

# Why

## A Software Verification Tool

Ming-Hsien Tsai

SVRL  
Dept. of Information Management  
National Taiwan University

12/15/2011

# Outline

- 1 Introduction
- 2 Why Language
  - Logic
  - Program
  - Weakest Precondition
- 3 Dijkstra's Dutch Flag
- 4 References

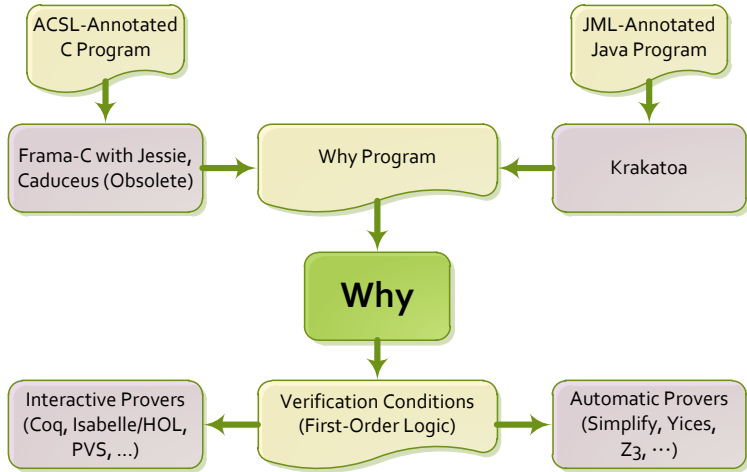
# What is Why<sup>1</sup>

- 🌐 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**

---

<sup>1</sup>The version considered here is Why2.

# Verification with Why



# 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
  - ☀ Z3, Yices, Simplify, CVC3, Alt-Ergo, etc.
- 🌐 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 programs.

# Demo

# Verification Condition Generation - 1

## Annotated Program

 $\{P_1\}$  $S_1$  $\vdots$  $S_i$  $\{\{P_2\}\}$  $S_{i+1}$  $\vdots$  $S_n$  $\{P_3\}$ 

## Verification Conditions

# Verification Condition Generation - 1

## Annotated Program

 $\{P_1\}$  $S_1$  $\vdots$  $S_i$  $\{\{P_2\}\}$  $S_{i+1}$  $\vdots$  $\{Q_1\} (Q_1 = wp(S_n, P_3))$  $S_n$  $\{P_3\}$ 

## Verification Conditions



# Verification Condition Generation - 1

## Annotated Program

 $\{P_1\}$  $S_1$  $\vdots$  $S_i$  $\{\{P_2\}\}$  $S_{i+1}$  $\{Q_2\}$  $\vdots$  $\{Q_1\} (Q_1 = wp(S_n, P_3))$  $S_n$  $\{P_3\}$ 

## Verification Conditions

# Verification Condition Generation - 1

## Annotated Program

 $\{P_1\}$  $S_1$  $\vdots$  $S_i$  $\{\{P_2\}\}$  $\{Q_3\}$  $S_{i+1}$  $\{Q_2\}$  $\vdots$  $\{Q_1\} (Q_1 = wp(S_n, P_3))$  $S_n$  $\{P_3\}$ 

## Verification Conditions

# Verification Condition Generation - 1

## Annotated Program

 $\{P_1\}$  $S_1$  $\vdots$  $S_i$  $\{\{P_2\}\}$  $\{Q_3\}$  $S_{i+1}$  $\{Q_2\}$  $\vdots$  $\{Q_1\} (Q_1 = wp(S_n, P_3))$  $S_n$  $\{P_3\}$ 

## Verification Conditions

 $\forall V. P_2 \Rightarrow Q_3$

# Verification Condition Generation - 1

## Annotated Program

 $\{P_1\}$  $S_1$  $\vdots$  $\{Q_4\} (Q_4 = WP(S_i, P_2))$  $S_i$  $\{\{P_2\}\}$  $\{Q_3\}$  $S_{i+1}$  $\{Q_2\}$  $\vdots$  $\{Q_1\} (Q_1 = wp(S_n, P_3))$  $S_n$  $\{P_3\}$ 

## Verification Conditions

 $\forall V. P_2 \Rightarrow Q_3$

# Verification Condition Generation - 1

## Annotated Program

 $\{P_1\}$  $S_1$  $\{Q_5\}$  $\vdots$  $\{Q_4\} (Q_4 = WP(S_i, P_2))$  $S_i$  $\{\{P_2\}\}$  $\{Q_3\}$  $S_{i+1}$  $\{Q_2\}$  $\vdots$  $\{Q_1\} (Q_1 = wp(S_n, P_3))$  $S_n$  $\{P_3\}$ 

## Verification Conditions

 $\forall V. P_2 \Rightarrow Q_3$

# Verification Condition Generation - 1

## Annotated Program

```

{P1}
{Q6}
S1
{Q5}
⋮
{Q4} (Q4 = WP(Si, P2))
Si
{{P2}}
{Q3}
Si+1
{Q2}
⋮
{Q1} (Q1 = wp(Sn, P3))
Sn
{P3}

```

## Verification Conditions

$$\forall V. P_2 \Rightarrow Q_3$$

# Verification Condition Generation - 1

## Annotated Program

```

{P1}
{Q6}
S1
{Q5}
⋮
{Q4} (Q4 = WP(Si, P2))
Si
{{P2}}
{Q3}
Si+1
{Q2}
⋮
{Q1} (Q1 = wp(Sn, P3))
Sn
{P3}

```

## Verification Conditions

$$\forall V. P_2 \Rightarrow Q_3$$

$$\forall V. P_1 \Rightarrow Q_6$$

# Verification Condition Generation - 2

## Annotated Program

 $\{P_1\}$  $S_1$  $\vdots$  $S_i$  $\{P_2\}$  $S_{i+1}$  $\vdots$  $S_n$  $\{P_3\}$ 

## Verification Conditions



# Verification Condition Generation - 2

## Annotated Program

 $\{P_1\}$  $S_1$  $\vdots$  $S_i$  $\{P_2\}$  $S_{i+1}$  $\vdots$  $\{Q_1\} (Q_1 = wp(S_n, P_3))$  $S_n$  $\{P_3\}$ 

## Verification Conditions

# Verification Condition Generation - 2

## Annotated Program

 $\{P_1\}$  $S_1$  $\vdots$  $S_j$  $\{P_2\}$  $S_{i+1}$  $\{Q_2\}$  $\vdots$  $\{Q_1\} (Q_1 = wp(S_n, P_3))$  $S_n$  $\{P_3\}$ 

## Verification Conditions

# Verification Condition Generation - 2

## Annotated Program

 $\{P_1\}$  $S_1$  $\vdots$  $S_i$  $\{P_2\}$  $\{Q_3\}$  $S_{i+1}$  $\{Q_2\}$  $\vdots$  $\{Q_1\}$  ( $Q_1 = wp(S_n, P_3)$ ) $S_n$  $\{P_3\}$ 

## Verification Conditions

# Verification Condition Generation - 2

## Annotated Program

 $\{P_1\}$  $S_1$  $\vdots$  $\{Q_4\} (Q_4 = wp(S_i, P_2 \wedge Q_3))$  $S_i$  $\{P_2\}$  $\{Q_3\}$  $S_{i+1}$  $\{Q_2\}$  $\vdots$  $\{Q_1\} (Q_1 = wp(S_n, P_3))$  $S_n$  $\{P_3\}$ 

## Verification Conditions

# Verification Condition Generation - 2

## Annotated Program

 $\{P_1\}$  $S_1$  $\{Q_5\}$  $\vdots$  $\{Q_4\} (Q_4 = wp(S_i, P_2 \wedge Q_3))$  $S_i$  $\{P_2\}$  $\{Q_3\}$  $S_{i+1}$  $\{Q_2\}$  $\vdots$  $\{Q_1\} (Q_1 = wp(S_n, P_3))$  $S_n$  $\{P_3\}$ 

## Verification Conditions

# Verification Condition Generation - 2

## Annotated Program

```
{P1}  
{Q6}  
S1  
{Q5}  
:  
{Q4} (Q4 = wp(Si, P2 ∧ Q3))  
Si  
{P2}  
{Q3}  
Si+1  
{Q2}  
:  
{Q1} (Q1 = wp(Sn, P3))  
Sn  
{P3}
```

## Verification Conditions

# Verification Condition Generation - 2

## Annotated Program

```
{P1}  
{Q6}  
S1  
{Q5}  
⋮  
{Q4} (Q4 = wp(Si, P2 ∧ Q3))  
Si  
{P2}  
{Q3}  
Si+1  
{Q2}  
⋮  
{Q1} (Q1 = wp(Sn, P3))  
Sn  
{P3}
```

## Verification Conditions

$$\forall V. P_1 \Rightarrow Q_6$$

# Outline

## 1 Introduction

## 2 Why Language

- Logic
- Program
- Weakest Precondition

## 3 Dijkstra's Dutch Flag

## 4 References



# Why Language

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

## Example

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

# Outline

- 1 Introduction
- 2 Why Language
  - Logic
  - Program
  - Weakest Precondition
- 3 Dijkstra's Dutch Flag
- 4 References

# Logic Syntax

$\langle term \rangle ::=$

- $\langle constant \rangle$
- $\langle term \rangle \langle arith\_op \rangle \langle term \rangle$
- $- \langle term \rangle$
- $\langle lab\_identifier \rangle$
- $\langle identifier \rangle ( \langle term \rangle^+ )$
- $\langle lab\_identifier \rangle [ \langle term \rangle ]$
- if**  $\langle term \rangle$  **then**  $\langle term \rangle$  **else**  $\langle term \rangle$
- let**  $\langle identifier \rangle = \langle term \rangle$  **in**  $\langle term \rangle$
- $( \langle term \rangle )$

$\langle constant \rangle ::=$

- $\langle integer\_constant \rangle$
- $\langle real\_constant \rangle$
- true**
- false**
- void**

$\langle arith\_op \rangle ::= + \mid - \mid * \mid / \mid \%$

# Logic Syntax (cont'd)

```

<predicate> ::= true
              | false
              | <identifier>
              | <identifier> ( <term>+ )
              | <term> <relation> <term> [ <relation> <term> ]
              | <predicate> -> <predicate>
              | <predicate> <-> <predicate>
              | <predicate> or <predicate>
              | <predicate> and <predicate>
              | not <predicate>
              | if <term> then <predicate> else <predicate>
              | let <identifier> = <term> in <predicate>
              | forall <identifier>+ : <primitive_type> [ <triggers> ] . <predicate>
              | exists <identifier>+ : <primitive_type> [ <triggers> ] . <predicate>
              | ( <predicate> )
              | ( <identifier> | <string> ) : <predicate>

<triggers> ::= [ <trigger>+ ]
<trigger>  ::= <term>+

<primitive_type> ::= int | bool | real | unit | <identifier> | ' <identifier>
                  | <primitive_type> <identifier> | ( <primitive_type>* ) <identifier>

<relation> ::= = | <> | < | <= | > | >=
  
```

# Logic Syntax (cont'd)

```

⟨I_declaration⟩ ::= [ external ] type [ ⟨type_parameters⟩ ] ⟨identifier⟩
                  | [ external ] logic ⟨identifier⟩+ : ⟨logic_type⟩
                  | function ⟨identifier⟩ ( ⟨logic_binder⟩* ) : ⟨primitive_type⟩
                  = ⟨term⟩
                  | predicate ⟨identifier⟩ ( ⟨logic_binder⟩* ) = ⟨predicate⟩
                  | inductive ⟨identifier⟩ : ⟨logic_type⟩ =
                    ( | ⟨identifier⟩ : ⟨predicate⟩ )*
                  | axiom ⟨identifier⟩ : ⟨predicate⟩
                  | goal ⟨identifier⟩ : ⟨predicate⟩

⟨logic_type⟩ ::= ⟨logic_arg_type⟩* -> prop
               | ⟨logic_arg_type⟩* -> ⟨primitive_type⟩

⟨logic_arg_type⟩ ::= ⟨primitive_type⟩ | ⟨primitive_type⟩ array

⟨logic_binder⟩ ::= ⟨identifier⟩ : ⟨primitive_type⟩

⟨type_parameters⟩ ::= ' ⟨identifier⟩ | ( ( ' ⟨identifier⟩ )+ )

```

## Examples

### Example (Type Declaration)

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

### Example (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  $\rightarrow$  int  
logic insert : 'a, 'a list  $\rightarrow$  '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

$$\tau ::= \alpha \mid (\tau, \dots, \tau) s$$

$$t ::= x \mid f(t, \dots, t)$$

$$P ::= p(t, \dots, t)$$

$$\mid \top \mid \perp \mid P \wedge P \mid P \vee P \mid \neg P \mid P \Rightarrow P$$

$$\mid \forall x : \tau. P \mid \exists x : \tau. P$$

## A theory $\Sigma$ is a finite list of declarations $\delta$ where

$$\delta ::= \mathbf{type} \vec{\alpha} s$$

$$\mid x : \tau$$

$$\mid \mathbf{logic} f : \forall \vec{\alpha}. \tau, \dots, \tau \rightarrow \tau$$

$$\mid \mathbf{logic} p : \forall \vec{\alpha}. \tau, \dots, \tau \rightarrow \mathbf{prop}$$

$$\mid \mathbf{axiom} \forall \vec{\alpha}. P$$

$$\mid \mathbf{goal} \forall \vec{\alpha}. P$$

# Well-Formed Types ( $\Sigma \vdash \tau \text{ wf}$ )

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

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

## Example

```
color
int list
color list
```



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

$$\frac{x : \tau \in \Sigma}{\Sigma \vdash x : \tau} T_1$$

$$\frac{\mathbf{logic} \ f : \forall \vec{\alpha}. \tau_1, \dots, \tau_n \rightarrow \tau \in \Sigma \quad \text{Subst}(\sigma, \Sigma) \quad \forall i, \Sigma \vdash t_i : \sigma(\tau_i)}{\Sigma \vdash f(t_1, \dots, t_n) : \sigma(\tau)} T_2$$

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

## Example

```
logic length : 'a list  $\rightarrow$  int
logic insert : 'a, 'a list  $\rightarrow$  'a list
length(x) : int
insert(0, x) : int list
```

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

$$\begin{array}{c}
 \text{logic } p : \forall \vec{\alpha}. \tau_1, \dots, \tau_n \rightarrow \text{prop} \in \Sigma \quad \text{Subst}(\sigma, \Sigma) \quad \forall i, \Sigma \vdash t_i : \sigma(\tau_i) \\
 \hline
 \Sigma \vdash p(t_1, \dots, t_n) \text{ wf} \quad P_1
 \end{array}$$

$$\begin{array}{c}
 \Sigma \vdash \top \text{ wf} \quad P_2 \quad \Sigma \vdash \perp \text{ wf} \quad P_3
 \end{array}$$

$$\begin{array}{c}
 \Sigma \vdash P_1 \text{ wf} \quad \Sigma \vdash P_2 \text{ wf} \quad P_4 \quad \Sigma \vdash P_1 \text{ wf} \quad \Sigma \vdash P_2 \text{ wf} \\
 \hline
 \Sigma \vdash P_1 \wedge P_2 \text{ wf} \quad \Sigma \vdash P_1 \vee P_2 \text{ wf} \quad P_5
 \end{array}$$

$$\begin{array}{c}
 \Sigma \vdash P \text{ wf} \quad P_6 \quad \Sigma \vdash P_1 \text{ wf} \quad \Sigma \vdash P_2 \text{ wf} \\
 \hline
 \Sigma \vdash \neg P \text{ wf} \quad \Sigma \vdash P_1 \Rightarrow P_2 \text{ wf} \quad P_7
 \end{array}$$

$$\begin{array}{c}
 \Sigma, x : \tau \vdash P \text{ wf} \quad P_8 \quad \Sigma, x : \tau \vdash P \text{ wf} \\
 \hline
 \Sigma \vdash \forall x : \tau. P \text{ wf} \quad \Sigma \vdash \exists x : \tau. P \text{ wf} \quad P_8
 \end{array}$$

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

$$\begin{array}{c}
\frac{}{\vdash \text{wf}} \text{Th}_1 \quad \frac{\text{type } s \notin \Sigma}{\vdash \Sigma, \text{type } (\alpha_1, \dots, \alpha_2) s \text{ wf}} \text{Th}_2 \quad \frac{x \notin \Sigma \quad \Sigma \vdash \tau \text{ wf}}{\vdash \Sigma, x : \tau \text{ wf}} \text{Th}_3 \\
\\
\frac{\text{logic } f \notin \Sigma \quad \forall i, \Sigma \vdash \tau_i \text{ wf}}{\vdash \Sigma, \text{logic } f : \forall \vec{\alpha}. \tau_1, \dots, \tau_n \rightarrow \tau_{n+1} \text{ wf}} \text{Th}_4 \\
\\
\frac{\text{logic } p \notin \Sigma \quad \forall i, \Sigma \vdash \tau_i \text{ wf}}{\vdash \Sigma, \text{logic } p : \forall \vec{\alpha}. \tau_1, \dots, \tau_n \rightarrow \text{prop wf}} \text{Th}_5 \\
\\
\frac{}{\vdash \Sigma, \text{axiom } \forall \vec{\alpha}. P \text{ wf}} \text{Th}_6 \quad \frac{}{\vdash \Sigma, \text{goal } \forall \vec{\alpha}. P \text{ wf}} \text{Th}_6
\end{array}$$

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

## Deduction Rules

$$\begin{array}{c}
 \frac{\mathbf{axiom} \quad \forall \vec{\alpha}. P \in \Sigma \quad \text{Subst}(\sigma, \Sigma)}{\Sigma \models \sigma(P)} \text{Ax} \quad \frac{\Sigma \models Q \quad \Sigma, Q \models P}{\Sigma \models \sigma(P)} \text{Cut} \\
 \\
 \frac{}{\Sigma \models \top} \text{True} \quad \frac{\Sigma \models \perp \quad \Sigma \vdash P \text{ wf}}{\Sigma \models P} \text{False} \quad \frac{\Sigma \vdash P \text{ wf}}{\Sigma \models P \vee \neg P} \text{EM}
 \end{array}$$

and many other rules not listed here

# Outline

- 1 Introduction
- 2 Why Language
  - Logic
  - Program
  - Weakest Precondition
- 3 Dijkstra's Dutch Flag
- 4 References

# Program Syntax

$\langle \text{simple\_value\_type} \rangle ::= \langle \text{primitive\_type} \rangle \mid \langle \text{primitive\_type} \rangle \text{ ref}$   
 $\mid \langle \text{primitive\_type} \rangle \text{ array} \mid ( \langle \text{value\_type} \rangle )$

$\langle \text{value\_type} \rangle ::= \langle \text{simple\_value\_type} \rangle$   
 $\mid \langle \text{simple\_value\_type} \rangle \rightarrow \langle \text{computation\_type} \rangle$   
 $\mid \langle \text{identifier} \rangle : \langle \text{simple\_value\_type} \rangle \rightarrow \langle \text{computation\_type} \rangle$

$\langle \text{computation\_type} \rangle ::= \{ [ \langle \text{precondition} \rangle ] \}$   
 $[ \text{returns } \langle \text{identifier} \rangle \text{ key: } ] \langle \text{value\_type} \rangle \langle \text{effects} \rangle$   
 $\{ [ \langle \text{postcondition} \rangle ] \}$   
 $\mid \langle \text{value\_type} \rangle$

$\langle \text{effects} \rangle ::= [ \text{reads } \langle \text{identifier} \rangle^* ]$   
 $[ \text{writes } \langle \text{identifier} \rangle^* ]$   
 $[ \text{raises } \langle \text{identifier} \rangle^* ]$

$\langle \text{precondition} \rangle ::= \langle \text{assertion} \rangle$

$\langle \text{postcondition} \rangle ::= \langle \text{assertion} \rangle \langle \text{exn\_condition} \rangle^*$

$\langle \text{exn\_condition} \rangle ::= \mid \langle \text{identifier} \rangle \Rightarrow \langle \text{assertion} \rangle$

$\langle \text{assertion} \rangle ::= \langle \text{predicate} \rangle [ \text{as } \langle \text{identifier} \rangle ]$

# Program Syntax (cont'd)

```

⟨prog⟩ ::=
  ⟨constant⟩ | ⟨identifier⟩ | ! ⟨identifier⟩
  | ⟨identifier⟩ := ⟨prog⟩
  | ⟨identifier⟩ [ ⟨prog⟩ ]
  | ⟨identifier⟩ [ ⟨prog⟩ ] := ⟨prog⟩
  | ⟨prog⟩ ⟨infix⟩ ⟨prog⟩ | ⟨prefix⟩ ⟨prog⟩
  | let ⟨identifier⟩ = ⟨prog⟩ in ⟨prog⟩
  | let ⟨identifier⟩ = ref ⟨prog⟩ in ⟨prog⟩
  | if ⟨prog⟩ then ⟨prog⟩ [ else ⟨prog⟩ ]
  | while ⟨prog⟩ do [ ⟨loop_annot⟩ ] ⟨prog⟩ done
  | ⟨prog⟩ ; ⟨prog⟩
  | ⟨identifier⟩ : ⟨prog⟩
  | assert ( { ⟨assertion⟩ } )+ ; ⟨prog⟩
  | ⟨prog⟩ { ⟨postcondition⟩ } | ⟨prog⟩ { { ⟨postcondition⟩ } }
  | fun ⟨binders⟩ -> ⟨prog⟩
  | let ⟨identifier⟩ ⟨binders⟩ = ⟨prog⟩ in ⟨prog⟩
  | let rec ⟨recfun⟩ [ in ⟨prog⟩ ]
  | ⟨prog⟩ ⟨prog⟩
  | raise ⟨identifier⟩ [ : ⟨value_type⟩ ]
  | raise ( ⟨identifier⟩ ⟨prog⟩ ) [ : ⟨value_type⟩ ]
  | try ⟨prog⟩ with ⟨handler⟩1+ end
  | absurd [ : ⟨value_type⟩ ]
  | ( ⟨prog⟩ )
  | begin ⟨prog⟩ end

```

# Program Syntax (cont'd)

$\langle infix \rangle ::= + \mid - \mid * \mid / \mid \% \mid = \mid \langle \rangle \mid < \mid \leq \mid > \mid \geq \mid || \mid \&\&$   
 $\langle prefix \rangle ::= - \mid \text{not}$

$\langle binders \rangle ::= ( \langle identifier \rangle^+ : \langle value\_type \rangle ) +$

$\langle recfun \rangle ::= \langle identifier \rangle \langle binders \rangle : \langle value\_type \rangle$   
 $\{ \text{variant } \langle wf\_arg \rangle \} = \langle prog \rangle$

$\langle loop\_annot \rangle ::= \{ [ \text{invariant } \langle assertion \rangle ] [ \text{variant } \langle wf\_arg \rangle ] \}$

$\langle wf\_arg \rangle ::= \langle term \rangle [ \text{for } \langle identifier \rangle ]$

$\langle handler \rangle ::= \langle identifier \rangle \rightarrow \langle prog \rangle$   
 $\mid \langle identifier \rangle \langle identifier \rangle \rightarrow \langle prog \rangle$



# Program Expressions

$t ::= x \mid !x \mid f(t, \dots, t)$

$e ::= t$

| **let**  $x = e$  **in**  $e$

| **let**  $x = \text{ref } e$  **in**  $e$

| **if**  $e$  **then**  $e$  **else**  $e$

| **loop**  $e$  {**invariant**  $p$  **variant**  $t$ }

|  $L : e$

| **raise**  $(E e) : \theta$

| **try**  $e$  **with**  $E x \rightarrow e$  **end**

| **assert**  $\{p\}; e$

|  $e \{q\}$

|  $e \{\{q\}\}$

| **fun**  $(x : \theta) \rightarrow \{p\} e$

| **rec**  $x (x : \theta) \dots (x : \theta) : \theta$  {**variant**  $t$ } =  $\{p\} e$

|  $e e$

$$p ::= \emptyset \mid d \ p$$
$$d ::= \mathbf{let} \ x = e$$
$$| \ \mathbf{val} \ x : \theta$$
$$| \ \mathbf{exception} \ E \ \text{of} \ \tau$$

# Syntactic Sugar

```

e1; e2      ≡  let _ = e1 in e2
x := e        ≡  ref_set x e
raise E      ≡  raise (E void) : unit
while e1 do e2 {invariant p variant t} ≡
try
  loop if e1 then e2 else raise Exit
  {invariant p variant t}
with Exit _ -> void end

```

## 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 }

```

# Program Types and Specifications

🌐 Program types:  $\theta$

🌐 Specifications:  $\kappa$

🌐 Postconditions:  $q$

🌐 Effects:  $\epsilon$

$$\theta ::= \tau^2 \mid \tau \mathbf{ref} \mid (x : \theta)^+ \rightarrow \kappa$$
$$\kappa ::= \{p\}\theta \in \{q\}$$
$$q ::= p; E \Rightarrow p; \dots; E \Rightarrow p$$
$$\epsilon ::= \mathbf{reads} \ x, \dots, x \ \mathbf{writes} \ x, \dots, x \ \mathbf{raises} \ E, \dots, E$$

---

${}^2\tau$  is called pure type.

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

## Typing (cont'd)

$$\frac{\text{Typeof}(c) = \tau}{\Sigma \vdash c : (\tau, \perp)} \quad \frac{x : \theta \in \Sigma \quad \theta \text{ pure}}{\Sigma \vdash x : (\theta, \perp)} \quad \frac{x : \tau \text{ \textbf{ref}} \in \Sigma}{\Sigma \vdash !x : (\tau, \textbf{reads } x)}$$

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

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

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

## Typing (cont'd)

$$\frac{\Sigma \vdash e_1 : (\tau_1, \epsilon_1) \quad \Sigma, x : \tau_1 \text{ **ref** } \vdash e_2 : (\theta_2, \epsilon_2) \quad x \notin \theta_2}{\Sigma \vdash \text{ **let** } x = \text{ **ref** } e_1 \text{ **in** } e_2 : (\theta_2, \epsilon_1 \sqcup \epsilon_2 \setminus x)}$$

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

$$\frac{\Sigma \vdash e : (\text{**unit**, \epsilon) \quad \Sigma \vdash p \text{ wf} \quad \Sigma \vdash t : \text{**int**}}{\Sigma \vdash \text{ **loop** } e \{ \text{**invariant** } p \text{ **variant** } t \} : (\text{**unit**, \epsilon)}$$

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

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

# Typing (cont'd)

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

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

$$\frac{\Sigma, x : \theta \vdash p \ \mathbf{wf} \quad \Sigma, x : \theta \vdash e \ \{q\} : (\theta', \epsilon)}{\Sigma \vdash \mathbf{fun} \ (x : \theta) \rightarrow \{p\} \ e \ \{q\} : ((x : \theta) \rightarrow \{p\} \theta' \ \epsilon \{q\}, \perp)}$$

$$\frac{\Sigma' \equiv \Sigma, x_1 : \theta_1, \dots, x_n : \theta_n \quad \Sigma' \vdash p \ \mathbf{wf} \quad \Sigma' \vdash t \ \mathbf{int} \quad \Sigma', f : (x_1 : \theta_1) \rightarrow \dots (x_n : \theta_n) \rightarrow \{p\} \theta \ \epsilon \{q\} \vdash e \ \{q\} : (\theta, \epsilon)}{\Sigma \vdash \mathbf{rec} \ f(x_1 : \theta_1) \dots (x_n : \theta_n) : \theta \ \{\mathbf{variant} \ t\} = \{p\} \ e \ \{q\} : ((x_1 : \theta_1) \rightarrow \dots (x_n : \theta_n) \rightarrow \{p\} \theta \ \epsilon \{q\}, \perp)}$$



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

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

$$\frac{\frac{\alpha \quad a:\text{int ref} \in \Sigma \quad a \notin (\text{int}, \text{reads } x \ y)}{\Sigma \vdash f \ a : ((y:\text{int ref}) \rightarrow ((\text{int}, \text{reads } a \ y), \perp), \perp)} \quad \frac{\text{not provable}}{a:\text{int ref} \in \Sigma \quad a \notin (\text{int}, \text{reads } a \ y)}}{\Sigma \vdash f \ a \ a : (\text{int}, \text{reads } a)}$$


$$\alpha : \frac{\beta}{\Sigma \vdash f : ((x:\text{int ref}) \rightarrow ((y:\text{int ref}) \rightarrow (\text{int}, \text{reads } x \ y), \perp), \perp)}$$

$$\frac{\frac{x : \text{int ref} \in \Sigma, x:\text{int ref}, y:\text{int ref}}{\Sigma, x:\text{int ref}, y:\text{int ref} \vdash !x : (\text{int}, \text{reads } x)} \quad \frac{y : \text{int ref} \in \Sigma, x:\text{int ref}, y:\text{int ref}}{\Sigma, x:\text{int ref}, y:\text{int ref} \vdash !y : (\text{int}, \text{reads } y)}}{\Sigma, x:\text{int ref}, y:\text{int ref} \vdash !x+!y : (\text{int}, \text{reads } x \ y)}$$

$$\beta : \frac{\Sigma, x:\text{int ref} \vdash \text{fun } (y:\text{int ref}) \rightarrow !x+!y : ((y:\text{int ref}) \rightarrow (\text{int}, \text{reads } x \ y), \perp)}{\Sigma \vdash \text{fun } (x:\text{int ref}) \rightarrow \text{fun } (y:\text{int ref}) \rightarrow !x+!y : ((x:\text{int ref}) \rightarrow ((y:\text{int ref}) \rightarrow (\text{int}, \text{reads } x \ y), \perp), \perp)}$$

where

  $a:\text{int ref} \in \Sigma$ , and

  $f \equiv \text{fun } (x:\text{int ref}) \rightarrow \text{fun } (y:\text{int ref}) \rightarrow !x+!y$

# Outline

- 1 Introduction
- 2 Why Language
  - Logic
  - Program
  - Weakest Precondition
- 3 Dijkstra's Dutch Flag
- 4 References

# 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
  - ☀  $r = E_1 \Rightarrow q_1; \dots; E_n \Rightarrow q_n$  is the set of properties to hold to each possibly uncaught exception.

# WP - Terms

## Rule

$$wp(t, q; r) = q[result \leftarrow t]$$

## Example

Program:

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

Verification Condition:

$$\begin{aligned} \text{🌐 } \forall x : Z, x > 0 &\rightarrow x - 1 \geq 0 \\ &wp(x - 1, result \geq 0) \\ &= (result \geq 0)[result \leftarrow x - 1] \\ &= x - 1 \geq 0 \end{aligned}$$

# WP - Let Expressions

## Rule

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

## Example

Program:

```
let x = 2 in x * x {result = 4}
```

Verification Condition:

$$\text{🌐 } 2 * 2 = 4$$

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

# WP - If-Then-Else

## Rule

$$wp(\mathbf{if } e_1 \mathbf{ then } e_2 \mathbf{ else } e_3, q; r) = wp(e_1, \mathbf{if } result \mathbf{ then } wp(e_2, q; r) \mathbf{ else } wp(e_3, q; r); r)$$

## Example

Program:

**if**  $x > y$  **then**  $x$  **else**  $y$  { **result**  $\geq x$  **and** **result**  $\geq y$  }

Verification Conditions:

$$\text{⊕ } \forall x : Z. \forall y : Z. \forall H1 : x > y. x \geq x \wedge x \geq y$$

$$\text{⊕ } \forall x : Z. \forall y : Z. \forall H1 : x \leq y. y \geq x \wedge y \geq y$$

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

# WP - Labels

## Rule

$$wp(L : e, q; r) = wp(e, q; r)[\mathbf{at}(x, L) \leftarrow x]$$

## Example

Program:

$L$ : **let**  $x = x + 1$  {  $result > x@L$  } **in**  $x$

Verification Condition:

  $x + 1 > x$

$$\begin{aligned} & wp(L : x + 1, result > x@L) \\ &= wp(x + 1, result > x@L)[\mathbf{at}(x, L) \leftarrow x] \\ &= (result > x@L)[result \leftarrow x + 1][\mathbf{at}(x, L) \leftarrow x] \\ &= (x + 1 > x) \end{aligned}$$



## Rules

$$wp(\mathbf{raise} (E e) : \theta, q; r) = wp(e, r(E); r)$$

$$wp(\mathbf{try} e_1 \mathbf{with} E x \rightarrow e_2 \mathbf{end}, q; r) = wp(e_1, q; E \Rightarrow wp(e_2, q; r)[x \leftarrow result]; r)$$

$$wp(\mathbf{assert} \{p\}; e, q; r) = p \wedge wp(e, q; r)$$

$$wp(e \{q'; r'\}, q; r) = wp(e, q' \wedge q; r' \wedge r)$$

$$wp(e \{\{q'; r'\}\}, q; r) = wp(e, q', r') \wedge \forall \omega^a. \forall result. q' \Rightarrow q \wedge r' \Rightarrow r$$

<sup>a</sup> $\omega$  stands for the set of possibly modified references.

## Example

Program:

```

try
  if  $x > 0$  then  $y / x$  else raise dbz
with dbz  $\rightarrow 0$ 
end
{result  $\geq 0$ }

```

Verification Conditions:

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

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

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

# WP - Assignments

## Rules

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

## Example

Program:

```

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

```

Verification Condition:

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

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

## Rules

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

$$\begin{aligned}
 & wp(\text{while } e_1 \text{ do } e_2 \{ \text{invariant } p \text{ variant } t \}, q; r) \\
 = & p \wedge \forall \omega. p \Rightarrow wp( \\
 & \quad \text{try} \\
 & \quad \quad \text{loop if } e_1 \text{ then } e_2 \text{ else raise } E \\
 & \quad \quad \{ \text{invariant } p \text{ variant } t \} \\
 & \quad \quad \text{with } E \_ \rightarrow \text{void} \\
 & \quad \quad \text{end}, q; r) \\
 = & p \wedge \forall \omega. p \Rightarrow wp(L : \text{if } e_1 \text{ then } e_2 \text{ else raise } E, p \wedge t < \text{at}(t, L); E \Rightarrow q, r) \\
 = & p \wedge \forall \omega. p \Rightarrow wp(e_1, \text{if result then } wp(e_2, p \wedge t < \text{at}(t, L)) \text{ else } q; r)[\text{at}(x, L) \leftarrow x]
 \end{aligned}$$

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

# WP - Loops (cont'd)

## Example

Program:

```
{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.

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

# WP - Loops (cont'd)

## Example (cont'd)

Program:

```

{ $x > 0$ } while ! $x > 0$  do {invariant  $x \geq 0$  variant  $x$ }
   $x := !x - 1$ 
done

```

Weakest precondition calculation:



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

## Function Constructs

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

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

where

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

# WP - Functions (cont'd)

## Function Calls

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

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

## Example

Program:

```

let f (x : int ref) =
  {x > 0} let _ = x := !x - 1 in !x {x = x@ - 1}

let _ = let x = ref 10 in let _ = f x {x < x@} in !x
  
```

Verification Conditions:

$$\begin{aligned}
 & wp(\mathbf{let} \ x = \mathbf{ref} \ 10 \ \mathbf{in} \ f \ x, x < \mathbf{old}(x)) \\
 = & wp(10, (wp(f \ x, x < \mathbf{old}(x)))[x \leftarrow result]) \\
 = & (wp(f \ x, x < \mathbf{old}(x)))[x \leftarrow result][result \leftarrow 10] \\
 = & (x > 0 \wedge \forall x. \forall result. (x = \mathbf{old}(x) - 1 \Rightarrow x < \mathbf{old}(x))[\mathbf{old}(x) \leftarrow x])[x \leftarrow 10] \\
 = & 10 > 0 \wedge \forall x. \forall result. (x = \mathbf{old}(x) - 1 \Rightarrow x < \mathbf{old}(x))[\mathbf{old}(x) \leftarrow 10] \\
 = & 10 > 0 \wedge \forall x. \forall result. x = 10 - 1 \Rightarrow x < 10
 \end{aligned}$$



# WP - Functions (cont'd)

## Example

Program:

```

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

Verification Conditions:

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

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

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

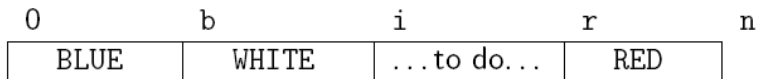
# Outline

- 1 Introduction
- 2 Why Language
  - Logic
  - Program
  - Weakest Precondition
- 3 Dijkstra's Dutch Flag
- 4 References

# 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  $\rightarrow$  c2 : color  $\rightarrow$ 
```

```
  {}
```

```
  bool
```

```
  {if result then c1 = c2 else c1  $\diamond$  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
  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) }
```

# 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.