Operational Semantics	$\dfrac{\langle e_1,s\rangle \to \langle e_1',s'\rangle}{\langle e_1;e_2,s\rangle \to \langle e_1';e_2,s'\rangle}$ (e.g.) $\dfrac{\langle e_1,s\rangle \Downarrow \langle n_1,s'\rangle \quad \langle e_2,s'\rangle \Downarrow \langle n_2,s''\rangle}{\langle e_1+e_2,s\rangle \Downarrow \langle n,s''\rangle}$ (where n = n_1 + n_2) (e.g.)
Run Time Errors	Trapped: cause execution to halt immediately (e.g. raising a top-level exception) Untrapped: may go unnoticed for a while and cause problems later (e.g. array out of bounds errors)
Safety	Language is safe if no untrapped errors can occur
Typing	$\Gamma \succ e:T$, assumptions Γ $\dfrac{\Gamma(l) = \text{intref} \quad \Gamma \succ \text{e} : \text{int}}{\Gamma \succ l := e : \text{unit}}$ (e.g.)
Determinancy	If $\langle e,s\rangle \to \langle e_1,s_1\rangle$, $\langle e,s\rangle \to \langle e_2,s_2\rangle$ then $\langle e_1,s_1\rangle = \langle e_2,s_2\rangle$
Progress	If $\Gamma \succ e:T$, $dom(\Gamma) \subseteq dom(s)$ then e is a value or exists some $\langle e,s\rangle \to \langle e',s'\rangle$
Type Preservation	If $\Gamma \succ e:T$, $dom(\Gamma) \subseteq dom(s)$, $\langle e,s\rangle \to \langle e',s'\rangle$ then $\Gamma \succ e':T$ and $dom(\Gamma) \subseteq dom(s')$
Safety	If $\Gamma \succ e:T$, $dom(\Gamma) \subseteq dom(s)$, $\langle e,s\rangle \to^* \langle e',s'\rangle$ then either e′ is a value or $\langle e',s'\rangle \to \langle e'',s''\rangle$
Typeability	Given Γ and e, find T such that $\Gamma \succ e:T$ is derivable or show that there is no such T
Type Checking	Given Γ, e and T, decide whether $\Gamma \succ e:T$ is right
Type Uniqueness	If $\Gamma \succ e:T$ and $\Gamma \succ e:T'$ then T = T′
Alpha Conversion	Maps symbols to variables in memory (applies scoping) A variable is free in an expression if it is not inside any (fn x:T => …) Convention: we can replace the symbol for a variable at any time in its binding location as long as we change the symbol at the binding sites at the same time (a-equivalence) Implement this with pointers / De Bruijn indices (the number of fn nodes you must traverse to reach the binder)
Substitution	{e/x}e′ is the result of substituting e for all free occurrences of x in e′
Call By Value	Evaluate left to right and parameter before application
Call By Name	Reduce left hand side until it is a function, then immediately substitute the parameter
Call By Need	As call by name, but the result of evaluating the parameter is cached for future usages
Full Beta	Allow both sides of an application to reduce, immediately apply a function to its parameter if possible (like call by name), allow reduction INSIDE functions
Recursion	Implement this by "let val rec" Operational semantics unroll the function one step and "let" the recursive function into the body of the function again
Products	T ::= …| T_1 * T_2 e ::= …| (e_1, e_2) | #1 e | #2 e
Sums	T ::= …| T_1 + T_2 e ::= …| inl e:T | inr e:T | (case e of inl (x_1:T_1) => e_1 | inr (x_2:T_2) => e_2) Need annotation because inl 3:(int + int and int + bool)
Records	Generalise products w/ labels T ::= …| {lab_1:T_1,…,lab_k:T_k} e ::= …| {lab_1 = e_1,…,lab_k=e_k} | #lab e Note that labels are unique within our expressions, types
References	T ::= …| T ref T_{loc} ::= …| T ref e ::= …| e_1 := e_2 | !e | ref e | l Must now type check the store in a more elaborate way than just $dom(\Gamma) \subseteq dom(s')$: $\Gamma \succ s$ if $\forall l \in dom(s)$: $.\exists T.\Gamma(l) = T \text{ ref} \wedge \Gamma \succ \text{s(l)} : \text{T}$ Type preservation now must ensure the store is typeable and that we extend the type assumptions after reduction by

some Γ′ with disjoint domain

Subtyping Can add a subtype relation using record types from L3

$\frac{}{T <: T}$, $\frac{T <: T' \quad T' <: T''}{T <: T''}$

Allow subtyping within record fields, forget fields on the right and reorder fields in records
Functions are contravariant on the left of □ and covariant on the right of □:

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

Can't allow type relationships or references because we could assign an inappropriate supertype / access an inappropriate subtype
Could add a downcast operator, but would require a dynamic type check for safety

Objects Can now do classes and objects using records!
Classes are a set of methods take a Representation record and access fields from it to manipulate its data (stored as refs if it is mutable)
Build constructors that return records of the right type
Can now reuse method code with subtyped records ☺

Semantic Equivalence Equivalent either if the two expressions reduce forever or if they reduce to the same value and store: this has the congruence property

Congruence Congruent if whenever $e_1 > e_2$ you have that for all contexts C and T′, if $\Gamma \succ C[e_1] : T'$ and $\Gamma \succ C[e_2] : T'$ then $C[e_1] \approx C[e_2]$

Threading T ::= …| proc
e ::= …| $e_1|e_2$
Now in the type rules let anything that returns a unit become a proc (process) and let two proc be | together
In the operational semantics allow reductions to happen on each side of |: non-determinism
Note that things like assignment and dereferencing are atomic (unlike real hardware?)

Mutexes Add M (a partial function from mutex names to Booleans): mutex state to configurations
e ::= …| lock m | unlock m
Define atomic transitions for these operations that change the mutex state and block if a mutex is currently held