarrow \textbf{prop} \\
 & \mid & \textbf{axiom}\ \forall \vec{\alpha}.P \\
 & \mid & \textbf{goal}\ \forall \vec{\alpha}.P
\end{array}
$$

Well-Formed Types ($\Sigma \vdash \tau$ wf)

$$\frac{}{\Sigma \vdash \alpha \text{ wf}}\ \text{Ty}_1$$

$$\frac{\textbf{type}\ (\alpha_1, \ldots, \alpha_n)\ s \in \Sigma \qquad \forall i, \Sigma \vdash \tau_i \text{ wf}}{\Sigma \vdash (\tau_1, \ldots, \tau_n)\ s \text{ wf}}\ \text{Ty}_2$$

Examples

```
color
int list
color list
```

Well-Typed Terms ($\Sigma \vdash t : \tau$)

$$\frac{x : \tau \in \Sigma}{\Sigma \vdash x : \tau}\ \mathsf{T}_1$$

$$\frac{\textbf{logic}\ f : \forall \vec{\alpha}.\tau_1, \ldots, \tau_n \to \tau \in \Sigma \qquad \mathtt{Subst}(\sigma, \Sigma) \qquad \forall i, \Sigma \vdash t_i : \sigma(\tau_i)}{\Sigma \vdash f(t_1, \ldots, t_n) : \sigma(\tau)}\ \mathsf{T}_2$$

where σ is a mapping from type variables to types, and $\mathtt{Subst}(\sigma, \Sigma)$ means that $\Sigma \vdash \sigma(\alpha)$ `wf` holds for any type variable α

Examples

```
length(x) : int
insert(0, x) : int list
```

Well-Typed Predicates ($\Sigma \vdash P$ wf)

$$\frac{\textbf{logic}\ p : \forall \vec{\alpha}.\tau_1, \ldots, \tau_n \to \textbf{prop} \in \Sigma \qquad \mathtt{Subst}(\sigma, \Sigma) \qquad \forall i, \Sigma \vdash t_i : \sigma(\tau_i)}{\Sigma \vdash p(t_1, \ldots, t_n)\ \mathtt{wf}}\ \mathsf{P}_1$$

$$\frac{}{\Sigma \vdash \top\ \mathtt{wf}}\ \mathsf{P}_2 \qquad \frac{}{\Sigma \vdash \bot\ \mathtt{wf}}\ \mathsf{P}_3$$

$$\frac{\Sigma \vdash P_1\ \mathtt{wf} \qquad \Sigma \vdash P_2\ \mathtt{wf}}{\Sigma \vdash P_1 \wedge P_2 \mathtt{wf}}\ \mathsf{P}_4 \qquad \frac{\Sigma \vdash P_1\ \mathtt{wf} \qquad \Sigma \vdash P_2\ \mathtt{wf}}{\Sigma \vdash P_1 \vee P_2 \mathtt{wf}}\ \mathsf{P}_5$$

$$\frac{\Sigma \vdash P\ \mathtt{wf}}{\Sigma \vdash \neg P \mathtt{wf}}\ \mathsf{P}_6 \qquad \frac{\Sigma \vdash P_1\ \mathtt{wf} \qquad \Sigma \vdash P_2\ \mathtt{wf}}{\Sigma \vdash P_1 \Rightarrow P_2 \mathtt{wf}}\ \mathsf{P}_7$$

$$\frac{\Sigma, x : \tau \vdash P\ \mathtt{wf}}{\Sigma \vdash \forall x : \tau.P\ \mathtt{wf}}\ \mathsf{P}_8 \qquad \frac{\Sigma, x : \tau \vdash P\ \mathtt{wf}}{\Sigma \vdash \exists x : \tau.P\ \mathtt{wf}}\ \mathsf{P}_8$$

Well-Formed Theories ($\vdash \Sigma$ wf)

$$\frac{}{\vdash \texttt{wf}}\ \mathsf{Th}_1 \qquad \frac{\textbf{type}\ s \notin \Sigma}{\vdash \Sigma, \textbf{type}\ (\alpha_1, \ldots, \alpha_2)\ s\ \texttt{wf}}\ \mathsf{Th}_2 \qquad \frac{x \notin \Sigma \qquad \Sigma \vdash \tau\ \texttt{wf}}{\vdash \Sigma, x : \tau\ \texttt{wf}}\ \mathsf{Th}_3$$

$$\frac{\textbf{logic}\ f \notin \Sigma \qquad \forall i, \Sigma \vdash \tau_i\ \texttt{wf}}{\vdash \Sigma, \textbf{logic}\ f : \forall \vec{\alpha}.\ \tau_1, \ldots, \tau_n \rightarrow \tau_{n+1}\ \texttt{wf}}\ \mathsf{Th}_4$$

$$\frac{\textbf{logic}\ p \notin \Sigma \qquad \forall i, \Sigma \vdash \tau_i\ \texttt{wf}}{\vdash \Sigma, \textbf{logic}\ p : \forall \vec{\alpha}.\ \tau_1, \ldots, \tau_n \rightarrow \textbf{prop}\ \texttt{wf}}\ \mathsf{Th}_5$$

$$\frac{}{\vdash \Sigma, \textbf{axiom}\ \forall \vec{\alpha}.\ P\ \texttt{wf}}\ \mathsf{Th}_6 \qquad \frac{}{\vdash \Sigma, \textbf{goal}\ \forall \vec{\alpha}.\ P\ \texttt{wf}}\ \mathsf{Th}_6$$

Validity

Validity is defined as a set of natural deduction rules ($\Sigma \models P$).

Deduction Rules

$$\frac{\textbf{axiom}\ \forall\vec{\alpha}.\ P \in \Sigma \qquad \mathtt{Subst}(\sigma, \Sigma)}{\Sigma \models \sigma(P)}\ \text{Ax} \qquad \frac{\Sigma \models Q \qquad \Sigma, Q \models P}{\Sigma \models \sigma(P)}\ \text{Cut}$$

$$\frac{}{\Sigma \models \top}\ \text{True} \qquad \frac{\Sigma \models \bot \qquad \Sigma \vdash P\ \mathtt{wf}}{\Sigma \models P}\ \text{False} \qquad \frac{\Sigma \vdash P\ \mathtt{wf}}{\Sigma \models P \vee \neg P}\ \text{EM}$$

and many other rules not listed here

Outline

Program Syntax

$$
\begin{array}{rcl}
\langle simple\_value\_type \rangle & ::= & \langle primitive\_type \rangle \mid \langle primitive\_type \rangle \ \textbf{ref} \\
& \mid & \langle primitive\_type \rangle \ \textbf{array} \mid \textbf{(} \ \langle value\_type \rangle \ \textbf{)} \\
\\
\langle value\_type \rangle & ::= & \langle simple\_value\_type \rangle \\
& \mid & \langle simple\_value\_type \rangle \ \textbf{->} \ \langle computation\_type \rangle \\
& \mid & \langle identifier \rangle \ \textbf{:} \ \langle simple\_value\_type \rangle \ \textbf{->} \ \langle computation\_type \rangle \\
\\
\langle computation\_type \rangle & ::= & \textbf{\{} \ [\ \langle precondition \rangle \ ] \ \textbf{\}} \\
& & [\ \textbf{returns} \ \langle identifier \rangle \ \text{key:} \ ] \ \langle value\_type \rangle \ \langle effects \rangle \\
& & \textbf{\{} \ [\ \langle postcondition \rangle \ ] \ \textbf{\}} \\
& \mid & \langle value\_type \rangle \\
\\
\langle effects \rangle & ::= & [\ \textbf{reads} \ \langle identifier \rangle^*_{,} \ ] \\
& & [\ \textbf{writes} \ \langle identifier \rangle^*_{,} \ ] \\
& & [\ \textbf{raises} \ \langle identifier \rangle^*_{,} \ ] \\
\\
\langle precondition \rangle & ::= & \langle assertion \rangle \\
\\
\langle postcondition \rangle & ::= & \langle assertion \rangle \ \langle exn\_condition \rangle^* \\
\\
\langle exn\_condition \rangle & ::= & \mid \ \langle identifier \rangle \ \text{=>} \ \langle assertion \rangle \\
\\
\langle assertion \rangle & ::= & \langle predicate \rangle \ [\ \textbf{as} \ \langle identifier \rangle \ ]
\end{array}
$$

Program Syntax (cont'd)

$$
\begin{array}{lcl}
\langle prog \rangle & ::= & \langle constant \rangle \mid \langle identifier \rangle \mid \texttt{!}\ \langle identifier \rangle \\
& \mid & \langle identifier \rangle\ \texttt{:=}\ \langle prog \rangle \\
& \mid & \langle identifier \rangle\ \texttt{[}\ \langle prog \rangle\ \texttt{]} \\
& \mid & \langle identifier \rangle\ \texttt{[}\ \langle prog \rangle\ \texttt{]}\ \texttt{:=}\ \langle prog \rangle \\
& \mid & \langle prog \rangle\ \langle infix \rangle\ \langle prog \rangle \mid \langle prefix \rangle\ \langle prog \rangle \\
& \mid & \textbf{let}\ \langle identifier \rangle\ \texttt{=}\ \langle prog \rangle\ \textbf{in}\ \langle prog \rangle \\
& \mid & \textbf{let}\ \langle identifier \rangle\ \texttt{=}\ \textbf{ref}\ \langle prog \rangle\ \textbf{in}\ \langle prog \rangle \\
& \mid & \textbf{if}\ \langle prog \rangle\ \textbf{then}\ \langle prog \rangle\ [\ \textbf{else}\ \langle prog \rangle\ ] \\
& \mid & \textbf{while}\ \langle prog \rangle\ \text{keydo}\ [\ \langle loop\_annot \rangle\ ]\ \langle prog \rangle\ \textbf{done} \\
& \mid & \langle prog \rangle\ \texttt{;}\ \langle prog \rangle \\
& \mid & \langle identifier \rangle\ \texttt{:}\ \langle prog \rangle \\
& \mid & \textbf{assert}\ (\{\ \langle assertion \rangle\ \})^{+}\ \texttt{;}\ \langle prog \rangle \\
& \mid & \langle prog \rangle\ \{\ \langle postcondition \rangle\ \} \mid \langle prog \rangle\ \{\{\ \langle postcondition \rangle\ \}\} \\
& \mid & \textbf{fun}\ \langle binders \rangle\ \texttt{->}\ \langle prog \rangle \\
& \mid & \textbf{let}\ \langle identifier \rangle\ \langle binders \rangle\ \texttt{=}\ \langle prog \rangle\ \textbf{in}\ \langle prog \rangle \\
& \mid & \textbf{let rec}\ \langle recfun \rangle\ [\ \textbf{in}\ \langle prog \rangle\ ] \\
& \mid & \langle prog \rangle\ \langle prog \rangle \\
& \mid & \textbf{raise}\ \langle identifier \rangle\ [\ \texttt{:}\ \langle value\_type \rangle\ ] \\
& \mid & \textbf{raise}\ \texttt{(}\ \langle identifier \rangle\ \langle prog \rangle\ \texttt{)}\ [\ \texttt{:}\ \langle value\_type \rangle\ ] \\
& \mid & \textbf{try}\ \langle prog \rangle\ \textbf{with}\ \langle handler \rangle^{+}_{|}\ \textbf{end} \\
& \mid & \textbf{absurd}\ [\ \texttt{:}\ \langle value\_type \rangle\ ] \\
& \mid & \texttt{(}\ \langle prog \rangle\ \texttt{)} \\
& \mid & \textbf{begin}\ \langle prog \rangle\ \textbf{end}
\end{array}
$$

Program Syntax (cont'd)

$$
\begin{array}{rcl}
\langle \mathit{infix} \rangle & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{=} \mid \texttt{<>} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{||} \mid \texttt{\&\&} \\
\langle \mathit{prefix} \rangle & ::= & \texttt{-} \mid \mathrm{not} \\
\\
\langle \mathit{binders} \rangle & ::= & \texttt{(}\ \langle \mathit{identifier} \rangle^{+}_{,}\ \texttt{:}\ \langle \mathit{value\_type} \rangle\ \texttt{)+} \\
\\
\langle \mathit{recfun} \rangle & ::= & \langle \mathit{identifier} \rangle\ \langle \mathit{binders} \rangle\ \texttt{:}\ \langle \mathit{value\_type} \rangle \\
 & & \texttt{\{ variant}\ \langle \mathit{wf\_arg} \rangle\ \texttt{\}} = \langle \mathit{prog} \rangle \\
\\
\langle \mathit{loop\_annot} \rangle & ::= & \texttt{\{}\ [\ \texttt{invariant}\ \langle \mathit{assertion} \rangle\ ]\ [\ \texttt{variant}\ \langle \mathit{wf\_arg} \rangle\ ]\ \texttt{\}} \\
\\
\langle \mathit{wf\_arg} \rangle & ::= & \langle \mathit{term} \rangle\ [\ \texttt{for}\ \langle \mathit{identifier} \rangle\ ] \\
\\
\langle \mathit{handler} \rangle & ::= & \langle \mathit{identifier} \rangle\ \texttt{->}\ \langle \mathit{prog} \rangle \\
 & \mid & \langle \mathit{identifier} \rangle\ \langle \mathit{identifier} \rangle\ \texttt{->}\ \langle \mathit{prog} \rangle
\end{array}
$$

Program Types and Specifications

- Program types: θ
- Specifications: κ
- Postconditions: q
- Effects: ϵ

$$
\begin{array}{lcl}
\theta & ::= & \tau \mid \tau \ \mathbf{ref} \mid (x : \theta) \rightarrow \kappa \\
\kappa & ::= & \{p\}\theta \ \epsilon\{q\} \\
q & ::= & p; E \Rightarrow p; \ldots; E \Rightarrow p \\
\epsilon & ::= & \mathbf{reads}\ x, \ldots, x\ \mathbf{writes}\ x, \ldots, x\ \mathbf{raises}\ E, \ldots, E
\end{array}
$$

Program Expressions

$$
\begin{array}{lll}
t & ::= & x \mid !x \mid f(t, \ldots, t) \\
\\
e & ::= & t \\
& & \mid \textbf{let } x = e \textbf{ in } e \\
& & \mid \textbf{let } x = \textbf{ref } e \textbf{ in } e \\
& & \mid \textbf{if } e \textbf{ then } e \textbf{ else } e \\
& & \mid \textbf{loop } e \ \{\textbf{invariant } p \textbf{ variant } t\} \\
& & \mid L : e \\
& & \mid \textbf{raise } (E\ e) : \theta \\
& & \mid \textbf{try } e \textbf{ with } E\ x \rightarrow e \textbf{ end} \\
& & \mid \textbf{assert } \{p\};\ e \\
& & \mid e\ \{q\} \\
& & \mid \mathrm{e}\ \{\{q\}\} \\
& & \mid \textbf{fun } (x : \theta) \rightarrow \{p\}\ e \\
& & \mid \textbf{rec } x\ (x : \theta) \ldots (x : \theta) : \theta\ \{\textbf{variant } t\} = \{p\}\ e \\
& & \mid e\ e
\end{array}
$$

Programs

$$
\begin{array}{lcl}
p & ::= & \emptyset \mid d\ p \\
d & ::= & \textbf{let}\ x = e \\
 & \mid & \textbf{val}\ x : \theta \\
 & \mid & \textbf{exception}\ E\ \text{of}\ \tau
\end{array}
$$

Syntactic Sugar

$$
\begin{array}{rcl}
e_1; e_2 & \equiv & \textbf{let } \_ = e_1 \textbf{ in } e_2 \\
x \textbf{ := } e & \equiv & \textbf{ref\_set } x\ e \\
\textbf{raise } E & \equiv & \textbf{raise } (E\ \textbf{void}) : \textbf{unit}
\end{array}
$$

$$
\begin{array}{l}
\textbf{while } e_1 \textbf{ do } e_2 \ \{\textbf{invariant } p \textbf{ variant } t\} \equiv \\
\quad \textbf{try} \\
\quad\quad \textbf{loop if } e_1 \textbf{ then } e_2 \textbf{ else raise } \mathit{Exit} \\
\quad\quad \{\textbf{invariant } p \textbf{ variant } t\} \\
\quad \textbf{with } \mathit{Exit}\ \_ \textbf{ -> void end}
\end{array}
$$

Parameters

A program without implementation can be defined by parameter. For example,

```
parameter ref_set : x:'a ref -> v:'a ->
                    {} unit writes x { x = v }
```

Typing

- Typing judgement: $\Sigma \vdash e : (\theta, \epsilon)$
  - e has type θ and effect ϵ in the environment Σ.
- An effect **reads** r **writes** w **raises** e is written as (r, w, e).
- $\bot = (\emptyset, \emptyset, \emptyset)$
- $(r_1, w_1, e_1) \sqcup (r_2, w_2, e_2) = (r_1 \cup r_2, w_1 \cup w_2, e_1 \cup e_2)$
- $(r, w, e) \backslash x = (r \backslash \{x\}, w \backslash \{x\}, e \backslash \{x\})$
- A main feature of the typing system is to prevent aliases.

Typing (cont'd)

$$\frac{Typeof(c) = \tau}{\Sigma \vdash c : (\tau, \bot)} \qquad \frac{x : \theta \in \Sigma \qquad \theta \text{ pure}}{\Sigma \vdash x : (\theta, \bot)} \qquad \frac{x : \tau \ \mathbf{ref} \in \Sigma}{\Sigma \vdash !x : (\tau, \mathbf{reads}\ x)}$$

$$\frac{\Sigma \vdash t_i : (\tau_i, \epsilon_i) \qquad Typeof(\phi) = \tau_1, \ldots, \tau_n \rightarrow \tau}{\Sigma \vdash \phi(t_1, \ldots, t_n) : (\tau, \bigsqcup_i \epsilon_i)}$$

$$\frac{x : \tau \ \mathbf{ref} \in \Sigma \qquad \Sigma \vdash e : (\tau, \epsilon)}{\Sigma \vdash x \ \mathbf{:=}\ e : (\mathbf{unit}, (\mathbf{writes}\ x) \sqcup \epsilon)}$$

$$\frac{\Sigma \vdash e_1 : (\theta_1, \epsilon_1) \qquad \theta_1 \text{ pure} \qquad \Sigma, x : \theta_1 \vdash e_2 : (\theta_2, \epsilon_2)}{\Sigma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : (\theta_2, \epsilon_1 \sqcup \epsilon_2)}$$

Typing (cont'd)

$$\frac{\Sigma \vdash e_1 : (\tau_1, \epsilon_1) \qquad \Sigma, x : \tau_1\ \mathbf{ref} \vdash e_2 : (\theta_2, \epsilon_2) \qquad x \notin \theta_2}{\Sigma \vdash \mathbf{let}\ x = \mathbf{ref}\ e_1\ \mathbf{in}\ e_2 : (\theta_2, \epsilon_1 \sqcup \epsilon_2 \backslash x)}$$

$$\frac{\Sigma \vdash e_1 : (\mathbf{bool}, \epsilon_1) \qquad \Sigma \vdash e_2 : (\theta, \epsilon_2) \qquad \Sigma \vdash e_3 : (\theta, \epsilon_3)}{\Sigma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : (\theta, \epsilon_1 \sqcup \epsilon_2 \sqcup \epsilon_3)}$$

$$\frac{\Sigma \vdash e : (\mathbf{unit}, \epsilon) \qquad \Sigma \vdash p\ \mathtt{wf} \qquad \Sigma \vdash t : \mathbf{int}}{\Sigma \vdash \mathbf{loop}\ e\ \{\mathbf{invariant}\ p\ \mathbf{variant}\ t\} : (\mathbf{unit}, \epsilon)}$$

$$\frac{\Sigma, \mathbf{label}\ L \vdash e : (\theta, \epsilon)}{\Sigma \vdash L{:}e : (\theta, \epsilon)}$$

$$\frac{\mathbf{exception}\ E\ \mathbf{of}\ \tau \in \Sigma \qquad \Sigma \vdash e : (\tau, \epsilon)}{\Sigma \vdash \mathbf{raise}\ (E\ e) : \theta : (\theta, (\mathbf{raises}\ E) \sqcup \epsilon))}$$

Typing (cont'd)

$$\frac{\textbf{exception } E \textbf{ of } \tau \in \Sigma \qquad \Sigma \vdash e_1 : (\theta, \epsilon_1) \qquad \Sigma, x : \tau \vdash e_2 : (\theta, \epsilon_2)}{\Sigma \vdash \textbf{try } e_1 \textbf{ with } E\ x \rightarrow e_2 \textbf{ end} : (\theta, \epsilon_1 \backslash \{\textbf{raise } E\} \sqcup \epsilon_2)}$$

$$\frac{\Sigma \vdash p \text{ wf} \qquad \Sigma \vdash e : (\theta, \epsilon)}{\Sigma \vdash \textbf{assert } \{p\}; e : (\theta, \epsilon)} \qquad \frac{\Sigma \vdash e : (\theta, \epsilon) \qquad \Sigma, \mathit{result} : \theta \vdash q \text{ wf}}{\Sigma \vdash e\ \{q\} : (\theta, \epsilon)}$$

$$\frac{\Sigma, x : \theta \vdash p \text{ wf} \qquad \Sigma, x : \theta \vdash e\ \{q\} : (\theta', \epsilon)}{\Sigma \vdash \textbf{fun } (x : \theta) \rightarrow \{p\}\ e\ \{q\} : ((x : \theta) \rightarrow \{p\}\theta'\ \epsilon\{q\}, \bot)}$$

$$\frac{\Sigma' \equiv \Sigma, x_1 : \theta_1, \ldots, x_n : \theta_n \qquad \Sigma' \vdash p \text{ wf} \qquad \Sigma' \vdash t\ \textbf{int} \qquad \Sigma', f : (x_1 : \theta_1) \rightarrow \cdots (x_n : \theta_n) \rightarrow \{p\}\theta\ \epsilon\{q\} \vdash e\ \{q\} : (\theta, \epsilon)}{\begin{array}{c}\Sigma \vdash \textbf{rec } f (x_1 : \theta_1) \ldots (x_n : \theta_n) : \theta\ \{\textbf{variant } t\} = \{p\}\ e\ \{q\} \\ : ((x_1 : \theta_1) \rightarrow \cdots (x_n : \theta_n) \rightarrow \{p\}\theta\ \epsilon\{q\}, \bot)\end{array}}$$

Typing (cont'd)

$$\frac{\Sigma \vdash e_1 : ((x : \theta_2) \rightarrow \{p\}\theta\ \epsilon\{q\}, \epsilon_1) \qquad \Sigma \vdash e_2 : (\theta_2, \epsilon_2) \qquad \theta_2\ \mathtt{pure}}{\Sigma \vdash e_1\ e_2 : (\theta, \epsilon_1 \sqcup \epsilon_2 \sqcup \epsilon)}$$

$$\frac{\Sigma \vdash e_1 : ((x : \tau\ \mathbf{ref}) \rightarrow \{p\}\theta\ \epsilon\{q\}, \epsilon_1) \qquad \Sigma \vdash x_2 : \tau\ \mathbf{ref} \in \Sigma \qquad x_2 \notin (\theta, \epsilon)}{\Sigma \vdash e_1\ x_2 : (\theta[x \leftarrow x_2], \epsilon_1 \sqcup \epsilon[x \leftarrow x_2])}$$

Alias Prevention

$$\dfrac{\dfrac{\alpha \qquad a{:}\mathbf{int\ ref} \in \Sigma \qquad a \notin (\mathbf{int}, \mathbf{reads}\ x\ y)}{\Sigma \vdash f\ a : ((y{:}\mathbf{int\ ref}) \to ((\mathbf{int}, \mathbf{reads}\ a\ y), \bot), \bot)} \qquad a{:}\mathbf{int\ ref} \in \Sigma \qquad \dfrac{\text{not provable}}{a \notin (\mathbf{int}, \mathbf{reads}\ a\ y)}}{\Sigma \vdash f\ a\ a : (\mathbf{int}, \mathbf{reads}\ a)}$$

$$\alpha : \dfrac{\beta}{\Sigma \vdash f : ((x{:}\mathbf{int\ ref}) \to ((y{:}\mathbf{int\ ref}) \to (\mathbf{int}, \mathbf{reads}\ x\ y), \bot), \bot)}$$

$$\beta : \dfrac{\dfrac{\dfrac{x : \mathbf{int\ ref} \in \Sigma, x{:}\mathbf{int\ ref}, y{:}\mathbf{int\ ref}}{\Sigma, x{:}\mathbf{int\ ref}, y{:}\mathbf{int\ ref} \vdash !x : (\mathbf{int}, \mathbf{reads}\ x)} \qquad \dfrac{y : \mathbf{int\ ref} \in \Sigma, x{:}\mathbf{int\ ref}, y{:}\mathbf{int\ ref}}{\Sigma, x{:}\mathbf{int\ ref}, y{:}\mathbf{int\ ref} \vdash !y : (\mathbf{int}, \mathbf{reads}\ y)}}{\Sigma, x{:}\mathbf{int\ ref}, y{:}\mathbf{int\ ref} \vdash !x+!y : (\mathbf{int}, \mathbf{reads}\ x\ y)}}{\Sigma, x{:}\mathbf{int\ ref} \vdash \mathbf{fun}\ (y{:}\mathbf{int\ ref}) \to !x+!y : ((y{:}\mathbf{int\ ref}) \to (\mathbf{int}, \mathbf{reads}\ x\ y), \bot)}}{\Sigma \vdash \mathbf{fun}\ (x{:}\mathbf{int\ ref}) \to \mathbf{fun}\ (y{:}\mathbf{int\ ref}) \to !x+!y : ((x{:}\mathbf{int\ ref}) \to ((y{:}\mathbf{int\ ref})) \to (\mathbf{int}, \mathbf{reads}\ x\ y), \bot), \bot)}$$

where

- $a{:}\mathbf{int\ ref} \in \Sigma$, and
- $f \equiv \mathbf{fun}\ (x{:}\mathbf{int\ ref}) \to \mathbf{fun}\ (y{:}\mathbf{int\ ref}) \to !x+!y$

Outline

Weakest Precondition

- Program correctness is defined using a calculus of weakest precondition.
- Let $wp(e, q; r)$ denote the weakest precondition for a program expression e and a postcondition $q; r$ where
  - q is the property to hold when terminating normally and
  - $r = E_1 \Rightarrow q_q; \ldots; E_n \Rightarrow q_n$ is the set of properties to hold to each possibly uncaught exception.

Weakest Precondition (cont'd)

Basic Constructs

$$
\begin{aligned}
wp(t, q; r) &= q[result \leftarrow t] \\
wp(\textbf{let } x = e_1 \textbf{ in } e_2, q; r) &= wp(e_1, wp(e_2, q; r)[x \leftarrow result]; r) \\
wp(\textbf{let } x = \textbf{ref } e_1 \textbf{ in } e_2, q; r) &= wp(e_1, wp(e_2, q; r)[x \leftarrow result]; r) \\
wp(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, q; r) &= wp(e_1, \texttt{if } result \texttt{ then } wp(e_2, q; r) \texttt{ else } wp(e_3, q; r); r) \\
wp(L : e, q; r) &= wp(e, q; r)[\textbf{at}(x, L) \leftarrow x]
\end{aligned}
$$

Exceptions and Annotations

$$
\begin{aligned}
wp(\textbf{raise } (E\ e) : \theta, q; r) &= wp(e, r(E); r) \\
wp(\textbf{try } e_1 \textbf{ with } E\ x \rightarrow e_2 \textbf{ end}, q; r) &= wp(e_1, q; E \Rightarrow wp(e_2, q; r)[x \leftarrow result]; r) \\
wp(\textbf{assert } \{p\}; e, q; r) &= p \wedge wp(e, q; r) \\
wp(e\ \{q'; r'\}, q; r) &= wp(e, q' \wedge q; r' \wedge r) \\
wp(e\ \{\{q'; r'\}\}, q; r) &= wp(e, q', r') \wedge \forall \omega. \forall result. q' \Rightarrow q \wedge r' \Rightarrow r
\end{aligned}
$$

Weakest Precondition (cont'd)

Loops

$$wp(\textbf{loop } e\ \{\textbf{invariant } p\ \textbf{variant } t\}, q; r) \quad = \quad p \wedge \forall \omega.\ p \Rightarrow wp(L : e, p \wedge t < \textbf{at}(t, L); r)$$

$$\begin{array}{ll} & wp(\textbf{while } e_1\ \textbf{do } e_2\ \{\textbf{invariant } p\ \textbf{variant } t\}, q; r) \\ = & p \wedge \forall \omega.\ p \Rightarrow wp(L : \textbf{if } e_1\ \textbf{then } e_2\ \textbf{else raise } E, p \wedge t < \textbf{at}(t, L), E \Rightarrow q; r) \\ = & p \wedge \forall \omega.\ p \Rightarrow wp(e_1, \texttt{if } \mathit{result} \texttt{ then } wp(e_2, p \wedge t < \textbf{at}(t, L)) \texttt{ else } q; r)[\textbf{at}(x, L) \leftarrow x] \end{array}$$

where ω stands for the set of references possibly modified by the loop body (the **writes** part of e's effect)

Weakest Precondition (cont'd)

Function Constructs

$$wp(\textbf{fun}\ (x:\theta) \rightarrow \{p\}e, q; r) \quad = \quad q \wedge \forall x.\forall \rho.p \Rightarrow (e, \mathtt{True})$$

$$wp(\textbf{rec}\ f\,(x_1:\theta_1)\ldots(x_n:\theta_n):\theta\{\textbf{variant}\ t\} = \{p\}e, q; r)$$
$$= \quad q \wedge \forall x_1 \ldots \forall x_n.\forall \rho.p \Rightarrow (L:e, \mathtt{True})$$

where

- ρ stands for the set of references possibly accessed by the loop body (the **reads** part of e's effect), and
- $wp(L:e, \mathtt{True})$ must be computed within an environment where f is assumed to have type $(x_1:\theta_1) \rightarrow \cdots \rightarrow (x_n:\theta_n) \rightarrow \{p \wedge t < \textbf{at}(t, L)\}\theta\ \epsilon\{q\}$.

Weakest Precondition (cont'd)

Function Calls

$$wp(x_1\ x_2, q) \quad = \quad p'[x \leftarrow x_2] \wedge \forall \omega. \forall result. (q'[x \leftarrow x_2] \Rightarrow q)[\mathbf{old}(t) \leftarrow t]$$

where $x_1 : (x : \theta) \rightarrow \{p'\}\theta'\ \epsilon\{q'\}$

Assignments

$$\begin{aligned} wp(x := e, q; r) &= wp(e, x = result \Rightarrow q; r) \\ wp(x := t, q) &= (x = t \Rightarrow q) \\ &= q[x \leftarrow t] \end{aligned}$$

Terms

- Rule: $wp(t, q; r) = q[result \leftarrow t]$
- Example:

$$\{x > 0\}\ x - 1\ \{result >= 0\}$$

- Verification Condition:
  - $\forall x : Z, x > 0 \rightarrow x - 1 \geq 0$

$$\begin{aligned} & wp(x - 1, result \geq 0) \\ = \quad & result \geq 0[result \leftarrow x - 1] \\ = \quad & x - 1 \geq 0 \end{aligned}$$

Let Expressions

- Rules:

$$\begin{aligned} wp(\textbf{let } x = e_1 \textbf{ in } e_2, q; r) &= wp(e_1, wp(e_2, q; r)[x \leftarrow result]; r) \\ wp(\textbf{let } x = \textbf{ref } e_1 \textbf{ in } e_2, q; r) &= wp(e_1, wp(e_2, q; r)[x \leftarrow result]; r) \end{aligned}$$

- Example:

$$\texttt{let x = 2 in x * x \{result = 4\}}$$

- Verification Condition:
  - $2 * 2 = 4$

$$\begin{aligned} & wp(\textbf{let } x = 2 \textbf{ in } x * x, result = 4) \\ = \quad & wp(2, wp(x * x, result = 4)[x \leftarrow result]) \\ = \quad & wp(2, (result = 4)[result \leftarrow x * x][x \leftarrow result]) \\ = \quad & wp(2, result * result = 4] \\ = \quad & (result * result = 4)[result \leftarrow 2] \\ = \quad & 2 * 2 = 4 \end{aligned}$$

If-Then-Else

- Rule:

$$
\begin{aligned}
& wp(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, q; r) \\
= \quad & wp(e_1, \texttt{if } result \texttt{ then } wp(e_2, q; r) \texttt{ else } wp(e_3, q; r); r)
\end{aligned}
$$

- Example:

```
if x > y then x else y {result >= x and result >= y}
```

- Verification Conditions:
  - $\forall x : Z. \forall y : Z. \forall H1 : x > y. x \geq x \wedge x \geq y$
  - $\forall x : Z. \forall y : Z. \forall H1 : x \leq y. y \geq x \wedge y \geq y$

$$
\begin{aligned}
& wp(\textbf{if } x > y \textbf{ then } x \textbf{ else } y, result \geq x \wedge result \geq y) \\
= \quad & wp(x > y, \texttt{if } result \texttt{ then } wp(x, result \geq x \wedge result \geq y) \texttt{ else } \\
& wp(y, result \geq x \wedge result \geq y)) \\
= \quad & wp(x > y, \texttt{if } result \texttt{ then } (x \geq x \wedge x \geq y) \texttt{ else } (y \geq x \wedge y \geq y)) \\
= \quad & \texttt{if } (x > y) \texttt{ then } (x \geq x \wedge x \geq y) \texttt{ else } (y \geq x \wedge y \geq y)
\end{aligned}
$$

Exceptions

- Rules:

$$\begin{array}{rcl} wp(\textbf{raise}\ (E\ e) : \theta, q; r) & = & wp(e, r(E); r) \\ wp(\textbf{try}\ e_1\ \textbf{with}\ E\ x \rightarrow e_2\ \textbf{end}, q; r) & = & wp(e_1, q; E \Rightarrow wp(e_2, q; r)[x \leftarrow result]; r) \end{array}$$

- Example:

```
try
  if x > 0 then y / x else raise dbz
with dbz -> 0
end
{result >= 0}
```

- Verification Conditions:
  - $\forall x : Z.\forall y : Z.\forall H1 : x > 0.y/x \geq 0$
  - $\forall x : Z.\forall H1 : x \leq 0.0 \geq 0$

$$\begin{array}{rl} & wp(\textbf{try if}\ x > 0\ \textbf{then}\ y/x\ \textbf{else raise}\ dbz\ \textbf{with}\ dbz \rightarrow 0\ \textbf{end}, result \geq 0) \\ = & wp(\textbf{if}\ x > 0\ \textbf{then}\ y/x\ \textbf{else raise}\ dbz, result \geq 0; dbz \Rightarrow wp(0, result \geq 0)) \\ = & wp(\textbf{if}\ x > 0\ \textbf{then}\ y/x\ \textbf{else raise}\ dbz, result \geq 0; dbz \Rightarrow 0 \geq 0) \\ & (\text{let}\ r = dbz \Rightarrow 0 \geq 0) \\ = & wp(x > 0, \text{if}\ result\ \text{then}\ wp(y/x, result \geq 0; r)\ \text{else}\ wp(\textbf{raise}\ dbz, result \geq 0; r); r) \\ = & wp(x > 0, \text{if}\ result\ \text{then}\ y/x \geq 0\ \text{else}\ 0 \geq 0; r) \\ = & \text{if}\ x > 0\ \text{then}\ y/x \geq 0\ \text{else}\ 0 \geq 0 \end{array}$$

Assignments

- Rules:

$$\begin{aligned} wp(x\texttt{:=}e, q; r) &= wp(e, x = result \Rightarrow q; r) \\ wp(x\texttt{:=}t, q) &= q[x \leftarrow t] \end{aligned}$$

- Example:

```
let _ = x := !x + 1 in
let _ = x := !x + 1 in
  !x
  {result > 0}
```

- Verification Condition:
  - $\forall x : Z.(x+1)+x > 0$

$$\begin{aligned} & wp(\textbf{let } \_ = x \texttt{ := } !x+1 \textbf{ in let } \_ = x \texttt{ := } !x+1 \text{ in } !x, result > 0) \\ = \; & wp(x \texttt{ := } !x+1, wp(\textbf{let } \_ = x \texttt{ :=} !x+1 \text{ in } !x, result > 0)) \\ = \; & wp(x \texttt{ := } !x+1, wp(x \texttt{ := } !x+1, wp(!x, result > 0))) \\ = \; & wp(x \texttt{ := } !x+1, wp(x \texttt{ := } !x+1, x > 0)) \\ = \; & wp(x \texttt{ := } !x+1, x+1 > 0) \\ = \; & (x+1)+1 > 0 \end{aligned}$$

Loops

- Rule:

$$
\begin{array}{ll}
 & wp(\textbf{loop } e \ \{\textbf{invariant } p \ \textbf{variant } t\}, q; r) \\
= & p \wedge \forall \omega.\ p \Rightarrow wp(L : e, p \wedge t < \textbf{at}(t, L); r) \\
 & wp(\textbf{while } e_1 \textbf{ do } e_2 \ \{\textbf{invariant } p \ \textbf{variant } t\}, q; r) \\
= & p \wedge \forall \omega.\ p \Rightarrow wp(e_1, \texttt{if } result \texttt{ then } wp(e_2, p \wedge t < \textbf{at}(t, L)) \texttt{ else } q; r)[\textbf{at}(x, L) \leftarrow x]
\end{array}
$$

- Example:

```
{x > 0} while !x > 0 do {invariant x >= 0 variant x}
  x := !x - 1
done
```

- Verification Conditions:
  - The invariant holds initially.

$$\forall x : Z.\forall H1 : x > 0.x \geq 0$$

  - The invariant is preserved after each iteration.

$$\forall x : Z.\forall H1 : x > 0.\forall x : Z.\forall H2 : x \geq 0.\forall H3 : x > 0.x - 1 \geq 0$$

  - The loop must terminate.

$$\forall x : Z.\forall H1 : x > 0.\forall x : Z.\forall H2 : x \geq 0.\forall H3 : x > 0.x - 1 < x$$

Loops (cont'd)

- Rule:

$$
\begin{array}{ll}
 & wp(\textbf{loop}\ e\ \{\textbf{invariant}\ p\ \textbf{variant}\ t\}, q; r) \\
= & p \wedge \forall \omega.\ p \Rightarrow wp(L : e, p \wedge t < \textbf{at}(t, L); r) \\
 & wp(\textbf{while}\ e_1\ \textbf{do}\ e_2\ \{\textbf{invariant}\ p\ \textbf{variant}\ t\}, q; r) \\
= & p \wedge \forall \omega.\ p \Rightarrow wp(e_1, \texttt{if}\ \mathit{result}\ \texttt{then}\ wp(e_2, p \wedge t < \textbf{at}(t, L))\ \texttt{else}\ q; r)[\textbf{at}(x, L) \leftarrow x]
\end{array}
$$

- Example:

```
{x > 0} while !x > 0 do {invariant x >= 0 variant x}
  x := !x - 1
done
```

- Weakest precondition calculation:

$$
\begin{array}{ll}
 & wp(\textbf{while}\ !x > 0\ \textbf{do}\ x\ \texttt{:=}\ !x - 1\ \{\textbf{invariant}\ x \geq 0\ \textbf{variant}\ x\}, \texttt{True}) \\
= & x \geq 0 \wedge \forall x.x \geq 0 \Rightarrow wp(!x > 0, \texttt{if}\ \mathit{result}\ \texttt{then} \\
 & wp(x\ \texttt{:=}\ !x - 1, x \geq 0 \wedge x < \textbf{at}(x, L))\ \texttt{else True})[\textbf{at}(x, L) \leftarrow x] \\
= & x \geq 0 \wedge \forall x.x \geq 0 \Rightarrow wp(!x > 0, \texttt{if}\ \mathit{result}\ \texttt{then} \\
 & x - 1 > 0 \wedge x - 1 < \textbf{at}(x, L)\ \texttt{else True})[\textbf{at}(x, L) \leftarrow x] \\
= & x \geq 0 \wedge \forall x.x \geq 0 \Rightarrow (\texttt{if}\ x > 0\ \texttt{then} \\
 & x - 1 \geq 0 \wedge x - 1 < \textbf{at}(x, L)\ \texttt{else True})[\textbf{at}(x, L) \leftarrow x] \\
= & x \geq 0 \wedge \forall x.x \geq 0 \Rightarrow (\texttt{if}\ x > 0\ \texttt{then}\ x - 1 \geq 0 \wedge x - 1 < x\ \texttt{else True})
\end{array}
$$

Functions and Function Calls

- Rules:
  - $wp(x_1\ x_2, q) = p'[x \leftarrow x_2] \wedge \forall\omega.\forall result.(q'[x \leftarrow x_2] \Rightarrow q)[\mathbf{old}(t) \leftarrow t]$
    where $x_1 : (x : \theta) \rightarrow \{p'\}\theta'\ \epsilon\{q'\}$
  - $wp(\mathbf{fun}\ (x : \theta) \rightarrow \{p\}e, q; r) = q \wedge \forall x.\forall\rho.p \Rightarrow (e, \mathrm{True})$
  - $wp(\mathbf{rec}\ f\ (x_1 : \theta_1)\ldots(x_n : \theta_n) : \theta\ \{\mathbf{variant}\ t\} = \{p\}e, q; r) =$
    $q \wedge \forall x_1 \ldots \forall x_n.\forall\rho.p \Rightarrow (L : e, \mathrm{True})$
- Example:

```
rec f (x : int) : int {variant x} =
  {x > 0} if x = 1 then 1 else x * f(x - 1)
```

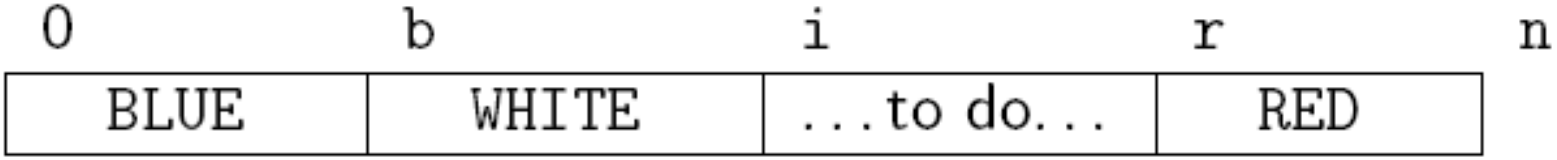
- Verification Conditions:
  - $\forall x : Z.\forall H1 : x > 0.\forall H2 : x \neq 1.x - 1 > 0$
  - $\forall x : Z.\forall H1 : x > 0.\forall H2 : x \neq 1.x - 1 < x$

$$
\begin{aligned}
& wp(\mathbf{rec}\ f(x : \mathbf{int}) : \mathbf{int}\ \{\mathbf{variant}\ x\} = \{x > 0\}\ \mathbf{if}\ x = 1\ \mathbf{then}\ 1\ \mathbf{else}\ x * f(x-1), \mathrm{True}) \\
= \quad & \mathrm{True} \wedge \forall x.x > 0 \Rightarrow wp(L : \mathbf{if}\ x = 1\ \mathbf{then}\ 1\ \mathbf{else}\ x * f(x-1), \mathrm{True}) \\
= \quad & \forall x.x > 0 \Rightarrow wp(\mathbf{if}\ x = 1\ \mathbf{then}\ 1\ \mathbf{else}\ x * f(x-1), \mathrm{True})[\mathbf{at}(x, L) \leftarrow x] \\
= \quad & \forall x.x > 0 \Rightarrow wp(x = 1, \mathrm{if}\ result\ \mathrm{then}\ wp(1, \mathrm{True})\ \mathrm{else}\ wp(x * f(x-1), \mathrm{True}))[\mathbf{at}(x, L) \leftarrow x] \\
= \quad & \forall x.x > 0 \Rightarrow (\mathrm{if}\ x = 1\ \mathrm{then}\ \mathrm{True}\ \mathrm{else} \\
& ((x > 0 \wedge x < \mathbf{at}(x, L))[x \leftarrow x - 1] \wedge \forall result.(\mathrm{True} \Rightarrow \mathrm{True})[\mathbf{old}(t) \leftarrow t]))[\mathbf{at}(x, L) \leftarrow x] \\
= \quad & \forall x.x > 0 \Rightarrow (\mathrm{if}\ x = 1\ \mathrm{then}\ \mathrm{True}\ \mathrm{else}\ (x - 1 > 0 \wedge x - 1 < \mathbf{at}(x, L)))[\mathbf{at}(x, L) \leftarrow x] \\
= \quad & \forall x.x > 0 \Rightarrow (\mathrm{if}\ x = 1\ \mathrm{then}\ \mathrm{True}\ \mathrm{else}\ (x - 1 > 0 \wedge x - 1 < \mathbf{at}(x, L)))[\mathbf{at}(x, L) \leftarrow x] \\
= \quad & \forall x.x > 0 \Rightarrow (\mathrm{if}\ x = 1\ \mathrm{then}\ \mathrm{True}\ \mathrm{else}\ (x - 1 > 0 \wedge x - 1 < x))
\end{aligned}
$$

Outline

Dijkstra's Dutch Flag

Goal: sort an array where elements only have three different values (blue, white and red)



Dutch Flag Example in C

```
typedef enum { BLUE, WHITE, RED } color;

void swap(int t[], int i, int j) {
  color c = t[i];
  t[i] = t[j];
  t[j] = c;
}

void flag(int t[], int n) {
  int b = 0, i = 0, r = n;
  while (i < r) {
    switch (t[i]) {
    case BLUE: swap(t, b++, i++); break;
    case WHITE: i++; break;
    case RED: swap(t, --r, i); break;
    }
  }
}
```

Dutch Flag Example in Why

Colors

```
type color

logic blue : color
logic white : color
logic red : color

predicate is_color (c : color) =
  c = blue or c = white or c = red

parameter eq_color :
  c1 : color -> c2 : color ->
    {}
    bool
    {if result then c1 = c2 else c1 <> c2}
```

Dutch Flag Example in Why (cont'd)

Color Arrays

```
type color_array

logic acc : color_array, int -> color
logic upd : color_array, int, color -> color_array
logic length : color_array -> int

axiom acc_upd_eq :
  forall t : color_array. forall i : int. forall c : color.
    acc(upd(t, i, c), i) = c

axiom acc_upd_neq :
  forall t : color_array. forall i : int. forall j : int.
    forall c : color.
      j <> i -> acc(upd(t, i, c), j) = acc(t, j)

axiom length_upd :
  forall t : color_array. forall i : int. forall c : color.
    length(upd(t, i, c)) = length(t)
```

Dutch Flag Example in Why (cont'd)

Color Arrays (cont'd)

```
parameter get :
  t : color_array ref -> i : int ->
    {0 <= i < length(t)}
    color reads t
    {result = acc(t, i)}

parameter set :
  t : color_array ref -> i : int -> c : color ->
    {0 <= i < length(t)}
    unit writes t
    {t = upd(t@, i, c)}
```

Dutch Flag Example in Why (cont'd)

Swap

```
let swap (t : color_array ref) (i : int) (j : int) =
  {0 <= i < length(t) and 0 <= j < length(t)}
  let c = get t i in
  set t i (get t j); set t j c
  {t = upd(upd(t@, i, acc(t@, j)), j, acc(t@, i))}
```

Dutch Flag Example in Why (cont'd)

Monochrome and Permutation

```
predicate monochrome (t:color_array, i:int, j:int, c:color) =
  forall k : int. i <= k < j -> acc(t, k) = c

logic permutation : color_array, color_array, int, int -> prop

axiom permut_refl :
  forall t:color_array. forall l,r:int. permutation(t, t, l, r)

axiom permut_sym :
  forall t1,t2:color_array. forall l,r:int.
    permutation(t1, t2, l, r) -> permutation(t2, t1, l, r)

axiom permut_trans :
  forall t1,t2,t3:color_array. forall l,r:int.
    permutation(t1, t2, l, r) -> permutation(t2, t3, l, r) ->
      permutation(t1, t3, l, r)

axiom permut_swap :
  forall t:color_array. forall l,r,i,j:int.
    l <= i <= r -> l <= j <= r ->
      permutation(t, upd(upd(t, i, acc(t,j)), j, acc(t,i)), l, r)
```

Dutch Flag Example in Why (cont'd)

Dutch Flag

```
let dutch_flag (t : color_array ref) (n : int) =
  { length(t) = n and
    forall k : int. 0 <= k < n -> is_color(acc(t, k)) }
  let b = ref 0 in
  let i = ref 0 in
  let r = ref n in
  while !i < !r do
    if (eq_color (get t !i) blue) then begin
        swap t !b !i; b := !b + 1; i := !i + 1
    end else if (eq_color (get t !i) white) then
      i := !i + 1
    else begin
        r := !r + 1; swap t !r !i
    end
  done
  { (exists b : int. exists r : int.
       monochrome(t, 0, b, blue) and
       monochrome(t, b, r, white) and
       monochrome(t, r, n, red)) and
    permutation(t, t@, 0, n - 1) }
```

Outline

Frama-C

- Frama-C is an extensible platform dedicated to source-code analysis of C software.
- Equipped with Frama-C, you can:
  - observe sets of possible values for the variables of the program at each point of the execution;
  - slice the original program into simplified ones;
  - navigate the dataflow of the program, from definition to use or from use to definition.
- Plugins available so far:
  - Value analysis
  - Jessie: the deductive verification plug-in
  - Impact analysis
  - Scope & Data-flow browsing
  - Variable occurrence browsing
  - Semantic constant folding
  - Slicing
  - Spare code
  - Aoraï

Frama-C-Jessie

- Jessie is based on weakest precondition computation techniques.
- Specification is expressed in ACSL (ANSI/ISO C Specification Language).
- The annotated C program will be translated to a annotated Why program.
- Frama-C-Jessie subsumes Caduceus.

ACSL - Basic Specification

```
/*@ requires n > 0;
    requires \valid(p+ (0..n-1));
    ensures \forall int i; 0 <= i <= n - 1 ==> \result >= p[i];
    ensures \exists int e; 0 < e <= n - 1 && \result == p[e];
*/
int max_seq(int * p, int n) {
  int res = *p;
  for (int i = 0; i < n; i++) {
    if (res < *p) { res = *p; }
    p++;
  }
  return res;
}
```

Problem: the specification does not prevent the implementation from altering the integer array.

ACSL - Basic Specification (cont'd)

```
/*@ requires n > 0;
    requires \valid(p+ (0..n-1));
    ensures \forall int i; 0 <= i <= n - 1 ==> p[i] == \old(p[i]);
    ensures \forall int i; 0 <= i <= n - 1 ==> \result >= p[i];
    ensures \exists int e; 0 < e <= n - 1 && \result == p[e];
*/
int max_seq(int * p, int n) {
  int res = *p;
  for (int i = 0; i < n; i++) {
    if (res < *p) { res = *p; }
    p++;
  }
  return res;
}
```

The red line ensures that the content of the integer array is not modified.

ACSL - Assigns Clauses

```
/*@ requires \valid(p) && \valid(q);
    assigns *p, *q;
*/
void swap (int *p, int *q) {
  int tmp = *p;
  *p = *q;
  *q = tmp;
  return;
}
```

- When no **assigns** clauses are specified, the function is allowed to modify every visible variable.
- If there are **assigns** clauses specified, the function can only modify the content of the locations that are explicitly mentioned in the clauses.

ACSL - Assigns Clauses (cont'd)

```
/*@ requires n > 0;
    requires \valid(p+ (0..n-1));
    assigns \nothing;
    ensures \forall int i; 0 <= i <= n - 1 ==> \result >= p[i];
    ensures \exists int e; 0 < e <= n - 1 && \result == p[e];
*/
int max_seq(int * p, int n) {
  int res = *p;
  for (int i = 0; i < n; i++) {
    if (res < *p) { res = *p; }
    p++;
  }
  return res;
}
```

The red line ensures that the integer array p and the integer n are not modified.

ACSL - Termination

```
/*@ assigns \nothing
    terminates c > 0;
*/
void f (int c) {
  while(!c)
    ;
  return;
}
```

The function f can be called with any argument c, but the function is not guaranteed to terminate if c <= 0.

ACSL - Behaviors

```
typedef enum { Max, Min } kind;

/*@ requires k == Max || k == Min;
    assigns \nothing;
    ensures \result == x || \result == y;
    ensures k == Max ==> \result >= x && \result >= y;
    ensures k == Min ==> \result <= x && \result <= y;
*/
int extremum (kind k, int x, int y) {
  return ((k == Max ? x > y : x < y) ? x : y);
}
```

This specification may not be clear to say that `extremum` has two distinct modes of operation.

ACSL - Behaviors (cont'd)

```
/*@ requires k == Max || k == Min;
    assigns \nothing;
    ensures \result == x || \result == y;
    behavior is_max:
      assumes k == Max;
      ensures \result >= x && \result >= y;
    behavior is_min:
      assumes k == Min;
      ensures \result <= x && \result <= y;
    complete behaviors is_max, is_min;
    disjoint behaviors is_max, is_min;
*/
int extremum (kind k, int x, int y);
```

ACSL - Predicates and Logic Functions

```
typedef struct _list {
  int element;
  struct _list* next;
} list;

/*@ inductive reachable{L} (list* root, list* node) {
      case root_reachable:
        \forall list* root; reachable(root, root);
      case next_reachable:
        \forall list* root, *node;
          \valid(root) ==> reachable(root->next, node) ==>
            reachable(root, node);
    }
*/

/*@ predicate finite{L} (list* node) =
      reachable(root, \null);
*/
```

ACSL - Predicates and Logic Functions (cont'd)

```
/*@ axiomatic Length {
      logic integer length{L} (list* l);

      axiom length_nil{L} : length(\null) == 0;

      axiom length_cons{L} :
        \forall list* l, integer n;
          finite(l) && \valid(l) ==>
            length(l) == length(l->next) + 1;
    }
*/
```

ACSL - Predicates and Logic Functions (cont'd)

```
/*@ requires \valid(root);
    assigns \nothing;
    terminates finite(root);
    ensures
      \forall list* l;
        \valid(l) && reachable(root, l) ==>
          \result >= l->element;
    ensures
      \exists list* l;
        \valid(l) && reachable(root, l) &&
          \result == l->element;
*/
int max_list (list* root) {
  int max = root->element;
  while (root->next) {
    root = root->next;
    if (root->element > max) max = root->element;
  }
}
```

ACSL - Data Invariants

Type Invariants

```
/*@ type invariant finite_list(list* root) = finite(root); */
```

With this type invariant, we can get rid of the `terminates` clause in the specification of `max_list`.

Global Invariants

```
list* glist;
/*@ global invariant glist_sorted : sorted(glist); */
```

The global invariant ensures that glist is always sorted

- at each entrance or exit of a function (for weak invariants), or
- at each point of the program (for strong invariants).

ACSL - Assertions

```
/*@ requires n >= 0 && n < 100;
*/
int f (int n) {
  int tmp = 100 - n;
  //@ assert tmp > 0;
  //@ assert tmp < 100;
  return tmp;
}
```

ACSL - Loop Invariants

```
int max_seq (int* p, int n) {
  int res = *p;
  /*@ loop invariant
      \forall integer j; 0 <= j <= i ==> res >= *(p + j);
  */
  for (int i = 0; i < n; i++) {
    if (res < *p) { res = *p; }
    p++;
  }
  return res;
}
```

ACSL - Ghost Variables

```
int max_seq (int* p, int n) {
  int res = *p;
  /*@ ghost int e = 0;
      loop invariant \forall int j;
                        0 <= j <= i ==> res >= p[j];
      loop invariant \forall \valid(p+e) && p[e] == res;
  */
  for (int i = 0; i < n; i++) {
    if (res < *p) {
      res = *p;
      //@ ghost e = i;
    }
    p++;
  }
  return res;
}
```

References

- J.-C. Filliâtre. *The WHY Verification Tool - Tutorial and Reference Manual*.
- J.-C. Filliâtre. *Verification of Non-Functional Programs using Interpretations in Type Theory*. Journal of Functional Programming 13(4):709-745, July 2003.
- J.-C. Filliâtre. *Why: an Intermediate Language for Program Verification*. TYPES Summer School 2007.
- V. Prevosto. *ACSL Mini-Tutorial*.