



COMP4161: Advanced Topics in Software Verification



Gerwin Klein, June Andronick, Ramana Kumar, Miki Tanaka
S2/2017



Content



- Intro & motivation, getting started [1]

- Foundations & Principles
 - Lambda Calculus, natural deduction [1,2]
 - Higher Order Logic [3^a]
 - Term rewriting [4]

- Proof & Specification Techniques
 - Inductively defined sets, rule induction [5]
 - Datatypes, recursion, induction [6, 7]
 - Hoare logic, proofs about programs, C verification [8^b,9]
 - (mid-semester break)
 - Writing Automated Proof Methods [10]
 - Isar, codegen, typeclasses, locales [11^c,12]

^aa1 due; ^ba2 due; ^ca3 due

Deep Embeddings



We used a **datatype** *com* to represent the **syntax** of IMP.

→ We then defined its semantics over this datatype.

Deep Embeddings



We used a **datatype** *com* to represent the **syntax** of IMP.

→ We then defined its semantics over this datatype.

This is called a **deep embedding**:

→ separate representation of language terms and their semantics.

Deep Embeddings



We used a **datatype** *com* to represent the **syntax** of IMP.

→ We then defined its semantics over this datatype.

This is called a **deep embedding**:

→ separate representation of language terms and their semantics.

Advantages:

→ Prove general theorems about the **language**, not just of programs.

→ e.g. expressiveness, correct compilation, inference completeness ...

→ usually by structural induction over the syntax type.

Deep Embeddings



We used a **datatype** *com* to represent the **syntax** of IMP.

→ We then defined its semantics over this datatype.

This is called a **deep embedding**:

→ separate representation of language terms and their semantics.

Advantages:

- Prove general theorems about the **language**, not just of programs.
- e.g. expressiveness, correct compilation, inference completeness ...
- usually by structural induction over the syntax type.

Disadvantages:

- Semantically equivalent programs are not obviously equal.
- e.g. "IF True THEN SKIP ELSE SKIP = SKIP" is not a true theorem.
- Many concepts already present in the logic are reinvented in the language.

Shallow Embeddings



Shallow Embedding: represent only the semantics, directly in the logic.

- A definition for each language construct, giving its **semantics**.
- Programs are represented as instances of these definitions.

Shallow Embeddings



Shallow Embedding: represent only the semantics, directly in the logic.

- A definition for each language construct, giving its **semantics**.
- Programs are represented as instances of these definitions.

Example: program semantics as functions $state \Rightarrow state$

SKIP \equiv

Shallow Embeddings



Shallow Embedding: represent only the semantics, directly in the logic.

- A definition for each language construct, giving its **semantics**.
- Programs are represented as instances of these definitions.

Example: program semantics as functions $state \Rightarrow state$

$$\text{SKIP} \equiv \lambda s. s$$

Shallow Embeddings



Shallow Embedding: represent only the semantics, directly in the logic.

- A definition for each language construct, giving its **semantics**.
- Programs are represented as instances of these definitions.

Example: program semantics as functions $state \Rightarrow state$

SKIP $\equiv \lambda s. s$

IF b THEN c ELSE d \equiv

Shallow Embeddings



Shallow Embedding: represent only the semantics, directly in the logic.

- A definition for each language construct, giving its **semantics**.
- Programs are represented as instances of these definitions.

Example: program semantics as functions $state \Rightarrow state$

SKIP $\equiv \lambda s. s$

IF b THEN c ELSE d $\equiv \lambda s. \text{if } b \text{ s then } c \text{ s else } d \text{ s}$

Shallow Embeddings



Shallow Embedding: represent only the semantics, directly in the logic.

- A definition for each language construct, giving its **semantics**.
- Programs are represented as instances of these definitions.

Example: program semantics as functions $state \Rightarrow state$

$SKIP \equiv \lambda s. s$

$IF\ b\ THEN\ c\ ELSE\ d \equiv \lambda s. \text{if } b\ s\ \text{then } c\ s\ \text{else } d\ s$

- “IF True THEN SKIP ELSE SKIP = SKIP” is now a true statement.
- can use the simplifier to do semantics-preserving program rewriting.

Shallow Embeddings



Shallow Embedding: represent only the semantics, directly in the logic.

- A definition for each language construct, giving its **semantics**.
- Programs are represented as instances of these definitions.

Example: program semantics as functions $state \Rightarrow state$

$SKIP \equiv \lambda s. s$

$IF\ b\ THEN\ c\ ELSE\ d \equiv \lambda s. \text{if } b\ s\ \text{then } c\ s\ \text{else } d\ s$

- “IF True THEN SKIP ELSE SKIP = SKIP” is now a true statement.
- can use the simplifier to do semantics-preserving program rewriting.

Today: a shallow embedding for (interesting parts of) C semantics

Records in Isabelle



Records are a tuples with named components

Records in Isabelle



Records are a tuples with named components

Example:

```
record A =  a :: nat  
            b :: int
```

Records in Isabelle



Records are a tuples with named components

Example:

```
record A =  a :: nat  
           b :: int
```

→ Selectors: a :: A ⇒ nat, b :: A ⇒ int, a r = Suc 0

Records in Isabelle



Records are a tuples with named components

Example:

```
record A =  a :: nat  
           b :: int
```

- Selectors: $a :: A \Rightarrow \text{nat}$, $b :: A \Rightarrow \text{int}$, $a \ r = \text{Suc } 0$
- Constructors: $(\lambda a = \text{Suc } 0, b = -1 \)$

Records in Isabelle



Records are a tuples with named components

Example:

```
record A =  a :: nat
           b :: int
```

- Selectors: $a :: A \Rightarrow \text{nat}$, $b :: A \Rightarrow \text{int}$, $a\ r = \text{Suc } 0$
- Constructors: $(\lambda a = \text{Suc } 0, b = -1)$
- Update: $r(\lambda a := \text{Suc } 0)$, $b_update (\lambda b. b + 1)\ r$

Records in Isabelle



Records are a tuples with named components

Example:

```
record A =  a :: nat
           b :: int
```

- Selectors: $a :: A \Rightarrow \text{nat}$, $b :: A \Rightarrow \text{int}$, $a\ r = \text{Suc } 0$
- Constructors: $(\lambda a = \text{Suc } 0, b = -1)$
- Update: $r(\lambda a := \text{Suc } 0)$, $b_update (\lambda b. b + 1)\ r$

Records are extensible:

```
record B = A +
           c :: nat list
```

Records in Isabelle



Records are a tuples with named components

Example:

```
record A =  a :: nat
           b :: int
```

- Selectors: $a :: A \Rightarrow \text{nat}$, $b :: A \Rightarrow \text{int}$, $a\ r = \text{Suc } 0$
- Constructors: $(\ | a = \text{Suc } 0, b = -1 \ |)$
- Update: $r(\ a := \text{Suc } 0 \ |)$, $b_update\ (\lambda b. b + 1)\ r$

Records are extensible:

```
record B = A +
           c :: nat list
```

```
(\ | a = \text{Suc } 0, b = -1, c = [0, 0] \ |)
```

A background pattern of white hexagons on a teal background, arranged in a staggered grid.

DATA
61



Demo

Nondeterministic State Monad with Failure



Shallow embedding suitable for (a useful fragment of) C.

Nondeterministic State Monad with Failure



Shallow embedding suitable for (a useful fragment of) C.

Can express lots of C ideas:

- Access to `volatile` variables, external APIs: **Nondeterminism**
- Undefined behaviour: **Failure**
- Early exit (`return`, `break`, `continue`): **Exceptional control flow**

Nondeterministic State Monad with Failure



Shallow embedding suitable for (a useful fragment of) C.

Can express lots of C ideas:

- Access to `volatile` variables, external APIs: **Nondeterminism**
- Undefined behaviour: **Failure**
- Early exit (`return`, `break`, `continue`): **Exceptional control flow**

Relatively straightforward Hoare logic

Nondeterministic State Monad with Failure



Shallow embedding suitable for (a useful fragment of) C.

Can express lots of C ideas:

- Access to `volatile` variables, external APIs: **Nondeterminism**
- Undefined behaviour: **Failure**
- Early exit (`return`, `break`, `continue`): **Exceptional control flow**

Relatively straightforward Hoare logic

Used extensively in the seL4 verification work:

- Formalism for the seL4 abstract, design and *capDL* specifications
- Calculus for proving **refinement** between them and down to code.

Nondeterministic State Monad with Failure



Shallow embedding suitable for (a useful fragment of) C.

Can express lots of C ideas:

- Access to `volatile` variables, external APIs: **Nondeterminism**
- Undefined behaviour: **Failure**
- Early exit (`return`, `break`, `continue`): **Exceptional control flow**

Relatively straightforward Hoare logic

Used extensively in the seL4 verification work:

- Formalism for the seL4 abstract, design and *capDL* specifications
- Calculus for proving **refinement** between them and down to code.

AutoCorres: verified translation of C to monadic representation

- Specifically designed for humans to do proofs over.

State Monad: Motivation



Model the **semantics** of a (deterministic) computation as a function

$$'s \Rightarrow ('a \times 's)$$

State Monad: Motivation



Model the **semantics** of a (deterministic) computation as a function

$$'s \Rightarrow ('a \times 's)$$

The computation operates over a **state** of type `'s`:

→ Includes all global variables, external devices, etc.

State Monad: Motivation



Model the **semantics** of a (deterministic) computation as a function

$$'s \Rightarrow ('a \times 's)$$

The computation operates over a **state** of type `'s`:

→ Includes all global variables, external devices, etc.

The computation also yields a **return value** of type `'a`:

→ e.g. a program's exit status (in POSIX, `'a` would be 8-bit words)

→ e.g. return-value of a C function

State Monad: Motivation



Model the **semantics** of a (deterministic) computation as a function

$$'s \Rightarrow ('a \times 's)$$

The computation operates over a **state** of type `'s`:

→ Includes all global variables, external devices, etc.

The computation also yields a **return value** of type `'a`:

→ e.g. a program's exit status (in POSIX, `'a` would be 8-bit words)

→ e.g. return-value of a C function

return – the computation that leaves the state unchanged and returns its argument:

$$\text{return } x \equiv \lambda s.$$

State Monad: Motivation



Model the **semantics** of a (deterministic) computation as a function

$$'s \Rightarrow ('a \times 's)$$

The computation operates over a **state** of type `'s`:

→ Includes all global variables, external devices, etc.

The computation also yields a **return value** of type `'a`:

→ e.g. a program's exit status (in POSIX, `'a` would be 8-bit words)

→ e.g. return-value of a C function

return – the computation that leaves the state unchanged and returns its argument:

$$\text{return } x \equiv \lambda s. (x, s)$$

State Monad: Basic Operations



get – returns the entire state without modifying it:

$$\text{get} \equiv \lambda s.$$

State Monad: Basic Operations



get – returns the entire state without modifying it:

$$\text{get} \equiv \lambda s. (s,s)$$

State Monad: Basic Operations



get – returns the entire state without modifying it:

$$\text{get} \equiv \lambda s. (s,s)$$

put – replaces the state and returns the unit value ():

$$\text{put } s \equiv$$

State Monad: Basic Operations



get – returns the entire state without modifying it:

$$\text{get} \equiv \lambda s. (s,s)$$

put – replaces the state and returns the unit value ():

$$\text{put } s \equiv \lambda_. ((),s)$$

State Monad: Basic Operations



get – returns the entire state without modifying it:

$$\text{get} \equiv \lambda s. (s,s)$$

put – replaces the state and returns the unit value ():

$$\text{put } s \equiv \lambda_. ((),s)$$

bind – sequences two computations; 2nd takes the first's result:

$$c \gg= d \equiv$$

State Monad: Basic Operations



get – returns the entire state without modifying it:

$$\text{get} \equiv \lambda s. (s,s)$$

put – replaces the state and returns the unit value ():

$$\text{put } s \equiv \lambda_. ((),s)$$

bind – sequences two computations; 2nd takes the first's result:

$$c \gg= d \equiv \lambda s. \text{let } (r,s') = c \text{ s in } d r s'$$

State Monad: Basic Operations



get – returns the entire state without modifying it:

$$\text{get} \equiv \lambda s. (s,s)$$

put – replaces the state and returns the unit value ():

$$\text{put } s \equiv \lambda_. ((),s)$$

bind – sequences two computations; 2nd takes the first's result:

$$c \gg= d \equiv \lambda s. \text{let } (r,s') = c \text{ s in } d r s'$$

gets – returns a projection of the state; leaves state unchanged:

$$\text{gets } f \equiv$$

State Monad: Basic Operations



get – returns the entire state without modifying it:

$$\text{get} \equiv \lambda s. (s,s)$$

put – replaces the state and returns the unit value ():

$$\text{put } s \equiv \lambda_. ((),s)$$

bind – sequences two computations; 2nd takes the first's result:

$$c \gg= d \equiv \lambda s. \text{let } (r,s') = c \text{ s in } d r s'$$

gets – returns a projection of the state; leaves state unchanged:

$$\text{gets } f \equiv \text{get } \gg= (\lambda s. \text{return } (f s))$$

State Monad: Basic Operations



get – returns the entire state without modifying it:

$$\text{get} \equiv \lambda s. (s, s)$$

put – replaces the state and returns the unit value ():

$$\text{put } s \equiv \lambda _. ((), s)$$

bind – sequences two computations; 2nd takes the first's result:

$$c \gg= d \equiv \lambda s. \text{let } (r, s') = c \text{ s in } d r s'$$

gets – returns a projection of the state; leaves state unchanged:

$$\text{gets } f \equiv \text{get} \gg= (\lambda s. \text{return } (f s))$$

modify – applies its argument to modify the state; returns ():

$$\text{modify } f \equiv \text{get} \gg= (\lambda s. \text{put } (f s))$$

Monads, Laws



Formally: a monad \mathbf{M} is a type constructor with two operations.

$\text{return} :: \alpha \Rightarrow \mathbf{M} \alpha$ $\text{bind} :: \mathbf{M} \alpha \Rightarrow (\alpha \Rightarrow \mathbf{M} \beta) \Rightarrow \mathbf{M} \beta$

Monads, Laws



Formally: a monad \mathbf{M} is a type constructor with two operations.

$\text{return} :: \alpha \Rightarrow \mathbf{M} \alpha$ $\text{bind} :: \mathbf{M} \alpha \Rightarrow (\alpha \Rightarrow \mathbf{M} \beta) \Rightarrow \mathbf{M} \beta$

Infix Notation: $a \gg= b$ is infix notation for $\text{bind } a b$

Monads, Laws



Formally: a monad \mathbf{M} is a type constructor with two operations.

$\text{return} :: \alpha \Rightarrow \mathbf{M} \alpha$ $\text{bind} :: \mathbf{M} \alpha \Rightarrow (\alpha \Rightarrow \mathbf{M} \beta) \Rightarrow \mathbf{M} \beta$

Infix Notation: $a \gg= b$ is infix notation for $\text{bind } a b$

Do-Notation: $a \gg= (\lambda x. b x)$ is often written as **do** $\{ x \leftarrow a; b x \}$

Monads, Laws



Formally: a monad \mathbf{M} is a type constructor with two operations.

$\text{return} :: \alpha \Rightarrow \mathbf{M} \alpha$ $\text{bind} :: \mathbf{M} \alpha \Rightarrow (\alpha \Rightarrow \mathbf{M} \beta) \Rightarrow \mathbf{M} \beta$

Infix Notation: $a \gg= b$ is infix notation for $\text{bind } a b$

Do-Notation: $a \gg= (\lambda x. b x)$ is often written as **do** $\{ x \leftarrow a; b x \}$

Monad Laws:

Monads, Laws



Formally: a monad \mathbf{M} is a type constructor with two operations.

$\text{return} :: \alpha \Rightarrow \mathbf{M} \alpha$ $\text{bind} :: \mathbf{M} \alpha \Rightarrow (\alpha \Rightarrow \mathbf{M} \beta) \Rightarrow \mathbf{M} \beta$

Infix Notation: $a \gg= b$ is infix notation for $\text{bind } a b$

Do-Notation: $a \gg= (\lambda x. b x)$ is often written as **do** $\{ x \leftarrow a; b x \}$

Monad Laws:

return-left: $(\text{return } x \gg= f) = f x$

Monads, Laws



Formally: a monad \mathbf{M} is a type constructor with two operations.

$\text{return} :: \alpha \Rightarrow \mathbf{M} \alpha$ $\text{bind} :: \mathbf{M} \alpha \Rightarrow (\alpha \Rightarrow \mathbf{M} \beta) \Rightarrow \mathbf{M} \beta$

Infix Notation: $a \gg= b$ is infix notation for $\text{bind } a b$

Do-Notation: $a \gg= (\lambda x. b x)$ is often written as **do** $\{ x \leftarrow a; b x \}$

Monad Laws:

return-left: $(\text{return } x \gg= f) = f x$

return-right: $(m \gg= \text{return}) = m$

Monads, Laws



Formally: a monad \mathbf{M} is a type constructor with two operations.

$\text{return} :: \alpha \Rightarrow \mathbf{M} \alpha$ $\text{bind} :: \mathbf{M} \alpha \Rightarrow (\alpha \Rightarrow \mathbf{M} \beta) \Rightarrow \mathbf{M} \beta$

Infix Notation: $a \gg= b$ is infix notation for $\text{bind } a b$

Do-Notation: $a \gg= (\lambda x. b x)$ is often written as **do** $\{ x \leftarrow a; b x \}$

Monad Laws:

return-left: $(\text{return } x \gg= f) = f x$

return-right: $(m \gg= \text{return}) = m$

bind-assoc: $((a \gg= b) \gg= c) = (a \gg= (\lambda x. b x \gg= c))$

State Monad: Example



A fragment of C:

```
void f(int *p) {  
    int x = *p;  
    if (x < 10) {  
        *p = x+1;  
    }  
}
```


State Monad: Example



A fragment of C:

```
void f(int *p) {  
    int x = *p;  
    if (x < 10) {  
        *p = x+1;  
    }  
}
```

record state =

hp :: int ptr \Rightarrow int

f :: "int ptr \Rightarrow (state \Rightarrow (unit,state))"

f p \equiv

do {

x \leftarrow gets (λ s. hp s p);

if x < 10 **then**

 modify (hp_update (λ h. (h(p := x + 1))))

else

 return ()

}

State Monad with Failure



Computations can **fail**: $'s \Rightarrow (('a \times 's) \times \underline{\text{bool}})$

State Monad with Failure



Computations can **fail**: $'s \Rightarrow (('a \times 's) \times \underline{\text{bool}})$

bind – fails when either computation fails

$\text{bind } a \ b \equiv \mathbf{let} \ ((r,s'),f) = a \ s; \ ((r'',s''),f') = b \ r \ s' \ \mathbf{in} \ ((r'',s''), f \vee f')$

State Monad with Failure



Computations can **fail**: $'s \Rightarrow (('a \times 's) \times \text{bool})$

bind – fails when either computation fails

$\text{bind } a \ b \equiv \text{let } ((r,s'),f) = a \ s; ((r'',s''),f') = b \ r \ s' \text{ in } ((r'',s''), f \vee f')$

fail – the computation that always fails:

$\text{fail} \equiv \lambda s. (\text{undefined}, \text{True})$

State Monad with Failure



Computations can **fail**: $'s \Rightarrow (('a \times 's) \times \text{bool})$

bind – fails when either computation fails

$\text{bind } a \ b \equiv \text{let } ((r,s'),f) = a \ s; ((r'',s''),f') = b \ r \ s' \text{ in } ((r'',s''), f \vee f')$

fail – the computation that always fails:

$\text{fail} \equiv \lambda s. (\text{undefined}, \text{True})$

assert – fails when given condition is False:

$\text{assert } P \equiv \text{if } P \text{ then return } () \text{ else fail}$

State Monad with Failure



Computations can **fail**: $'s \Rightarrow (('a \times 's) \times \underline{\text{bool}})$

bind – fails when either computation fails

$\text{bind } a \ b \equiv \text{let } ((r,s'),f) = a \ s; ((r'',s''),f') = b \ r \ s' \text{ in } ((r'',s''), f \vee f')$

fail – the computation that always fails:

$\text{fail} \equiv \lambda s. (\text{undefined}, \text{True})$

assert – fails when given condition is False:

$\text{assert } P \equiv \text{if } P \text{ then return } () \text{ else fail}$

guard – fails when given condition applied to the state is False:

$\text{guard } P \equiv \text{get } \gg = (\lambda s. \text{assert } (P \ s))$

Guards



Used to assert the absence of **undefined behaviour** in C

Guards



Used to assert the absence of **undefined behaviour** in C

→ pointer validity, absence of divide by zero, signed overflow, etc.

Guards



Used to assert the absence of **undefined behaviour** in C

→ pointer validity, absence of divide by zero, signed overflow, etc.

```
f p ≡  
do {  
  y ← guard (λs. valid s p);  
  x ← gets (λs. hp s p);  
  if x < 10 then  
    modify (hp_update (λh. (h(p := x + 1))))  
  else  
    return ()  
}
```

Nondeterministic State Monad with Failure



Computations can be **nondeterministic**: $'s \Rightarrow (('a \times 's) \text{ set} \times \text{bool})$

Nondeterministic State Monad with Failure



Computations can be **nondeterministic**: $'s \Rightarrow (('a \times 's) \text{ set} \times \text{bool})$

Nondeterminism: computations return a **set** of possible results.

→ Allows **underspecification**: e.g. malloc, external devices, etc.

Nondeterministic State Monad with Failure



Computations can be **nondeterministic**: $'s \Rightarrow (('a \times 's) \text{ set} \times \text{bool})$

Nondeterminism: computations return a **set** of possible results.

→ Allows **underspecification**: e.g. malloc, external devices, etc.

bind – runs 2nd computation for all results returned by the first:

$$\text{bind } a \ b \equiv \ \lambda s. (\{(r'', s''). \exists (r', s') \in \text{fst } (a \ s). (r'', s'') \in \text{fst } (b \ r' \ s')\}, \\ \text{snd } (a \ s) \vee (\exists (r', s') \in \text{fst } (a \ s). \text{snd } (b \ r' \ s')))$$

Nondeterministic State Monad with Failure



Computations can be **nondeterministic**: $'s \Rightarrow (('a \times 's) \text{ set} \times \text{bool})$

Nondeterminism: computations return a **set** of possible results.

→ Allows **underspecification**: e.g. malloc, external devices, etc.

bind – runs 2nd computation for all results returned by the first:

$$\text{bind } a \ b \equiv \ \lambda s. (\{(r'', s''). \exists (r', s') \in \text{fst } (a \ s). (r'', s'') \in \text{fst } (b \ r' \ s')\}, \\ \text{snd } (a \ s) \vee (\exists (r', s') \in \text{fst } (a \ s). \text{snd } (b \ r' \ s')))$$

All non-failing computations so far are **deterministic**:

→ e.g. $\text{return } x \equiv \lambda s. (\{(x, s)\}, \text{False})$

→ Others are similar.

Nondeterministic State Monad with Failure



Computations can be **nondeterministic**: $'s \Rightarrow (('a \times 's) \text{ set} \times \text{bool})$

Nondeterminism: computations return a **set** of possible results.

→ Allows **underspecification**: e.g. malloc, external devices, etc.

bind – runs 2nd computation for all results returned by the first:

$$\text{bind } a \ b \equiv \ \lambda s. (\{(r'', s''). \exists (r', s') \in \text{fst } (a \ s). (r'', s'') \in \text{fst } (b \ r' \ s')\}, \\ \text{snd } (a \ s) \vee (\exists (r', s') \in \text{fst } (a \ s). \text{snd } (b \ r' \ s')))$$

All non-failing computations so far are **deterministic**:

→ e.g. $\text{return } x \equiv \lambda s. (\{(x, s)\}, \text{False})$

→ Others are similar.

select – nondeterministic selection from a set:

$$\text{select } A \equiv \lambda s. ((A \times \{s\}), \text{False})$$

A background pattern of white hexagons on a dark teal background, arranged in a staggered grid.

DATA
61



Demo

While Loops



Monadic while loop, defined **inductively**.

While Loops



Monadic while loop, defined **inductively**.

$$\begin{aligned} \text{whileLoop} &:: ('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow \\ &('a \Rightarrow ('s \Rightarrow ('a \times 's) \text{ set} \times \text{bool})) \Rightarrow \\ &('a \Rightarrow ('s \Rightarrow ('a \times 's) \text{ set} \times \text{bool})) \end{aligned}$$

While Loops



Monadic while loop, defined **inductively**.

$$\begin{aligned} \text{whileLoop} &:: ('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow \\ &('a \Rightarrow ('s \Rightarrow ('a \times 's) \text{ set} \times \text{bool})) \Rightarrow \\ &('a \Rightarrow ('s \Rightarrow ('a \times 's) \text{ set} \times \text{bool})) \end{aligned}$$

$\text{whileLoop } C B$

- **condition** C : takes **loop parameter** and **state** as arguments, returns **bool**
- **monadic body** B : takes **loop parameter** as argument, return-value is the **updated** loop parameter
- **fails** if the loop body ever fails or if the loop never terminates

While Loops



Monadic while loop, defined **inductively**.

$$\begin{aligned} \text{whileLoop} &:: ('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow \\ &('a \Rightarrow ('s \Rightarrow ('a \times 's) \text{ set} \times \text{bool})) \Rightarrow \\ &('a \Rightarrow ('s \Rightarrow ('a \times 's) \text{ set} \times \text{bool})) \end{aligned}$$

$\text{whileLoop } C B$

- **condition** C : takes **loop parameter** and **state** as arguments, returns **bool**
- **monadic body** B : takes **loop parameter** as argument, return-value is the **updated** loop parameter
- **fails** if the loop body ever fails or if the loop never terminates

Example: $\text{whileLoop } (\lambda p s. \text{hp } s \ p = 0) (\lambda p. \text{return } (\text{ptrAdd } p \ 1)) \ p$

Defining While Loops Inductively



Two-part definition: results and termination

Defining While Loops Inductively



Two-part definition: results and termination

Results: $\text{while_results} :: ('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow$
 $('a \Rightarrow ('s \Rightarrow ('a \times 's) \text{ set} \times \text{bool})) \Rightarrow$
 $((('a \times 's) \text{ option}) \times (('a \times 's) \text{ option})) \text{ set}$

Defining While Loops Inductively



Two-part definition: results and termination

Results: $\text{while_results} :: ('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow$
 $('a \Rightarrow ('s \Rightarrow ('a \times 's) \text{ set} \times \text{bool})) \Rightarrow$
 $((('a \times 's) \text{ option}) \times (('a \times 's) \text{ option})) \text{ set}$

$$\frac{\neg C r s}{(\text{Some } (r,s), \text{Some } (r,s)) \in \text{while_results } C B} \text{ (terminate)}$$

Defining While Loops Inductively



Two-part definition: results and termination

Results: $\text{while_results} :: ('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow$
 $('a \Rightarrow ('s \Rightarrow ('a \times 's) \text{ set} \times \text{bool})) \Rightarrow$
 $((('a \times 's) \text{ option}) \times (('a \times 's) \text{ option})) \text{ set}$

$$\frac{\neg C r s}{(\text{Some } (r,s), \text{Some } (r,s)) \in \text{while_results } C B} \text{ (terminate)}$$

$$\frac{C r s \text{ snd } (B r s)}{(\text{Some } (r,s), \text{None}) \in \text{while_results } C B} \text{ (fail)}$$

Defining While Loops Inductively



Two-part definition: results and termination

Results: $\text{while_results} :: ('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow$
 $('a \Rightarrow ('s \Rightarrow ('a \times 's) \text{ set} \times \text{bool})) \Rightarrow$
 $((('a \times 's) \text{ option}) \times (('a \times 's) \text{ option})) \text{ set}$

$$\frac{\neg C r s}{(\text{Some } (r,s), \text{Some } (r,s)) \in \text{while_results } C B} \text{ (terminate)}$$

$$\frac{C r s \quad \text{snd } (B r s)}{(\text{Some } (r,s), \text{None}) \in \text{while_results } C B} \text{ (fail)}$$

$$\frac{C r s \quad (r',s') \in \text{fst } (B r s) \quad (\text{Some } (r', s'), z) \in \text{while_results } C B}{(\text{Some } (r,s), z) \in \text{while_results } C B} \text{ (loop)}$$

Defining While Loops Inductively



Termination:

$$\begin{aligned} \text{while_terminates} &:: ('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow \\ &('a \Rightarrow ('s \Rightarrow ('a \times 's) \text{ set} \times \text{bool})) \Rightarrow \\ &'a \Rightarrow 's \Rightarrow \text{bool} \end{aligned}$$

Defining While Loops Inductively



Termination:

$\text{while_terminates} :: ('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow$
 $('a \Rightarrow ('s \Rightarrow ('a \times 's) \text{ set} \times \text{bool})) \Rightarrow$
 $'a \Rightarrow 's \Rightarrow \text{bool}$

$$\frac{\neg C r s}{\text{while_terminates } C B r s} \text{ (terminate)}$$

Defining While Loops Inductively



Termination:

$\text{while_terminates} :: ('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow$
 $('a \Rightarrow ('s \Rightarrow ('a \times 's) \text{ set} \times \text{bool})) \Rightarrow$
 $'a \Rightarrow 's \Rightarrow \text{bool}$

$$\frac{\neg C r s}{\text{while_terminates } C B r s} \text{ (terminate)}$$

$$\frac{C r s \quad \forall (r',s') \in \text{fst } (B r s). \text{ while_terminates } C B r' s'}{\text{while_terminates } C B r s} \text{ (loop)}$$

Defining While Loops Inductively



Termination:

$$\text{while_terminates} :: ('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow$$
$$('a \Rightarrow ('s \Rightarrow ('a \times 's) \text{ set} \times \text{bool})) \Rightarrow$$
$$'a \Rightarrow 's \Rightarrow \text{bool}$$

$$\frac{\neg C r s}{\text{while_terminates } C B r s} \text{ (terminate)}$$

$$\frac{C r s \quad \forall (r',s') \in \text{fst } (B r s). \text{ while_terminates } C B r' s'}{\text{while_terminates } C B r s} \text{ (loop)}$$

$\text{whileLoop } C B \equiv$

$$(\lambda r s. (\{(r',s'). (\text{Some } (r, s), \text{Some } (r', s')) \in \text{while_results } C B\},$$
$$(\text{Some } (r, s), \text{None}) \in \text{while_results} \vee$$
$$\neg \text{while_terminates } C B r s))$$

Hoare Logic over Nondeterministic State Monads



Partial correctness:

$$\{P\} m \{Q\} \equiv \forall s. P s \longrightarrow \forall (r, s') \in \text{fst} (m s). Q r s'$$

→ Post-condition Q is a predicate of return-value and result state.

Weakest Precondition Rules

$$\{ \quad \} \text{ return } x \{ \lambda r s. P r s \} \quad \{ \quad \} \text{ get } \{P\} \quad \{ \quad \} \text{ put } x \{P\}$$

$$\{ \quad \} \text{ gets } f \{P\} \quad \{ \quad \} \text{ modify } f \{P\}$$

$$\{ \quad \} \text{ assert } P \{Q\} \quad \{ \quad \} \text{ fail } \{Q\}$$

Hoare Logic over Nondeterministic State Monads



Partial correctness:

$$\{P\} m \{Q\} \equiv \forall s. P s \longrightarrow \forall (r,s') \in \text{fst} (m s). Q r s'$$

→ Post-condition Q is a predicate of return-value and result state.

Weakest Precondition Rules

$$\{\lambda s. P x s\} \text{return } x \quad \{\lambda r s. P r s\} \quad \{ \quad \} \text{get } \{P\} \quad \{ \quad \} \text{put } x \{P\}$$

$$\{ \quad \} \text{gets } f \{P\} \quad \{ \quad \} \text{modify } f \{P\}$$

$$\{ \quad \} \text{assert } P \{Q\} \quad \{ \quad \} \text{fail } \{Q\}$$

Hoare Logic over Nondeterministic State Monads



Partial correctness:

$$\{P\} m \{Q\} \equiv \forall s. P s \longrightarrow \forall (r, s') \in \text{fst} (m s). Q r s'$$

→ Post-condition Q is a predicate of return-value and result state.

Weakest Precondition Rules

$$\{\lambda s. P x s\} \text{return } x \quad \{\lambda r s. P r s\} \quad \{\lambda s. P s s\} \text{get } \{P\} \quad \{ \quad \} \text{put } x \{P\}$$

$$\{ \quad \} \text{gets } f \{P\} \quad \{ \quad \} \text{modify } f \{P\}$$

$$\{ \quad \} \text{assert } P \{Q\} \quad \{ \quad \} \text{fail } \{Q\}$$

Hoare Logic over Nondeterministic State Monads



Partial correctness:

$$\{P\} m \{Q\} \equiv \forall s. P s \longrightarrow \forall (r,s') \in \text{fst} (m s). Q r s'$$

→ Post-condition Q is a predicate of return-value and result state.

Weakest Precondition Rules

$$\{\lambda s. P x s\} \text{ return } x \quad \{\lambda r s. P r s\} \quad \{\lambda s. P s s\} \text{ get } \{P\} \quad \{\lambda s. P () x\} \text{ put } x \{P\}$$

$$\{ \quad \} \text{ gets } f \{P\} \quad \{ \quad \} \text{ modify } f \{P\}$$

$$\{ \quad \} \text{ assert } P \{Q\} \quad \{ \quad \} \text{ fail } \{Q\}$$

Hoare Logic over Nondeterministic State Monads



Partial correctness:

$$\{P\} m \{Q\} \equiv \forall s. P s \longrightarrow \forall (r, s') \in \text{fst} (m s). Q r s'$$

→ Post-condition Q is a predicate of return-value and result state.

Weakest Precondition Rules

$$\{\lambda s. P x s\} \text{ return } x \{ \lambda r s. P r s \} \quad \{\lambda s. P s s\} \text{ get } \{P\} \quad \{\lambda s. P () x\} \text{ put } x \{P\}$$

$$\{\lambda s. P (f s) s\} \text{ gets } f \{P\} \quad \{ \quad \} \text{ modify } f \{P\}$$

$$\{ \quad \} \text{ assert } P \{Q\} \quad \{ \quad \} \text{ fail } \{Q\}$$

Hoare Logic over Nondeterministic State Monads



Partial correctness:

$$\{P\} m \{Q\} \equiv \forall s. P s \longrightarrow \forall (r, s') \in \text{fst} (m s). Q r s'$$

→ Post-condition Q is a predicate of return-value and result state.

Weakest Precondition Rules

$$\{\lambda s. P x s\} \text{ return } x \{ \lambda r s. P r s \} \quad \{\lambda s. P s s\} \text{ get } \{P\} \quad \{\lambda s. P () x\} \text{ put } x \{P\}$$

$$\{\lambda s. P (f s) s\} \text{ gets } f \{P\} \quad \{\lambda s. P () (f s)\} \text{ modify } f \{P\}$$

$$\{ \quad \} \text{ assert } P \{Q\} \quad \{ \quad \} \text{ fail } \{Q\}$$

Hoare Logic over Nondeterministic State Monads



Partial correctness:

$$\{P\} m \{Q\} \equiv \forall s. P s \longrightarrow \forall (r, s') \in \text{fst} (m s). Q r s'$$

→ Post-condition Q is a predicate of return-value and result state.

Weakest Precondition Rules

$$\{\lambda s. P x s\} \text{ return } x \{\lambda r s. P r s\} \quad \{\lambda s. P s s\} \text{ get } \{P\} \quad \{\lambda s. P () x\} \text{ put } x \{P\}$$

$$\{\lambda s. P (f s) s\} \text{ gets } f \{P\} \quad \{\lambda s. P () (f s)\} \text{ modify } f \{P\}$$

$$\{\lambda s. P \longrightarrow Q () s\} \text{ assert } P \{Q\} \quad \{\quad\} \text{ fail } \{Q\}$$

Hoare Logic over Nondeterministic State Monads



Partial correctness:

$$\{P\} m \{Q\} \equiv \forall s. P s \longrightarrow \forall (r, s') \in \text{fst} (m s). Q r s'$$

→ Post-condition Q is a predicate of return-value and result state.

Weakest Precondition Rules

$$\{\lambda s. P x s\} \text{ return } x \{\lambda r s. P r s\} \quad \{\lambda s. P s s\} \text{ get } \{P\} \quad \{\lambda s. P () x\} \text{ put } x \{P\}$$

$$\{\lambda s. P (f s) s\} \text{ gets } f \{P\} \quad \{\lambda s. P () (f s)\} \text{ modify } f \{P\}$$

$$\{\lambda s. P \longrightarrow Q () s\} \text{ assert } P \{Q\} \quad \{\lambda _ . \text{True}\} \text{ fail } \{Q\}$$

More Hoare Logic Rules



{ } if P then f else g { S }

More Hoare Logic Rules



$$\frac{P \implies \{Q\} f \{S\} \quad \neg P \implies \{R\} g \{S\}}{\{\lambda s.(P \longrightarrow Q s) \wedge (\neg P \longrightarrow R s)\} \text{ if } P \text{ then } f \text{ else } g \{S\}}$$

More Hoare Logic Rules



$$\frac{P \implies \{Q\} f \{S\} \quad \neg P \implies \{R\} g \{S\}}{\{\lambda s. (P \longrightarrow Q s) \wedge (\neg P \longrightarrow R s)\} \text{ if } P \text{ then } f \text{ else } g \{S\}}$$

$$\frac{\bigwedge x. \{B x\} g x \{C\} \quad \{A\} f \{B\}}{\{A\} \text{ do } \{ x \leftarrow f; g x \} \{C\}}$$

More Hoare Logic Rules



$$\frac{P \implies \{Q\} f \{S\} \quad \neg P \implies \{R\} g \{S\}}{\{\lambda s. (P \longrightarrow Q s) \wedge (\neg P \longrightarrow R s)\} \text{ if } P \text{ then } f \text{ else } g \{S\}}$$

$$\frac{\bigwedge x. \{B x\} g x \{C\} \quad \{A\} f \{B\}}{\{A\} \text{ do}\{ x \leftarrow f; g x \} \{C\}}$$

$$\frac{\{R\} m \{Q\} \quad \bigwedge s. P s \implies R s}{\{P\} m \{Q\}}$$

More Hoare Logic Rules



$$\frac{P \implies \{Q\} f \{S\} \quad \neg P \implies \{R\} g \{S\}}{\{\lambda s. (P \longrightarrow Q s) \wedge (\neg P \longrightarrow R s)\} \text{ if } P \text{ then } f \text{ else } g \{S\}}$$

$$\frac{\bigwedge x. \{B x\} g x \{C\} \quad \{A\} f \{B\}}{\{A\} \text{ do } \{ x \leftarrow f; g x \} \{C\}}$$

$$\frac{\{R\} m \{Q\} \quad \bigwedge s. P s \implies R s}{\{P\} m \{Q\}}$$

$$\frac{\bigwedge r. \{\lambda s. I r s \wedge C r s\} B \{I\} \quad \bigwedge r s. [I r s; \neg C r s] \implies Q r s}{\{I r\} \text{ whileLoop } C B r \{Q\}}$$

A background pattern of white hexagons on a teal background, arranged in a staggered grid.

DATA
61



Demo

We have seen today



We have seen today



→ Deep and shallow embeddings

We have seen today



- Deep and shallow embeddings
- Isabelle records

We have seen today



- Deep and shallow embeddings
- Isabelle records
- Nondeterministic State Monad with Failure

We have seen today



- Deep and shallow embeddings
- Isabelle records
- Nondeterministic State Monad with Failure
- Monadic Weakest Precondition Rules