Automated Algebraic Reasoning for Collections and Local Variables with Lenses

Simon Foster James Baxter

University of York

26th October 2020

Unifying Theories of Programming (UTP)

- framework for formulation of denotational semantic models
- based on the idea of programs-as-predicates
- predicates encode the set of observable behaviours
- alphabet gives the domain of possible observations
- alpha predicates $(\alpha P, P)$ over input, output variables (x / x')
- alphabetised relational calculus – models expressed as relations

Unifying Theories of Programming (UTP)

- framework for formulation of denotational semantic models
- based on the idea of programs-as-predicates
- predicates encode the set of observable behaviours
- alphabet gives the domain of possible observations
- alpha predicates $(\alpha P, P)$ over input, output variables $(x \ / \ x')$
- alphabetised relational calculus – models expressed as relations

$$
\begin{aligned}
x := v &\triangleq x' = v \land y' = y \\
P \ ; \ Q &\triangleq \exists\, x_0 \bullet P[x_0/x'] \land Q[x_0/x] \\
P \lhd b \rhd Q &\triangleq (b \land P) \lor (\neg b \land Q) \\
P^* &\triangleq \nu X \bullet \mathit{II} \lor (P \ ; \ X)
\end{aligned}
$$

Unifying Theories of Programming (UTP)

- framework for formulation of denotational semantic models
- based on the idea of programs-as-predicates
- predicates encode the set of observable behaviours
- alphabet gives the domain of possible observations
- alpha predicates $(\alpha P, P)$ over input, output variables $(x \ / \ x')$
- alphabetised relational calculus – models expressed as relations

$$
\begin{aligned}
x := v &\triangleq x' = v \land y' = y \\
P \ ; \ Q &\triangleq \exists\, x_0 \bullet P[x_0/x'] \land Q[x_0/x] \\
P \lhd b \rhd Q &\triangleq (b \land P) \lor (\neg b \land Q) \\
P^* &\triangleq \nu X \bullet \mathit{II} \lor (P \ ; \ X)
\end{aligned}
$$

- avoids fixing a language's abstract syntax tree
- enables composition of semantic models (via UTP theories)

Isabelle/UTP and Alphabet Modelling

- mechanisation of UTP in the Isabelle proof assistant
- representation of alphabets is important

Isabelle/UTP and Alphabet Modelling

- mechanisation of UTP in the Isabelle proof assistant
- representation of alphabets is important
- naïve approach: alphabets as sets of variable names
  - need to check variables are in the set – complicates proof
  - need syntactic meta-logic operators (freshness, substitution)

Isabelle/UTP and Alphabet Modelling

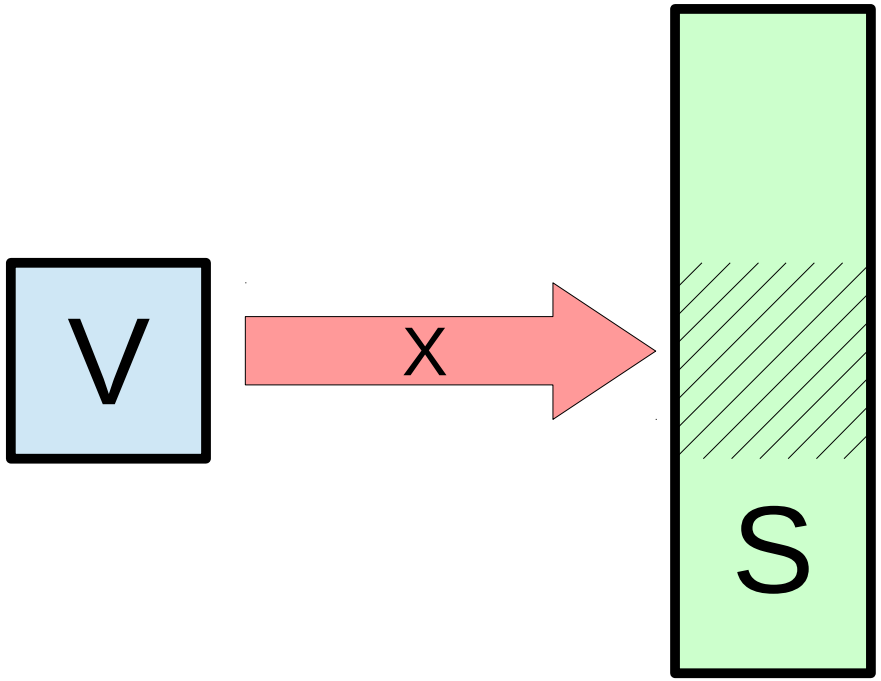
- mechanisation of UTP in the Isabelle proof assistant
- representation of alphabets is important
- naïve approach: alphabets as sets of variable names
  - need to check variables are in the set – complicates proof
  - need syntactic meta-logic operators (freshness, substitution)
- expressing alphabets in the type system loses expressivity

Isabelle/UTP and Alphabet Modelling

- mechanisation of UTP in the Isabelle proof assistant
- representation of alphabets is important
- naïve approach: alphabets as sets of variable names
  - need to check variables are in the set – complicates proof
  - need syntactic meta-logic operators (freshness, substitution)
- expressing alphabets in the type system loses expressivity
- solution: lenses
  - uniform semantic interface for variables
  - identify variables by the position they occupy in the state

What is a lens?

- $X : V \Longrightarrow S$ for view type V and ("bigger") source type S
- allow to focus on V independently of rest of S



What is a lens?

- $X : V \Longrightarrow S$ for view type V and ("bigger") source type S
- allow to focus on V independently of rest of S
- a lens is a quadruple $\langle \mathcal{V} \mid \mathcal{S} \mid \mathit{get} : \mathcal{S} \to \mathcal{V} \mid \mathit{put} : \mathcal{S} \to \mathcal{V} \to \mathcal{S} \rangle$
  - view type $\mathcal{V}$ and source type $\mathcal{S}$ are non-empty
  - *get* and *put* are total functions
  - for $X : V \Longrightarrow S$, write get_X and put_X

What is a lens?

- $X : V \Longrightarrow S$ for view type V and ("bigger") source type S
- allow to focus on V independently of rest of S
- a lens is a quadruple $\langle \mathcal{V} \mid \mathcal{S} \mid get : \mathcal{S} \to \mathcal{V} \mid put : \mathcal{S} \to \mathcal{V} \to \mathcal{S} \rangle$
  - view type $\mathcal{V}$ and source type $\mathcal{S}$ are non-empty
  - *get* and *put* are total functions
  - for $X : V \Longrightarrow S$, write get_X and put_X
- characterised through a number of algebraic laws

$$get\,(put\,s\,v) = v \qquad \text{(PutGet)}$$

$$put\,(put\,s\,v')\,v = put\,s\,v \qquad \text{(PutPut)}$$

$$put\,s\,(get\,s) = s \qquad \text{(GetPut)}$$

What is a lens?

- $X : V \Longrightarrow S$ for view type V and ("bigger") source type S
- allow to focus on V independently of rest of S
- a lens is a quadruple $\langle \mathcal{V} \mid \mathcal{S} \mid \mathit{get} : \mathcal{S} \to \mathcal{V} \mid \mathit{put} : \mathcal{S} \to \mathcal{V} \to \mathcal{S} \rangle$
  - view type $\mathcal{V}$ and source type $\mathcal{S}$ are non-empty
  - *get* and *put* are total functions
  - for $X : V \Longrightarrow S$, write get_X and put_X
- characterised through a number of algebraic laws

$$\mathit{get}\,(\mathit{put}\,s\,v) = v \qquad \text{(PutGet)}$$

$$\mathit{put}\,(\mathit{put}\,s\,v')\,v = \mathit{put}\,s\,v \qquad \text{(PutPut)}$$

$$\mathit{put}\,s\,(\mathit{get}\,s) = s \qquad \text{(GetPut)}$$

  - PutGet + PutPut characterise partial lenses
  - addition of GetPut characterises total lenses
  - partial lenses are motivated by partial structures such as lists

What is a lens?

- $X : V \Longrightarrow S$ for view type V and ("bigger") source type S
- allow to focus on V independently of rest of S
- a lens is a quadruple $\langle \mathcal{V} \mid \mathcal{S} \mid get : \mathcal{S} \to \mathcal{V} \mid put : \mathcal{S} \to \mathcal{V} \to \mathcal{S} \rangle$
  - view type $\mathcal{V}$ and source type $\mathcal{S}$ are non-empty
  - *get* and *put* are total functions
  - for $X : V \Longrightarrow S$, write get_X and put_X
- characterised through a number of algebraic laws

$$get\,(put\,s\,v) = v \qquad \text{(PutGet)}$$

$$put\,(put\,s\,v')\,v = put\,s\,v \qquad \text{(PutPut)}$$

$$put\,s\,(get\,s) = s \qquad \text{(GetPut)}$$

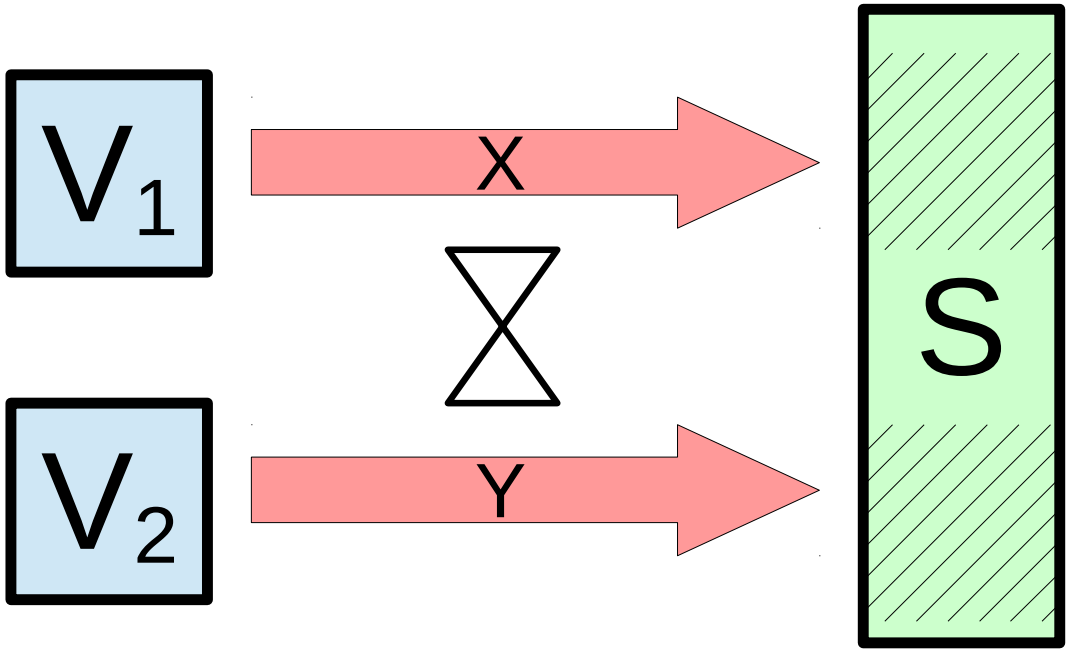
  - PutGet + PutPut characterise partial lenses
  - addition of GetPut characterises total lenses
  - partial lenses are motivated by partial structures such as lists
- related to Back and Von Wright's variable functions

Lens Examples

- pairs – $\mathbf{fst}^A_B : A \Longrightarrow A \times B$ and $\mathbf{snd}^A_B : B \Longrightarrow A \times B$
- records – a total lens viewing each field
- functions – $\mathit{fun\text{-}lens}_B(x) : B \Longrightarrow (A \to B)$ for each $x \in A$
  - views the value of the function at x
  - partial function lens – *get* has arbitrary value outside domain
- lists – similar to partial functions
  - *put* extends the list with arbitrary elements if necessary
  - there is also a (partial) tail lens – views the tail of a list

Lens independence

- $X : V_1 \Longrightarrow S$ and $Y : V_2 \Longrightarrow S$ are independent ($X \bowtie Y$) if they view separate areas of the source



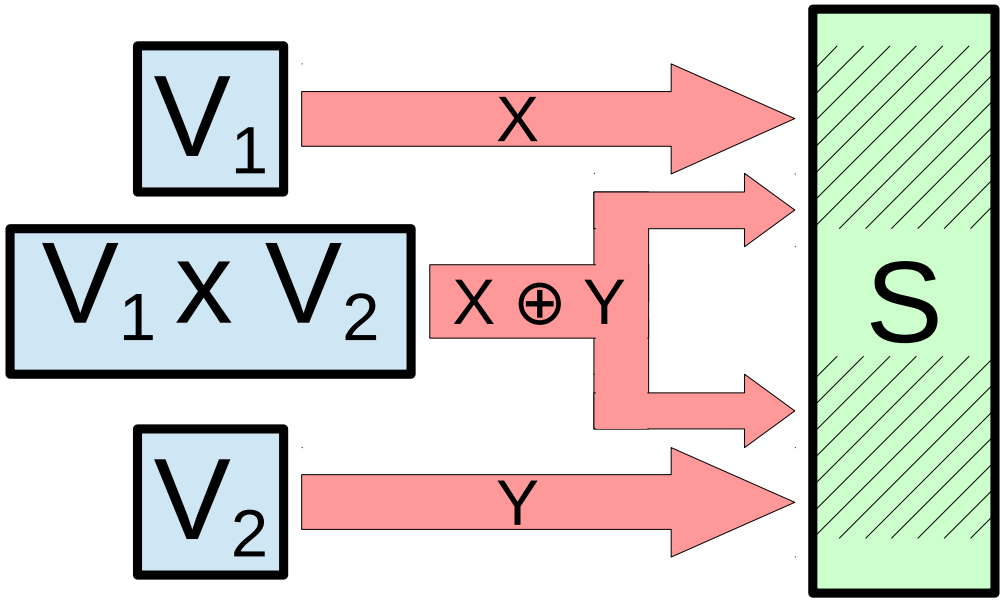
$$put_X(put_Y\ s\ v)\ u = put_Y(put_X\ s\ u)\ v$$

$$get_X(put_Y\ s\ v) = get_X\ s$$

$$get_Y(put_X\ s\ u) = get_Y\ s$$

Lens sum

- $X \oplus Y$ parallel composes two independent lenses



$$\begin{aligned} \mathit{fst} ⨟ (X \oplus Y) &= X & X \bowtie Y \\ \mathit{snd} ⨟ (X \oplus Y) &= Y & X \bowtie Y \\ \mathit{fst} &\bowtie \mathit{snd} \\ \mathit{fst} \oplus \mathit{snd} &= 1 \end{aligned}$$

Modelling state with lenses

- pick a source type to represent the alphabet
  - lenses unify different ways of representing state
- one lens for each variable
- lens view type is the variable's type
- lens sum builds collections of variables
  - e.g. $x \oplus y \oplus z$ represents $\{x, y, z\}$
- lens to view the whole state $(\mathbf{1})$
- expressions are functions that use *get* to access variable values
  - the domain of the function is the state space
  - e.g. $1 + x$ represents $s \mapsto 1 + get_x\ s$
  - easy to add operators without fixing expression syntax
- assignment evaluates an expression and updates with *put*
  - e.g. $x := y$ represents a state update $s \mapsto put_x\ s\ (get_y\ s)$

Modelling state with lenses

- we want to model more complex programs:

function *insertion-sort*$(arr : [int]array)$
  var $i, j : nat \bullet$
  for $i := 1$ **to** $(length(arr) - 1)$ **do**
    $j := i$;
    while $(0 < j \wedge arr[j] < arr[j-1])$ **do**
      $(arr[j-1], arr[j]) := (arr[j], arr[j-1])$; $j := j - 1$
    od od

Modelling state with lenses

- we want to model more complex programs:

function *insertion-sort*$(arr : [int]array)$
  var $i, j : nat \bullet$
  for $i := 1$ **to** $(length(arr) - 1)$ **do**
    $j := i$;
    while $(0 < j \wedge arr[j] < arr[j-1])$ **do**
      $(arr[j-1], arr[j]) := (arr[j], arr[j-1])$; $j := j - 1$
    od od

- collections require lenses that depend on expressions
- variable blocks require state space to change

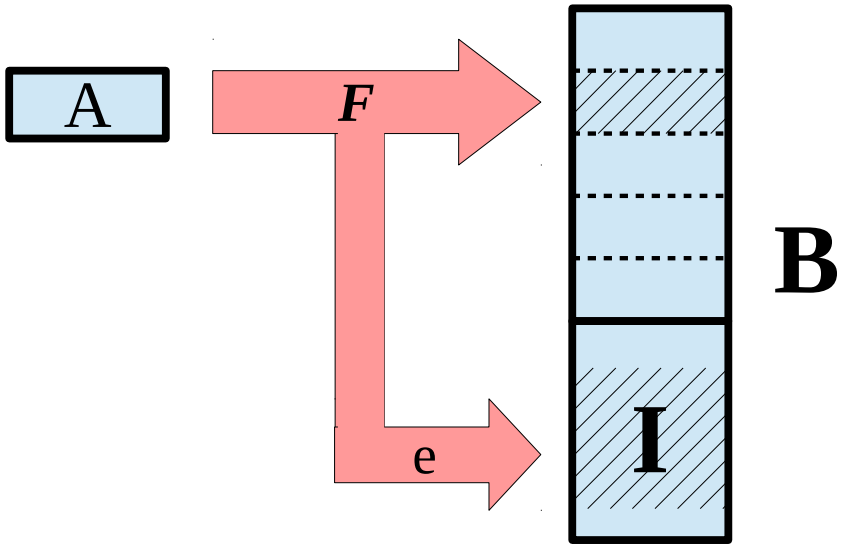
Modelling state with lenses

- we want to model more complex programs:

function *insertion-sort*$(arr : [int]array)$
  var $i, j : nat$ •
  for $i := 1$ **to** $(length(arr) - 1)$ **do**
    $j := i$;
    while $(0 < j \wedge arr[j] < arr[j-1])$ **do**
      $(arr[j-1], arr[j]) := (arr[j], arr[j-1])$; $j := j - 1$
    od od

- collections require lenses that depend on expressions
- variable blocks require state space to change
- can be implemented with dynamic lenses and symmetric lenses

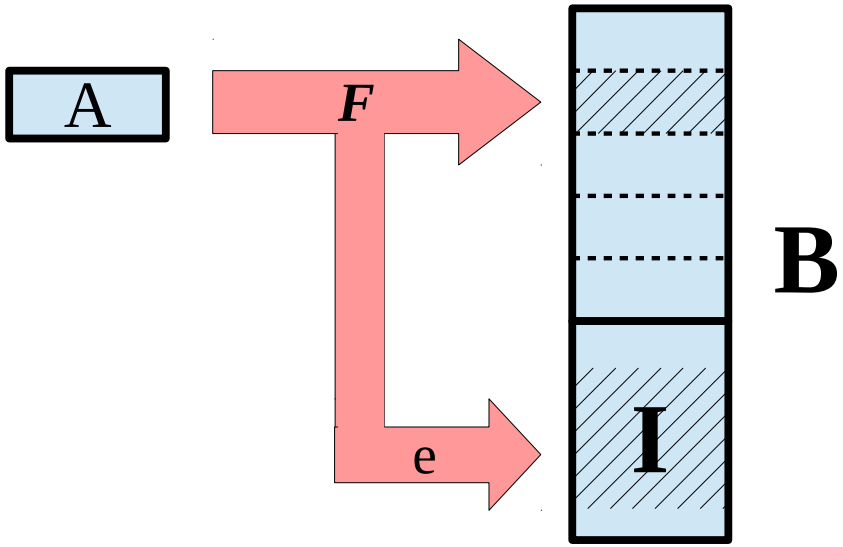
Dynamic Lenses and Collection Lenses



- dynamic lens – given $F : I \to (A \Longrightarrow B)$ and $e : B \to I$,

$$\mathit{dyn\text{-}lens}\ F\ e \mathrel{\widehat{=}} (A, B, \lambda\, s \bullet \mathit{get}_{F(e\ s)}\ s, \lambda\, s\ v \bullet \mathit{put}_{F(e\ s)}\ s\ v)$$

Dynamic Lenses and Collection Lenses

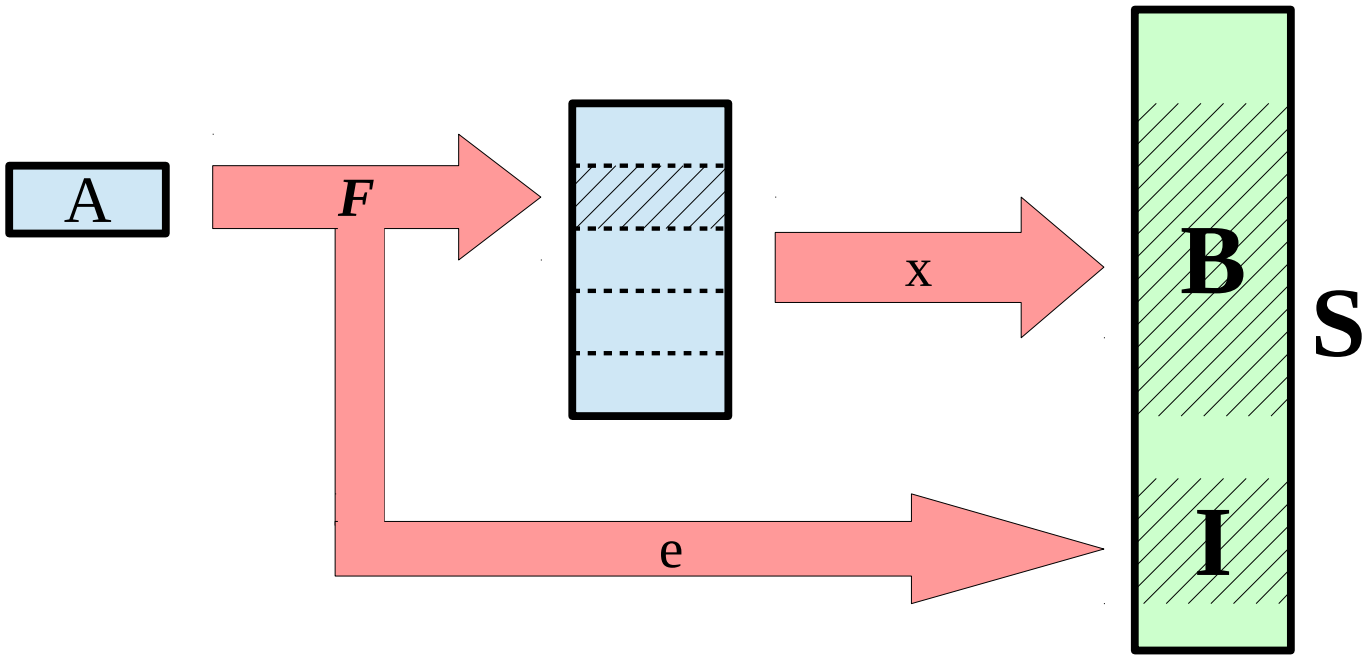


- dynamic lens – given $F : I \to (A \Longrightarrow B)$ and $e : B \to I$,

$$\mathit{dyn\text{-}lens}\ F\ e \mathrel{\widehat{=}} (A, B, \lambda\, s \bullet \mathit{get}_{F(e\ s)}\ s, \lambda\, s\ v \bullet \mathit{put}_{F(e\ s)}\ s\ v)$$

- $\mathit{dyn\text{-}lens}\ F\ e$ is a lens when e does not refer to $F\ i$ for any i
- $\mathit{dyn\text{-}lens}\ F\ e$ is partial when each $F\ i$ is partial

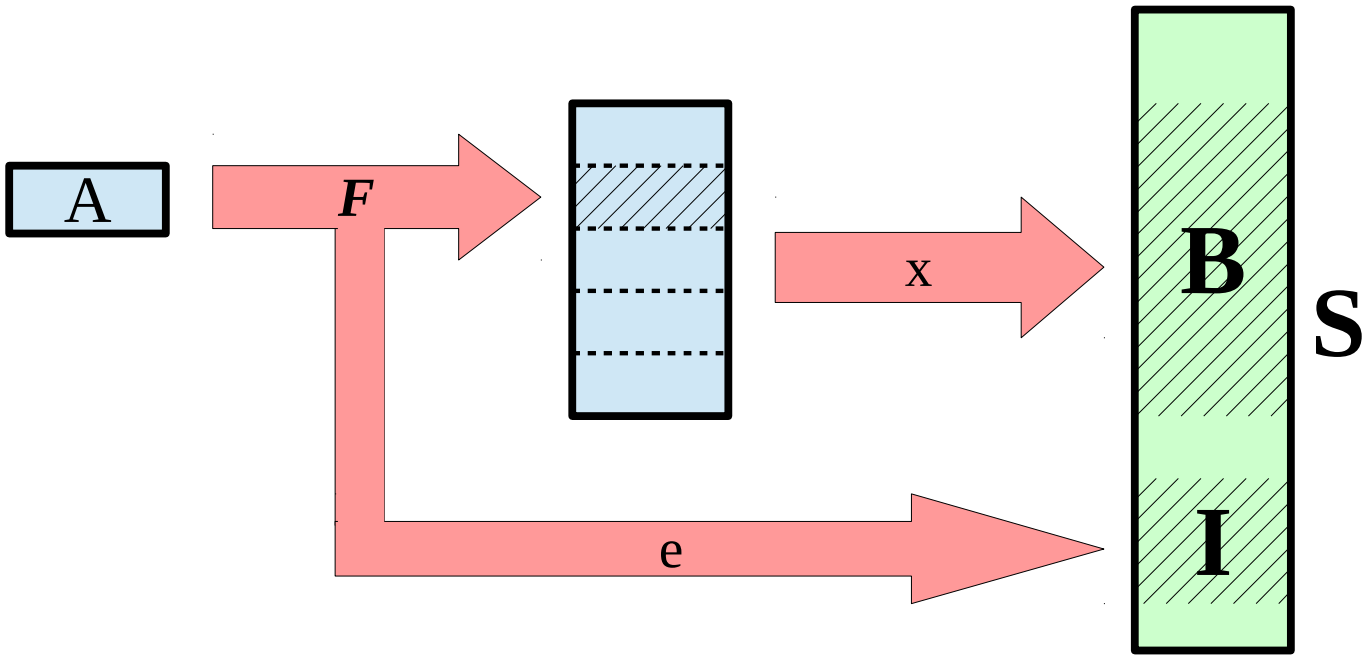
Dynamic Lenses and Collection Lenses



- collection lens – fix $F : I \to (A \Longrightarrow B)$, given $x : B \Longrightarrow S$ and $e : S \to I$,

$$x[e] \mathrel{\widehat{=}} \mathit{dyn\text{-}lens}\,(\lambda\, i : I \bullet F\, i \fatsemi x)\, e$$

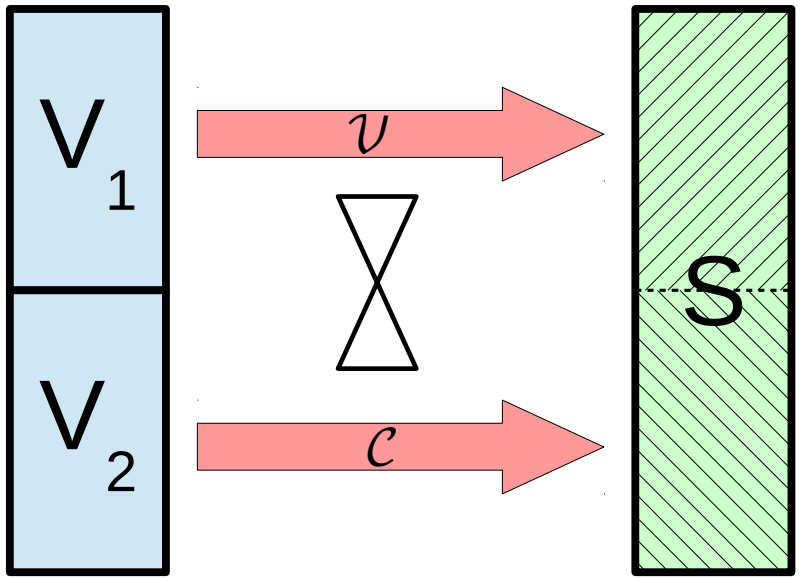
Dynamic Lenses and Collection Lenses



- collection lens – fix $F : I \to (A \Longrightarrow B)$, given $x : B \Longrightarrow S$ and $e : S \to I$,
$$x[e] \mathrel{\widehat{=}} \mathit{dyn\text{-}lens}\,(\lambda\, i : I \bullet F\, i \fatsemi x)\, e$$
- F depends on the type B of the collection identified by x
- example: F total function lens, with $x : (X \to Y) \Longrightarrow S$

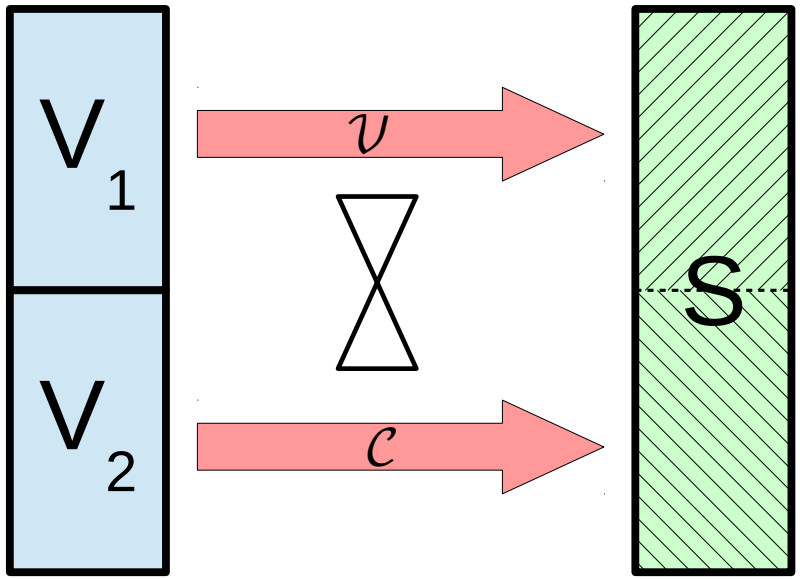
Symmetric Lenses

- symmetric lens – $\mathcal{X} : [V_1, V_2] \Longleftrightarrow [S]$
  - pair of lenses $\mathcal{X} = (\mathcal{V}, \mathcal{C})$ partitioning the state space
  - view lens $\mathcal{V} : V_1 \Longrightarrow S$, coview lens $\mathcal{C} : V_2 \Longrightarrow S$



Symmetric Lenses

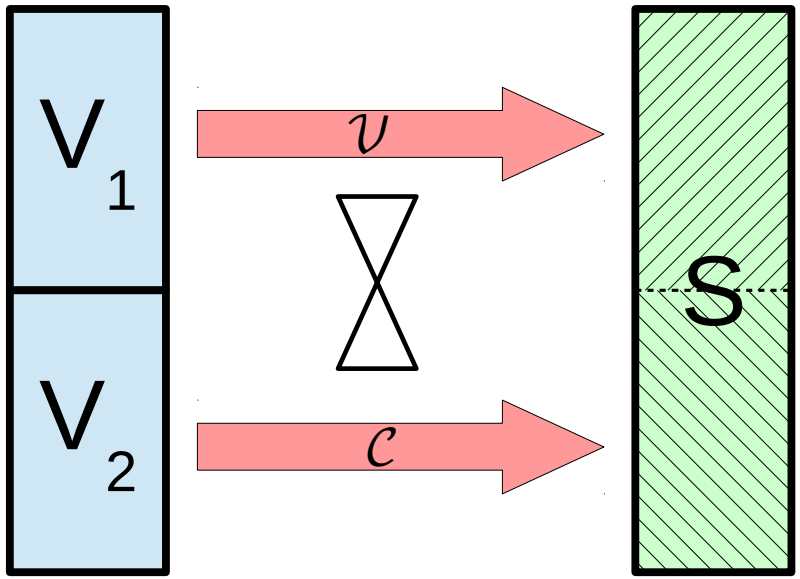
- symmetric lens – $\mathcal{X} : [V_1, V_2] \Longleftrightarrow [S]$
  - pair of lenses $\mathcal{X} = (\mathcal{V}, \mathcal{C})$ partitioning the state space
  - view lens $\mathcal{V} : V_1 \Longrightarrow S$, coview lens $\mathcal{C} : V_2 \Longrightarrow S$



- the view and coview are independent ($\mathcal{V} \bowtie \mathcal{C}$)

Symmetric Lenses

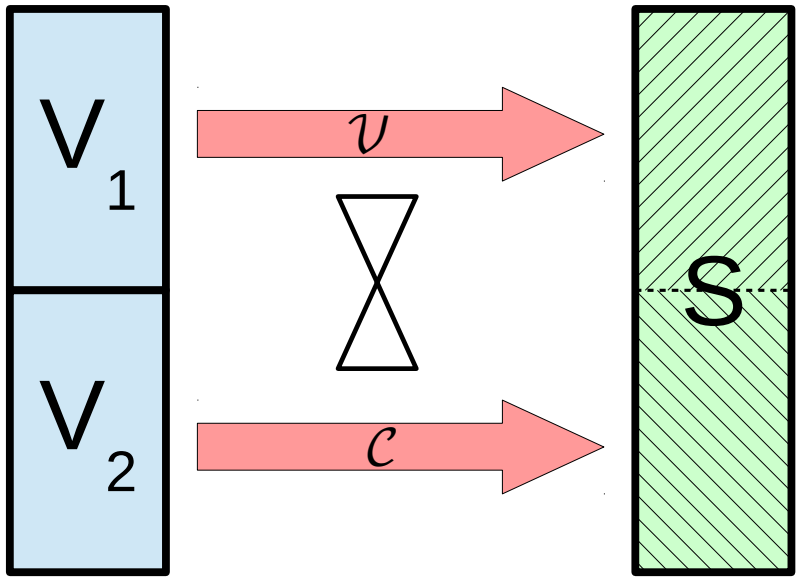
- symmetric lens – $\mathcal{X} : [V_1, V_2] \Longleftrightarrow [S]$
  - pair of lenses $\mathcal{X} = (\mathcal{V}, \mathcal{C})$ partitioning the state space
  - view lens $\mathcal{V} : V_1 \Longrightarrow S$, coview lens $\mathcal{C} : V_2 \Longrightarrow S$



- the view and coview are independent ($\mathcal{V} \bowtie \mathcal{C}$)
- $\mathcal{V} \oplus \mathcal{C}$ covers the whole state space

Symmetric Lenses

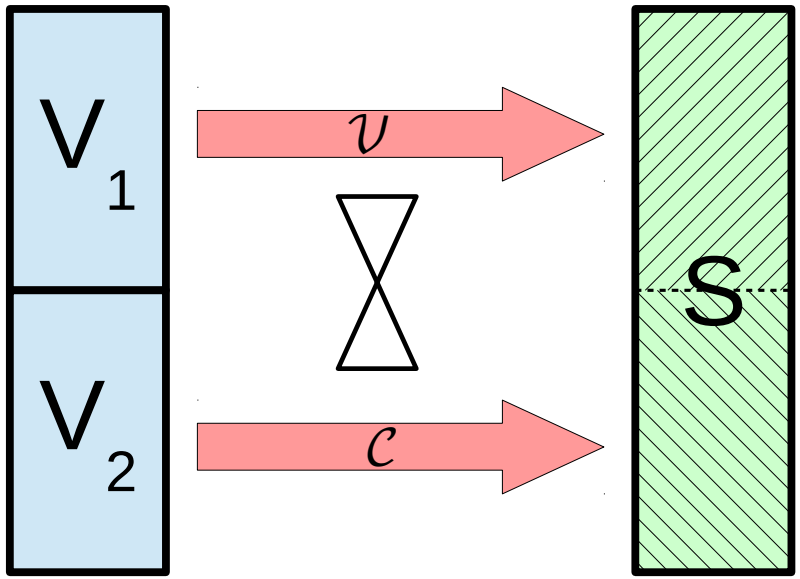
- symmetric lens – $\mathcal{X} : [V_1, V_2] \Longleftrightarrow [S]$
  - pair of lenses $\mathcal{X} = (\mathcal{V}, \mathcal{C})$ partitioning the state space
  - view lens $\mathcal{V} : V_1 \Longrightarrow S$, coview lens $\mathcal{C} : V_2 \Longrightarrow S$



- the view and coview are independent ($\mathcal{V} \bowtie \mathcal{C}$)
- $\mathcal{V} \oplus \mathcal{C}$ covers the whole state space
  - bijective lens – $put\ s\ (get\ s') = s'$, partial: $put\ s\ v = put\ s'\ v$

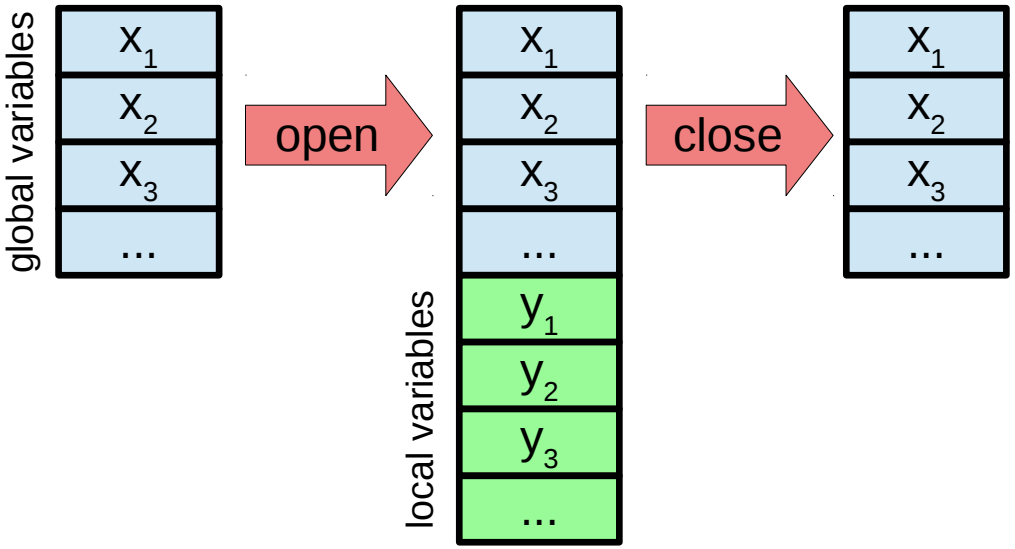
Symmetric Lenses

- symmetric lens – $\mathcal{X} : [V_1, V_2] \Longleftrightarrow [S]$
  - pair of lenses $\mathcal{X} = (\mathcal{V}, \mathcal{C})$ partitioning the state space
  - view lens $\mathcal{V} : V_1 \Longrightarrow S$, coview lens $\mathcal{C} : V_2 \Longrightarrow S$

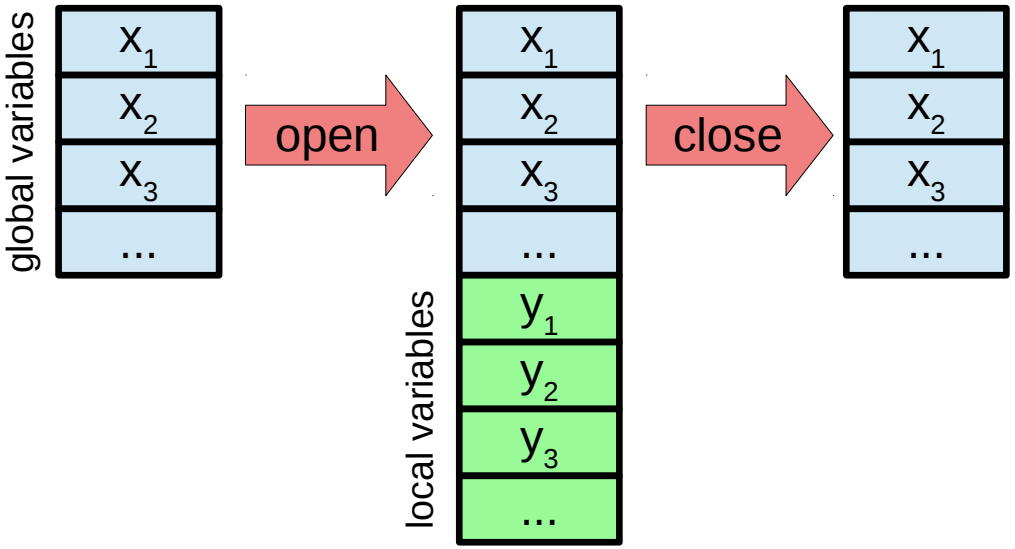


- Example: pair symmetric lens – $(\mathbf{fst}^A_B, \mathbf{snd}^A_B)$
- Example: list partial symmetric lens – head and tail lenses

Symmetric Lenses and Variable Blocks

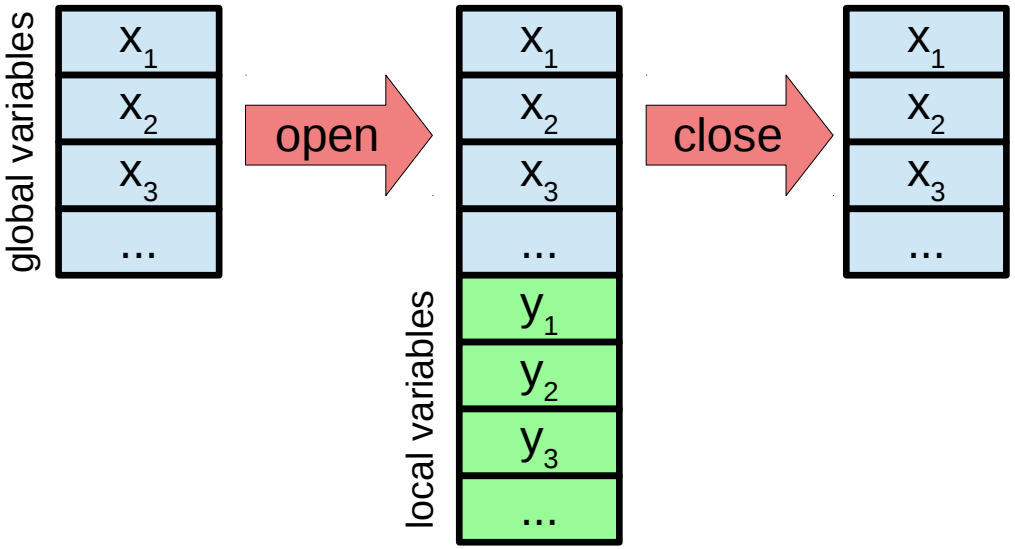


Symmetric Lenses and Variable Blocks



- fix symmetric lens to partition global and local state
  - $\mathcal{X} = (\mathcal{V}_{\mathcal{X}}, \mathcal{C}_{\mathcal{X}}) : [G, L] \Longleftrightarrow [S]$

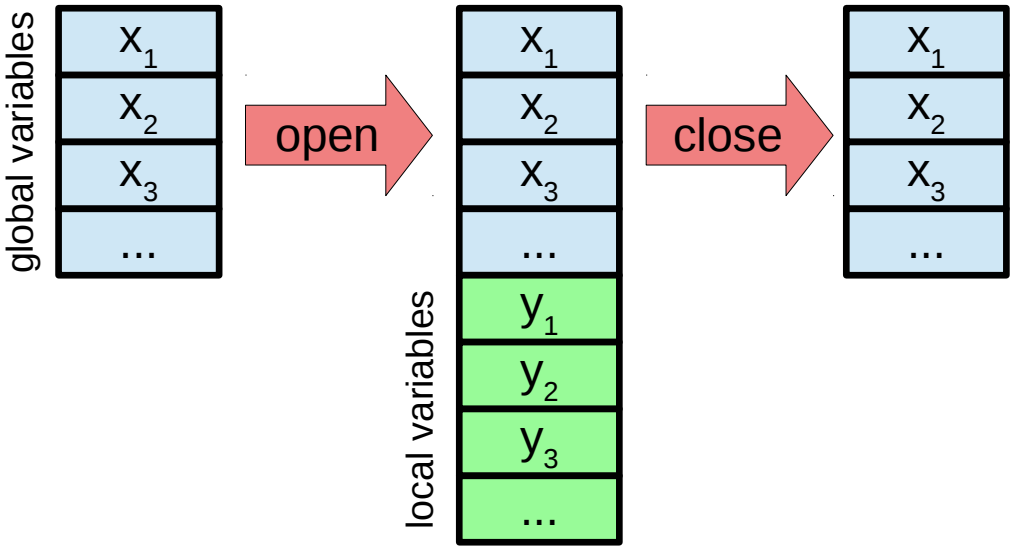
Symmetric Lenses and Variable Blocks



- fix symmetric lens to partition global and local state
  - $\mathcal{X} = (\mathcal{V}_\mathcal{X}, \mathcal{C}_\mathcal{X}) : [G, L] \Longleftrightarrow [S]$
- define state substitutions $\boldsymbol{ext}_\mathcal{X} : G \to S$ and $\boldsymbol{con}_\mathcal{X} : S \to G$

$$\boldsymbol{ext}_\mathcal{X} \mathrel{\widehat{=}} (\!|\mathcal{V}_\mathcal{X} \mapsto \mathbf{1}, \mathcal{C}_\mathcal{X} \mapsto \varepsilon v \bullet v \in L|\!) \qquad \boldsymbol{con}_\mathcal{X} \mathrel{\widehat{=}} (\!|\mathbf{1} \mapsto \mathcal{V}_\mathcal{X}|\!)$$

Symmetric Lenses and Variable Blocks



- fix symmetric lens to partition global and local state
  - $\mathcal{X} = (\mathcal{V}_\mathcal{X}, \mathcal{C}_\mathcal{X}) : [G, L] \Longleftrightarrow [S]$
- define state substitutions $\boldsymbol{ext}_\mathcal{X} : G \to S$ and $\boldsymbol{con}_\mathcal{X} : S \to G$

  $\boldsymbol{ext}_\mathcal{X} \mathrel{\widehat{=}} (\!|\mathcal{V}_\mathcal{X} \mapsto \mathbf{1}, \mathcal{C}_\mathcal{X} \mapsto \varepsilon v \bullet v \in L|\!)$ $\quad$ $\boldsymbol{con}_\mathcal{X} \mathrel{\widehat{=}} (\!|\mathbf{1} \mapsto \mathcal{V}_\mathcal{X}|\!)$
- lift variable block opening and closing operators

  $\boldsymbol{open}_\mathcal{X} \mathrel{\widehat{=}} \langle \boldsymbol{ext}_\mathcal{X} \rangle \, ; \mathcal{C}_\mathcal{X} := *$ $\quad$ $\boldsymbol{close}_\mathcal{X} \mathrel{\widehat{=}} \langle \boldsymbol{con}_\mathcal{X} \rangle$

Variable Block Examples

Variable Block Examples

- pair – $(\boldsymbol{fst}_L^G, \boldsymbol{snd}_L^G) : [G, L] \iff [G \times L]$

Variable Block Examples

- pair – $(\textbf{\textit{fst}}^G_L, \textbf{\textit{snd}}^G_L) : [G, L] \Longleftrightarrow [G \times L]$
- record extension – $(\textbf{\textit{base}}, \textbf{\textit{more}}) : [G, L] \Longleftrightarrow [[L]\, G_{ext}]$
  - works with Isabelle's parametric extensible records
  - allows global variable lenses to be used unmodified

Variable Block Examples

- pair – $(\boldsymbol{fst}_L^G, \boldsymbol{snd}_L^G) : [G, L] \Longleftrightarrow [G \times L]$
- record extension – $(\boldsymbol{base}, \boldsymbol{more}) : [G, L] \Longleftrightarrow [[L]\, G_{ext}]$
  - works with Isabelle's parametric extensible records
  - allows global variable lenses to be used unmodified
- list – $(\boldsymbol{tl}_L, \boldsymbol{hd}_L) : [[L]\, list, L] \Longleftrightarrow [[L]\, list]$
  - represents a stack of local states
  - view lens $\boldsymbol{tl}_L : [L]\, list \Longrightarrow [L]\, list$ doesn't change state type
  - can be used in a recursive setting
  - partial lens may complicate proof

Variable Block Examples

- pair – $(\boldsymbol{fst}^G_L, \boldsymbol{snd}^G_L) : [G, L] \Longleftrightarrow [G \times L]$
- record extension – $(\boldsymbol{base}, \boldsymbol{more}) : [G, L] \Longleftrightarrow [[L]\, G_{ext}]$
  - works with Isabelle's parametric extensible records
  - allows global variable lenses to be used unmodified
- list – $(\boldsymbol{tl}_L, \boldsymbol{hd}_L) : [[L]\, list, L] \Longleftrightarrow [[L]\, list]$
  - represents a stack of local states
  - view lens $\boldsymbol{tl}_L : [L]\, list \Longrightarrow [L]\, list$ doesn't change state type
  - can be used in a recursive setting
  - partial lens may complicate proof
- total function – $(\boldsymbol{tl}^f_L, \boldsymbol{hd}^f_L) : [\mathbb{N} \to L, L] \Longleftrightarrow [\mathbb{N} \to L]$
  - similar to list symmetric lens, works as total lens

Insertion Sort Revisited

function *insertion-sort*$(arr : [int]array)$
  var $i, j : nat \bullet$
  for $i := 1$ **to** $(length(arr) - 1)$ **do**
    $j := i$;
    while $(0 < j \wedge arr[j] < arr[j-1])$ **do**
      $(arr[j-1], arr[j]) := (arr[j], arr[j-1])$; $j := j - 1$
    od od

- variables are outside of the loop – extensible records can be used
- global state lenses can be used unmodified
- access to arr modelled using partial function lens

Conclusion

- UTP – framework for denotational program semantics
- lenses provide an algebraic system for describing UTP variables
- lenses abstract the concept of getting and setting variables
- lenses unify different ways of modelling state
- various lens operators to combine lenses
- lenses into collections can be specified with dynamic lenses
- symmetric lenses allow for definition of variable blocks
- various instances of symmetric lenses
- all mechanised in Isabelle
- future work:
  - further exploration of symmetric lenses e.g. different models
  - relationship of lenses to separation logic