COMP4161: Advanced Topics in Software Verification

>>=

Gerwin Klein, June Andronick, Christine Rizkallah, Miki Tanaka

S2/2018

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, invariants [8[b],9]
- (mid-semester break)
- C verification [10]
- CakeML, Isar [11[c]]
- Concurrency [12]

[a] a1 due; [b] a2 due; [c] a3 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 $state \Rightarrow state$

$$\text{SKIP} \equiv$$

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 $state \Rightarrow state$

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

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*

$$\text{SKIP} \equiv \lambda s.\ s$$
$$\text{IF b THEN c ELSE d} \equiv$$

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*

$$\begin{array}{rl} \mathsf{SKIP} \equiv & \lambda \mathsf{s.}\ \mathsf{s} \\ \mathsf{IF\ b\ THEN\ c\ ELSE\ d} \equiv & \lambda \mathsf{s.}\ \mathsf{if\ b\ s\ then\ c\ s\ else\ d\ s} \end{array}$$

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* ⇒ *state*

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

- ➜ "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*

$$\begin{aligned} \text{SKIP} &\equiv \quad \lambda\text{s. s} \\ \text{IF b THEN c ELSE d} &\equiv \quad \lambda\text{s. if b s then c s else d s} \end{aligned}$$

- ➜ "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:

$$\textbf{record } A = \quad \begin{array}{l} a :: \text{nat} \\ b :: \text{int} \end{array}$$

Records in Isabelle

Records are a tuples with named components

Example:

$$\textbf{record}\ \mathsf{A} = \quad \mathsf{a} :: \mathsf{nat}$$
$$\mathsf{b} :: \mathsf{int}$$

➜ Selectors: $\mathsf{a} :: \mathsf{A} \Rightarrow \mathsf{nat}$, $\mathsf{b} :: \mathsf{A} \Rightarrow \mathsf{int}$, $\mathsf{a}\ r = \mathsf{Suc}\ 0$

Records in Isabelle

Records are a tuples with named components

Example:

$$\textbf{record}\ \mathsf{A} = \quad \mathsf{a} :: \mathsf{nat} \qquad \mathsf{b} :: \mathsf{int}$$

- ➜ Selectors: $\mathsf{a} :: \mathsf{A} \Rightarrow \mathsf{nat}$, $\mathsf{b} :: \mathsf{A} \Rightarrow \mathsf{int}$, $\mathsf{a}\ r = \mathsf{Suc}\ 0$
- ➜ Constructors: $(\!|\ \mathsf{a} = \mathsf{Suc}\ 0,\ \mathsf{b} = -1\ |\!)$

Records in Isabelle

Records are a tuples with named components

Example:

$$\textbf{record}\ \mathsf{A} = \quad \mathsf{a} :: \mathsf{nat} \qquad \mathsf{b} :: \mathsf{int}$$

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

Records in Isabelle

Records are a tuples with named components

Example:

$$\textbf{record}\ \mathsf{A} = \quad \mathsf{a} :: \mathsf{nat}$$
$$\mathsf{b} :: \mathsf{int}$$

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

Records are extensible:

$$\textbf{record}\ \mathsf{B} = \mathsf{A}\ +$$
$$\mathsf{c} :: \mathsf{nat\ list}$$

Records in Isabelle

Records are a tuples with named components

Example:

$$\textbf{record } A = \quad a :: \text{nat} \quad b :: \text{int}$$

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

Records are extensible:

$$\textbf{record } B = A + \quad c :: \text{nat list}$$

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

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:

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

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:

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

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:

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

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:

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

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:

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

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 **M** is a type constructor with two operations.

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

Monads, Laws

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

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

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

Monads, Laws

Formally: a monad **M** is a type constructor with two operations.

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

Infix Notation: $a \gg\!= b$ is infix notation for 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 **M** is a type constructor with two operations.

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

Infix Notation: $a \gg= b$ is infix notation for 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 **M** is a type constructor with two operations.

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

Infix Notation: $a \gg= b$ is infix notation for 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: $(\text{return}\ x \gg= f) \quad = \quad f\ x$

Monads, Laws

Formally: a monad **M** is a type constructor with two operations.

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

Infix Notation: $a \gg= b$ is infix notation for 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: $(\text{return}\ x \gg= f) \quad = \quad f\ x$

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

Monads, Laws

Formally: a monad **M** is a type constructor with two operations.

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

Infix Notation: $a \gg= b$ is infix notation for 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: $(\text{return}\ x \gg= f) \quad = \quad f\ x$

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

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

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 (λs. hp s p);
  if x $<$ 10 **then**
    modify (hp_update (λh. (h(p := x + 1))))
  else
    return ()
}

State Monad with Failure

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

State Monad with Failure

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

bind – fails when either computation fails

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

State Monad with Failure

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

bind – fails when either computation fails

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

fail – the computation that always fails:

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

State Monad with Failure

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

bind – fails when either computation fails

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

fail – the computation that always fails:

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

assert – fails when given condition is False:

assert P $\equiv$ **if** P **then** return () **else** fail

State Monad with Failure

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

bind – fails when either computation fails

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

fail – the computation that always fails:

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

assert – fails when given condition is False:

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

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

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

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.

$$
\begin{array}{l}
\mathsf{f}\ p\ \equiv \\
\quad \textbf{do}\ \{ \\
\quad\quad \mathsf{y} \leftarrow \mathsf{guard}\ (\lambda \mathsf{s.\ valid\ s\ p}); \\
\quad\quad \mathsf{x} \leftarrow \mathsf{gets}\ (\lambda \mathsf{s.\ hp\ s\ p}); \\
\quad\quad \textbf{if}\ \mathsf{x} < 10\ \textbf{then} \\
\quad\quad\quad \mathsf{modify}\ (\mathsf{hp\_update}\ (\lambda \mathsf{h.}\ (\mathsf{h}(\mathsf{p} := \mathsf{x} + 1)))) \\
\quad\quad \textbf{else} \\
\quad\quad\quad \mathsf{return}\ () \\
\quad \}
\end{array}
$$

Nondeterministic State Monad with Failure

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

Nondeterministic State Monad with Failure

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

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

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:

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

Nondeterministic State Monad with Failure

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

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:

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

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

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

➜ Others are similar.

Nondeterministic State Monad with Failure

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

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:

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

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

- ➜ e.g. return $x \equiv \lambda \text{s}.\ (\{(x,s)\},\text{False})$
- ➜ Others are similar.

select – nondeterministic selection from a set:

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

Demo

While Loops

Monadic while loop, defined **inductively**.

While Loops

Monadic while loop, defined **inductively**.

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

While Loops

Monadic while loop, defined **inductively**.

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

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 paramter
- **fails** if the loop body ever fails or if the loop never terminates

While Loops

Monadic while loop, defined **inductively**.

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

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 paramter
- ➜ **fails** if the loop body ever fails or if the loop never terminates

Example: whileLoop ($\lambda p\ s.$ hp $s\ p = 0$) (λp. return (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: 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: 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: while_results :: $('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow$
$('a \Rightarrow ('s \Rightarrow ('a \times 's)\ \text{set} \times \text{bool})) \Rightarrow$
$((('a \times 's)\ \text{option}) \times (('a \times 's)\ \text{option}))\ \text{set}$

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

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

Defining While Loops Inductively

Two-part definition: results and termination

Results: 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:

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

Defining While Loops Inductively

Termination:

$$\text{while\_terminates} :: (\textit{'a} \Rightarrow \textit{'s} \Rightarrow \text{bool}) \Rightarrow (\textit{'a} \Rightarrow (\textit{'s} \Rightarrow (\textit{'a} \times \textit{'s})\ \text{set} \times \text{bool})) \Rightarrow \textit{'a} \Rightarrow \textit{'s} \Rightarrow \text{bool}$$

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

Defining While Loops Inductively

Termination:

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

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

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

Defining While Loops Inductively

Termination:

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

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

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

$$\begin{aligned}&\text{whileLoop}\ C\ B \equiv\\ &\ (\lambda r\ s.\ (\{(r',s').\ (\text{Some}\ (r,\ s),\ \text{Some}\ (r',\ s')) \in \text{while\_results}\ C\ B\},\\ &\qquad (\text{Some}\ (r,\ s),\ \text{None}) \in \text{while\_results} \lor\\ &\qquad \neg\text{while\_terminates}\ C\ B\ r\ s))\end{aligned}$$

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

$\{\!|\qquad|\!\}$ return x $\{\!|\lambda r\ s.\ P\ r\ s|\!\}$ $\quad\{\!|\qquad|\!\}$ get $\{\!|P|\!\}$ $\quad\{\!|\qquad|\!\}$ put x $\{\!|P|\!\}$

$\{\!|\qquad|\!\}$ gets f $\{\!|P|\!\}$ $\quad\{\!|\qquad|\!\}$ modify f $\{\!|P|\!\}$

$\{\!|\qquad|\!\}$ assert P $\{\!|Q|\!\}$ $\quad\{\!|\qquad|\!\}$ 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

$\{\!|\lambda s.\ P\ x\ s|\!\}\ \text{return}\ x\ \{\!|\lambda r\ s.\ P\ r\ s|\!\}$ $\{\!| \qquad |\!\}\ \text{get}\ \{\!|P|\!\}$ $\{\!| \qquad |\!\}\ \text{put}\ x\ \{\!|P|\!\}$

$\{\!| \qquad |\!\}\ \text{gets f}\ \{\!|P|\!\}$ $\{\!| \qquad |\!\}\ \text{modify}\ f\ \{\!|P|\!\}$

$\{\!| \qquad |\!\}\ \text{assert}\ P\ \{\!|Q|\!\}$ $\{\!| \qquad |\!\}\ \text{fail}\ \{\!|Q|\!\}$

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

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

$\{\!|\quad|\!\}$ gets f $\{\!|P|\!\}$ $\{\!|\quad|\!\}$ modify f $\{\!|P|\!\}$

$\{\!|\quad|\!\}$ assert P $\{\!|Q|\!\}$ $\{\!|\quad|\!\}$ 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

$\{\!|\lambda s.\ P\ x\ s|\!\}$ return x $\{\!|\lambda r\ s.\ P\ r\ s|\!\}$ $\{\!|\lambda s.\ P\ s\ s|\!\}$ get $\{\!|P|\!\}$ $\{\!|\lambda s.\ P\ ()\ x|\!\}$ put x $\{\!|P|\!\}$

$\{\!|\qquad\qquad|\!\}$ gets f $\{\!|P|\!\}$ $\{\!|\qquad\qquad|\!\}$ modify f $\{\!|P|\!\}$

$\{\!|\qquad\qquad|\!\}$ assert P $\{\!|Q|\!\}$ $\{\!|\qquad\qquad|\!\}$ 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

$\{\!|\lambda s.\ P\ x\ s|\!\}$ return x $\{\!|\lambda r\ s.\ P\ r\ s|\!\}$ $\{\!|\lambda s.\ P\ s\ s|\!\}$ get $\{\!|P|\!\}$ $\{\!|\lambda s.\ P\ ()\ x|\!\}$ put x $\{\!|P|\!\}$

$\{\!|\lambda s.\ P\ (f\ s)\ s|\!\}$ gets f $\{\!|P|\!\}$ $\{\!| \qquad |\!\}$ modify f $\{\!|P|\!\}$

$\{\!| \qquad |\!\}$ assert P $\{\!|Q|\!\}$ $\{\!| \qquad |\!\}$ fail $\{\!|Q|\!\}$

Hoare Logic over Nondeterministic State Monads

Partial correctness:

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

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

Weakest Precondition Rules

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

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

$\{\!|\qquad|\!\}$ assert P $\{\!|Q|\!\}$ $\quad$ $\{\!|\qquad|\!\}$ 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

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

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

$\{\!|\lambda s.\ P \longrightarrow Q\ ()\ s|\!\}$ assert P $\{\!|Q|\!\}$ $\quad$ $\{\!| \qquad |\!\}$ 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

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

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

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

More Hoare Logic Rules

$$\frac{}{\{\!|\qquad\qquad\qquad\qquad\qquad|\!\}\ \textbf{if}\ P\ \textbf{then}\ f\ \textbf{else}\ g\ \{\!|S|\!\}}$$

More Hoare Logic Rules

$$\frac{P \Longrightarrow \{\!|Q|\!\}\ f\ \{\!|S|\!\} \quad \neg\ P \Longrightarrow \{\!|R|\!\}\ g\ \{\!|S|\!\}}{\{\!|\lambda s.(P \longrightarrow Q\ s) \wedge (\neg P \longrightarrow R\ s)|\!\}\ \textbf{if}\ P\ \textbf{then}\ f\ \textbf{else}\ g\ \{\!|S|\!\}}$$

More Hoare Logic Rules

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

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

More Hoare Logic Rules

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

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

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

More Hoare Logic Rules

$$\frac{P \Longrightarrow \{\!|Q|\!\}\ f\ \{\!|S|\!\} \quad \neg\ P \Longrightarrow \{\!|R|\!\}\ g\ \{\!|S|\!\}}{\{\!|\lambda s.(P \longrightarrow Q\ s) \wedge (\neg P \longrightarrow R\ s)|\!\}\ \textbf{if}\ P\ \textbf{then}\ f\ \textbf{else}\ g\ \{\!|S|\!\}}$$

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

$$\frac{\{\!|R|\!\}\ m\ \{\!|Q|\!\} \quad \bigwedge s.\ P\ s \Longrightarrow R\ s}{\{\!|P|\!\}\ m\ \{\!|Q|\!\}}$$

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

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