# Part IB Semantics of Programming Languages Notes

## 0.1 Operational semantics

- Set $Config$ of states of the form $\langle e, s \rangle$
- The reduction operator $\rightarrow \subseteq Config \times Config$
- The unary predicate $\nrightarrow$, $c \nrightarrow iff \neg \exists c'.c \rightarrow c'$

## 0.2 Design choices

- Evaluation order, the rules encode left to right evaluation by enforcing the left hand side is a value before evaluating the right
- Result of assignment, an assignment can result in a `skip` or the value being assigned
- Sequence mechanism, $e_1;\ e_2$ can reduce only if $e_1$ is `skip` or if it is any value.
- Store initialisation, L1 requires all locations be defined before execution, but could instead
  - Initialise all locations to zero
  - Allow assignment to undefined locations to initialise them.
- Allow programs to store any value, not only integers
- Allow a local store, not just global

## 0.3 Types

- Have the ternary operator $\Gamma \vdash e : T$, read as the expression $e$ has type $T$ under the assumptions $\Gamma$ about the type of each location that may occur in $e$.
- Types of expression $T ::= int \mid bool \mid unit$ and types of location $T_{loc} ::= intref$
- $\Gamma$ is an element of $TypeEnv$ the set of all possible type environments for L1, a finite partial function from locations to types of locations.
- The typing is syntax directed, each rule from the abstract syntax has one typing rule

# 1 Theorems

## 1.1 Determinacy

if $\langle e, s \rangle \rightarrow \langle e_1, s_1 \rangle$ and $\langle e, s \rangle \rightarrow \langle e_2, s_2 \rangle$ then $\langle e_1, s_1 \rangle = \langle e_2, s_2 \rangle$.

## 1.2 Progress

if $\Gamma \vdash e : T$ and $dom(\Gamma) \subseteq dom(s)$ then either $value(e)$ or there exists an $e',\ s'$ such that $\langle e, s \rangle \rightarrow \langle e', s' \rangle$.

## 1.3 Type preservation

if $\Gamma \vdash e : T$ and $dom(\Gamma) \subseteq dom(s)$ and $\langle e, s \rangle \rightarrow \langle e', s' \rangle$ then $\Gamma \vdash e' : T$ and $dom(\Gamma) \subseteq dom(s')$

### 1.4 Safety

$$\text{if } \Gamma \vdash e : T \text{ and } dom(\Gamma) \subseteq dom(s) \text{ and } \langle e, s\rangle \to^* \langle e', s'\rangle \text{ then either } value(e')$$

$$\text{or there exits an } e'', s'' \text{ such that } \langle e', s'\rangle \to \langle e'', s''\rangle$$

### 1.5 Type inference

$$\text{Given } \Gamma, e \text{ find a } T \text{ such that } \Gamma \vdash e : T \text{ or show there is none}$$

### 1.6 Decidability of type checking

$$\text{Given } \Gamma, e, T, \text{ one can decide } \Gamma \vdash e : T$$

### 1.7 Uniqueness of typing

$$\text{If } \Gamma \vdash e : T \text{ and } \Gamma \vdash e : T' \text{ then } T = T'$$

## 2 Induction

### 2.1 Mathematical induction

- Used to prove facts about the natural numbers
- To prove a property $\Phi(x)$ for all natural numbers prove $\Phi(0)$ and $\Phi(n) \implies \Phi(n+1)$

### 2.2 Structural induction

- Prove facts about all terms of a grammar
- Prove that for every tree constructor, $c$, if the property, $\Phi$, holds for all subtrees, $e_1, ..., e_n$, it holds for that constructor.
- For example prove over the syntax of the language

### 2.3 Rule induction

- Prove facts about all elements of a relation defined by rules.
- For a rule

$$\frac{c}{h_1, ..., h_n}$$

  you must prove that for every rule assuming the property holds for the hypotheses, $h_1, ..., h_n$, implies that the property, $\Phi$, holds for the conclusion, $c$.

- For example prove over the reduction or the typing.

### 2.4 Inversion lemma

- In proofs involving multiple inductive definitions need an inversion property
- Used to determine the last rule used, for example this could be for the reduction or typing relation.

# 3 Functions

- Function application is left associative $e_1\, e_2\, e_3 = (e_1\, e_2)\, e_3$
- Type arrows are right associative $T_1 \to T_2 \to T_3 = T_1 \to (T_2 \to T_3)$
- $fn$ extends as far right as parentheses permit
- Functions have the form $fn\ x : T => e$
- Outside the $fn$ binding the name of the formal parameter does not matter.
- Within the $fn$ binding anything with that name (which isn't within another $fn\ x : T' => e$ or a binder) takes it's value.
- Function application represented using @ in abstract syntax tree
- Function application operational semantics

$$\langle (fn\ x : T => e)\, v, s\rangle \to \langle \{v/x\}e, s\rangle$$

## 3.1 Alpha equivalence

- At anytime a parameter can be renamed as long as this doesn't change the binding graph

$$fn\ x : T => x + z = fn\ y : T => y + z \neq fn\ z : T => z + z$$

- This is called *working up to alpha conversion*
- De Bruijn indices
  - Variable names are replace with natural numbers
  - The numbers represent the number of binders in scope between the binder and instance, eg.

$$fn\ x : int => fn\ y : int => x + y \text{ becomes } fn\ \bullet : int => fn\ \bullet : int => v_0 + v_1$$

## 3.2 Free variables and substitution

- The set of free variables in an expression are those which are not bound
- An expression with no free variables is closed

$$\begin{aligned} fv(x) &= x \\ fv(e_1\ op\ e_2) &= fv(e_1) \cup fv(e_2) \\ fv(fn\ x : T => e) &= fv(e) - \{x\} \end{aligned}$$

- Write substitutions as $\{e/x\}e'$ to mean replace all free occurrences of $x$ in the expression $e'$ with $e$
- In the substitution

$$\{(y+2)/x\}(fn\ y => x + y)$$

  must first alpha rename $y$ in the $fn$ expression

$$\{(y+2)/x\}(fn\ y' => x + y')$$

- A substitution $\sigma$ is a finite partial function from variables to expressions, eg.

$$\begin{aligned} &\sigma(x + y) \\ =&\{(z+2)/x, 3/y\}(x + y) \end{aligned}$$

## 3.3 Function behaviour

### 3.3.1 Call-by-value

Preform the following

1. Reduce the left-hand-side to an $fn$ term
2. Reduce the right-hand-side to a value
3. Replace all occurrences of the parameter with the value

### 3.3.2 Call-by-name

Perform the following

1. Reduce the left-hand-side to an $fn$ term
2. Replace all occurrences of the formal parameter with the argument

### 3.3.3 Call-by-need

- Similar to call by name, by the first time the argument is evaluated all other copies are overwritten
- In a pure language this has the same effect as call-by-name but is more efficient

### 3.3.4 Full beta

- All left-hand-side or right to reduce and anytime
- When the left-hand-side is a $fn$ term apply the right-hand-side
- Allow reductions within lambdas
- In a pure language (without IO, etc.) irrespective of the order of reductions this will always give the same result

### 3.3.5 Normal-order reduction

- Leftmost, outermost variant of full beta

## 3.4 Typing functions

- $\Gamma$ is now an assumption about the types of locations and the types of variables
- $\Gamma$ is made up of $\Gamma_{\text{loc}}$ and $\Gamma_{\text{var}}$
- Write $\Gamma, x : T$ to be $\Gamma$ but in $\Gamma_{\text{var}}$ $x$ maps to $T$, for example in the typing of $fn$

$$\frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash fn\ x : T \implies e : T \to T'}$$

- Progress and type preservation can now be updated to rely on the expression being closed.
- Normalisation - In a sublanguage without while loops and store operations if $\Gamma \vdash e : T$ and e is closed then there does not exist an infinite reduction sequence

$$\langle e_0, \{\}\rangle \to \langle e_1, \{\}\rangle \to \langle e_2, \{\}\rangle \to ...$$

### 3.5 Recursion

- *let val* can be implemented as a function application
- Could add $let\ val\ rec\ x : T = eine'\ end$ where $x$ binds in both $e$ and $e'$ - this causes problems, eg. $let\ val\ rec\ x = (x, x)\ in\ x\ end$
- Specialise *let val rec* for recursive functions only
  - Have the following structure

$$let\ val\ rec\ x : T_1 \to T_2 = (fn\ y : T_1 \implies e_1)\ in\ e_2\ end$$

  - Has the following typing rule (the same as normal let, but defines $x$ in itself

$$\frac{\Gamma, x : T_1 \to T_2, y : T_1 \vdash e_1 : T_2 \qquad \Gamma, x : T_1 \to T_2 \vdash e_2 : T}{\Gamma \vdash let\ val\ rec\ x : T_1 \to T_2 = (fn\ y : T_1 \implies e_1)\ in\ e_2\ end : T}$$

- Both sequences and while loops can be encoded as function applications (recursive in the second case)

## 4 Data

- The Curry-Howard correspondence
  - The typing rules of a pure programming language correspond to the rules for natural deduction

### 4.1 Products

- Have type $T_1 * T_2$ and written as $e * e$
- Design decisions
  - Don't have arbitrary sized tuples, only pairs
  - $(e * e) * e$ is not the same as $e * (e * e)$
  - Have projections #1 $e$ and #2 $e$ instead of pattern matching
  - Don't allow computed projection, $\#e\ e'$, as would not be able to type check
  - Order of evaluations (left-to-right, right-to-left, full-beta, etc.)

### 4.2 Sums

- Can be thought of as ML's

```
datatype ('a, 'b) Sum = inl of 'a | inr of 'b;
```

- Must annotate *inl* and *inr* with types otherwise *inl* 4 could be of type $int + int$, $int + bool$, etc.

$$inl\ e : T_1 + T_2 \text{ and } inr\ e : T_1 + T_2$$

- Use a case statement to unpack a sum type variable

$$case\ e\ of\ inl\ (x_1 : T_1) \implies e_1 \mid inl\ (x_2 : T_2) \implies e_2$$

- A alternative to the type labels is having a type inference algorithm or requiring all sum types be declared and named

## 4.3 Records

- Have the form $\{lab_1 = x_1, ..., lab_k = x_k\}$
- Have the type $\{lab_1 : T_1, ..., lab_k : T_k\}$
- $lab$s are distinct from normal variables
- The value associated with the label $lab_i$ in a record $r$ is retrieved using $\#lab_i\ r$
- Use left-to-right evaluation order, use the following reduction rule to indicate this

$$\frac{\langle e_i, s\rangle \to \langle e_i', s'\rangle}{\langle\{lab_1 = v_1, ..., lab_i = e_i, ..., lab_k = e_k\}, s\rangle \to \langle\{lab_1 = v_1, ..., lab_i = e_i', ..., lab_k = e_k\}, s'\rangle}$$

## 4.4 Mutable store

### 4.4.1 Design choices

- Similar to ML
  - Locations store mutable values
  - Variables immutably store previously calculated value
  - Explicit dereferencing and assignment
- Similar to C
  - Variables are mutable and store previously calculated value
  - Implicit dereferencing

### 4.4.2 References

- Make reference have the type $T\ ref$, for some type $T$
- Locations are values
- Add the keyword $ref$ to create references, eg. $ref\ 4$ will create an int reference to 4

## 4.5 Theorems

### 4.5.1 Well typed store

$$(dom(\Gamma) = dom(s) \land (\forall l \in dom(s).\Gamma(l) = T\ ref \implies \Gamma \vdash s(l) : T)) \implies \Gamma \vdash s$$

### 4.5.2 Progress

if $e$ is closed and $\Gamma \vdash e : T$ and $\Gamma \vdash s$ then either $value(e)$ or there exists an $e',\ s'$ such that $\langle e, s\rangle \to \langle e', s'\rangle$.

### 4.5.3 Type preservation

if $e$ is closed and $\Gamma \vdash e : T$ and $\Gamma \vdash s$ and $\langle e, s\rangle \to \langle e', s'\rangle$ then $e'$ is closed

and for some $\Gamma'$ with disjoint domain to $\Gamma$ we have $\Gamma, \Gamma' \vdash e' : T$ and $\Gamma' \vdash s'$

### 4.5.4 Type safety

if $e$ is closed and $\Gamma \vdash e : T$ and $\Gamma \vdash s$ and $\langle e, s\rangle \to^* \langle e', s'\rangle$ then either $value(e')$

or there exits an $e'', s''$ such that $\langle e', s'\rangle \to \langle e'', s''\rangle$

# 5 Subtyping and Objects

## 5.1 Subtype polymorphism

- Define $T <: T'$ to mean $T$ is a subtype of $T'$
- Define the subsumption typing rule

$$\frac{\Gamma \vdash e : T \qquad T <: T'}{\Gamma \vdash e : T'}$$

- Define the subtype relation (reflexive and transitive)

$$\frac{}{T <: T} \qquad\qquad \frac{T <: T' \qquad T' <: T''}{T <: T''}$$

- Subtyping records
  - Forgetting fields to the right

$$\{lab_1 : T_1, ..., lab_i : T_i, ..., lab_k : T_k\} <: \{lab_1 : T_1, ..., lab_i : T_i\}$$

  - Subtyping within fields

$$\frac{T_1 <: T_1' \qquad ... \qquad T_k <: T_k'}{\{lab_1 : T_1, ..., lab_k : T_k\} <: \{lab_1 : T_1', ..., lab_k : T_k'\}}$$

  - Reordering of records

$$\frac{\pi \text{ a permutation of } 1, ..., k}{\{lab_1 : T_1, ..., lab_k : T_k\} <: \{lab_{\pi(1)} : T_{\pi(1)}, ..., lab_{\pi(k)} : T_{\pi(k)}\}}$$

- Subtyping functions
  - Covariant - When making a type more general the associated type also becomes more general
  - Contravariant - When making a type more general the associated type becomes less general
  - When subtyping functions the inputs become more general, the outputs less
  - "Be liberal in what you accept and conservative in what you produce."
  - More specifically

$$\frac{T_1' <: T_1 \qquad T_2 <: T_2'}{T_1 \to T_2 <: T_1' \to T_2'}$$

  - Input is contravariant and out is covariant
- Subtyping products

$$\frac{T_1 <: T_1' \qquad T_2 <: T_2'}{T_1 * T_2 <: T_1' * T_2'}$$

- Subtyping sums

$$\frac{T_1 <: T_1' \qquad T_2 <: T_2'}{T_1 + T_2 <: T_1' + T_2'}$$

## 5.2 Design choice

- Could add down-casting, but would require dynamic type checking
- Adds flexibility but at the risk of runtime errors

# 6 Concurrency

## 6.1 Problems

- Combinatorial explosion in state spaces, n threads each with 2 states gives $2^n$ states
- Computation nondeterministic, different threads operate at different speeds
- Deadlock, starvation, etc.

## 6.2 Semantics and Types

- Add a new type *proc*, to represent a thread of execution
- Add a new operator, |, to indicate parallel execution
- New typing rules

$$\frac{\Gamma \vdash e : \mathit{unit}}{\Gamma \vdash e : \mathit{proc}} \qquad \frac{\Gamma \vdash e_1 : \mathit{proc} \quad \Gamma \vdash e_2 : \mathit{proc}}{\Gamma \vdash e_1 | e_2 : \mathit{proc}}$$

- New operational semantics

$$\frac{\langle e_1,\, s\rangle \to \langle e_1',\, s'\rangle}{\langle e_1 | e_2,\, s\rangle \to \langle e_1' | e_2,\, s'\rangle} \qquad \frac{\langle e_2,\, s\rangle \to \langle e_2',\, s'\rangle}{\langle e_1 | e_2,\, s\rangle \to \langle e_1 | e_2',\, s'\rangle}$$

- Design choices
  - Threads don't return a value
  - Threads have no identity
  - Termination of a thread can not be observed in the language
  - Threads can not be externally killed
- Need some way of synchronising between threads
  - Add a set of mutexes to the configuration $\langle e, s, m\rangle$
  - Add lock and unlock keywords, lock will only reduce if the mutex is unlocked
  - With multiple mutexes can get deadlock

## 6.3 Thread local semantics

- Currently only have a way to talk about global reductions, need some way of talking about thread local reductions
- Define per-thread $e \xrightarrow{a} e'$

$$a ::= \tau \mid l := n \mid !l = n \mid \mathit{lock}\ m \mid \mathit{unlock}\ m$$

- $a$ records what interacts the reduction has with the store mutexes, $\tau$ indicates no interaction
- When dereferencing

$$(l :=\!!l + 1) \xrightarrow{!l=n} (l := n + 1) \quad \text{for any } n \in \mathbb{Z}$$

### 6.4 Two phase locking

- Associate a mutex, $m_i$, with each location, $l_i$
- Acquire the mutex $m_i$ before accessing its associated location $l_i$
- Acquire and release locks in a properly bracketed way
- Once a lock has been release do not acquire any more
- Acquire locks in increasing order

## 7 Semantic Equivalence

- $C[\_] = e$ is a context, eg. $C[\_] = \_ + 1$ would give $C[(l := 1; 4)] = (l := 1; 4) + 1$
- To determine the contexts for a language examine the operational semantics rules and replace each subexpression
- $e_1 \simeq e_2$ means that the expression $e_1$ is equivalent to $e_2$
- Definition of $\simeq$
  - Programs which result in different values must not be equivalent
  - Programs which terminate must not be equivalent to those which do not
  - Must be an equivalence relation (symmetric, transitive, and reflexive)
  - For any context $C[\_]$ must have that $e_1 \simeq e_2 \implies C[e_1] \simeq C[e_2]$
  - Must relate to as many programs as possible subject to the above
  - For typed L1 define $e_1 \simeq^T_\Gamma e_2$ to hold iff forall $s$ such that $dom(\Gamma) \subseteq dom(s)$ we have $\Gamma \vdash e_1 : T$ and $\Gamma \vdash e_2 : T$ and either
    - $\langle e_1, s\rangle \to^\omega$ and $\langle e_2, s\rangle \to^\omega$, or
    - $\langle e_1, s\rangle \to \langle v, s'\rangle$ and $\langle e_2, s\rangle \to \langle v, s'\rangle$
- General laws
  1. $e_1; (e_2; e_3) \simeq (e_1; e_2); e_3$ for any $\Gamma, T, e_1, e_2$, and $e_3$ such that $\Gamma \vdash e_1 : \mathit{unit}$, $\Gamma \vdash e_2 : \mathit{unit}$, and $\Gamma \vdash e_3 : T$
  2. (if $e_1$ then $e_2$ else $e_3$); $e \simeq$ if $e_1$ then $e_2; e$ else $e_3; e$ for any $\Gamma, T, e_1, e_2$, and $e_3$ such that $\Gamma \vdash e_1 : \mathit{bool}$, $\Gamma \vdash e_2 : \mathit{unit}$, $\Gamma \vdash e_3 : \mathit{unit}$, and $\Gamma \vdash e : T$
  3. $e$; (if $e_1$ then $e_2$ else $e_3$) $\simeq$ if $e_1$ then $e; e_2$ else $e; e_3$ for any $\Gamma, T, e_1, e_2$, and $e_3$ such that $\Gamma \vdash e_1 : \mathit{bool}$, $\Gamma \vdash e_2 : T$, $\Gamma \vdash e_3 : T$, and $\Gamma \vdash e : \mathit{unit}$
- Contextual equivalence
  - In general undecidable
  - For L3 two programs are contextually equivalent if for every context $C$ such that $\{\} \vdash C[e_1] : \mathit{unit}$ and $\{\} \vdash C[e_2] : \mathit{unit}$, we have either
    1. $\langle C[e_1], \{\}\rangle \to^\omega$ and $\langle C[e_2], \{\}\rangle \to^\omega$, or
    2. $\langle C[e_1], \{\}\rangle \to^* \langle \mathit{skip}, s_1\rangle$ and $\langle C[e_2], \{\}\rangle \to^* \langle \mathit{skip}, s_2\rangle$