Queens' College Cambridge

Semantics of Programming Languages

Alistair O'Brien

Department of Computer Science

April 17, 2021

Contents

1 Languages

1.1 Simply Typed Lambda Calculus $\lambda_{\mathbf{rec}}^{\to}$

Definition 1.1.1. (**Simply Typed Lambda Calculus $\lambda_{\mathbf{rec}}^{\to}$ Syntax**) Let Σ_{var} be a countably infinite set of variables.

Let Σ_δ be the set of δ-functions (operators / base functions) defined by

$$\Sigma_\delta = \{\cdot +^2 \cdot, \cdot \geq^2 \cdot\} \cup \left\{\mathrm{fix}^2 \ \cdot \cdot\right\}$$

and $C^n \in \Sigma_{\mathrm{constructor}}$ is the set of constructors,

$$\Sigma_{\mathrm{constructor}} = \{n^0 : n \in \mathbb{Z}\} \cup \left\{\mathrm{true}^0, \mathrm{false}^0\right\} \cup \left\{()^0\right\}$$

We define the set of *primitives* as $\Sigma_{\mathrm{primitive}} = \Sigma_{\mathrm{constructor}} \cup \Sigma_\delta$.

The simply typed lambda calculus (STLC) $\lambda_{\mathrm{rec}}^{\to}$ with *recursion* has the syntax:

$$
\begin{aligned}
v \ \texttt{::=}\ & \lambda x : \tau . e \\
\texttt{|}\ & \underbrace{P^n\, v_1\ \ldots\ v_n}_{\texttt{constructed values w/ arity } n} \\
\texttt{|}\ & \underbrace{p^n\, v_1\ \ldots\ v_k}_{\texttt{partailly constructed values w/ arity } n} \qquad k < n
\end{aligned}
$$

$$
\begin{aligned}
e \ \texttt{::=}\ & x \in \Sigma_{\texttt{var}} \\
\texttt{|}\ & e_1\ e_2 \\
\texttt{|}\ & \lambda x : \tau . e \\
\texttt{|}\ & v \\
\texttt{|}\ & \texttt{let}\ x : \tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \\
\texttt{|}\ & \texttt{case}\ e\ \texttt{of}\ (C_1^{m_1} x_1 : \tau_1^{C_1} \ldots x_{m_1} : \tau_n^{C_1} \to e_1 \mid \ldots \mid C_n^{m_n} x_1 : \tau_1^{C_n} \ldots x_{m_n} : \tau_m^{C_n} \to e_n)
\end{aligned}
$$

where $\tau \in \Sigma_\tau$ is the set of types for $\lambda_{\mathrm{rec}}^{\to}$ (See definition ??), $P^n \in \Sigma_{\mathrm{primitive}}$ and $p^n\ v_1 \ldots v_k \triangleq \lambda x_{k+1} : \tau_{k+1}^P, \ldots, x_n : \tau_n^P . P^n\ v_1 \ldots v_k\ x_{k+1} \ldots x_n$.

- Syntactic equivalence is denoted = (Defined by equivalent *abstract syntax trees*)
- Precedence: (in order) $\geq, +$, application.

Definition 1.1.2. ($\lambda^{\rightarrow}_{\mathbf{rec}}$ **Types**) The set of types $\tau \in \Sigma_\tau$ for $\lambda^{\rightarrow}_{\text{rec}}$ is defined by

$$
\begin{aligned}
\tau \ \texttt{::=}\ & \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \\
\mid\ & \tau_1 \rightarrow \tau_2
\end{aligned}
$$

- $\rightarrow$ is right associative.
- **Syntactic Sugar** (*Derived operators*):

$$\lambda x_1 : \tau_1, \ldots, x_n : \tau_n.e \triangleq \lambda x_1 : \tau_1.\lambda x_2 : \tau_2.\ldots\lambda x_n : \tau_n.e$$

$$
\begin{aligned}
&\texttt{let rec}\ x : \tau_1 \rightarrow \tau_2\ \texttt{=}\ \lambda y : \tau_1.e_2\ \texttt{in}\ e_2 \\
&\quad \triangleq \texttt{let}\ x : \tau_1 \rightarrow \tau_2\ \texttt{=}\ \texttt{fix}\ (\lambda x : \tau_1 \rightarrow \tau_2.\lambda y : \tau_1.e_2)\ \texttt{in}\ e_2
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{let rec}\ x_1 : \tau_{11} \rightarrow \tau_{12}\ \texttt{=}\ \lambda y_1 : \tau_{11}.e_1 \\
&\quad \texttt{and}\ \texttt{...} \\
&\quad \texttt{and}\ x_n : \tau_{n1} \rightarrow \tau_{n2}\ \texttt{=}\ \lambda y_n : \tau_{n1}.e_n\ \texttt{in}\ e \\
&\triangleq \\
&\texttt{let rec}\ x_1' : (\tau_{21} \rightarrow \tau_{22}) \rightarrow \cdots \rightarrow (\tau_{n1} \rightarrow \tau_{n2}) \rightarrow (\tau_{11} \rightarrow \tau_{12}) \\
&\quad \texttt{=}\ \lambda x_2 : \tau_{21} \rightarrow \tau_{22} \ldots x_n : \tau_{n1} \rightarrow \tau_{n2}.\lambda y_1 : \tau_{11}. \\
&\qquad \texttt{let}\ x_1 = x_1'\ x_2\ \ldots\ x_n\ \texttt{in}\ e_1 \\
&\texttt{in let rec}\ x_2' : (\tau_{31} \rightarrow \tau_{32}) \rightarrow \cdots \rightarrow (\tau_{n1} \rightarrow \tau_{n2}) \rightarrow (\tau_{21} \rightarrow \tau_{22}) \\
&\quad \texttt{=}\ \lambda x_3 : \tau_{31} \rightarrow \tau_{32} \ldots x_n : \tau_{n1} \rightarrow \tau_{n2}.\lambda y_2 : \tau_{21}. \\
&\qquad \texttt{let}\ x_2 = x_2'\ x_3\ \ldots\ x_n\ \texttt{in} \\
&\qquad \texttt{let}\ x_1 = x_1'\ x_2\ \ldots\ x_n\ \texttt{in}\ e_2 \\
&\texttt{...} \\
&\texttt{in let rec}\ x_n : \tau_{n1} \rightarrow \tau_{n2} \\
&\quad \texttt{=}\ \lambda y_n : \tau_{n1}. \\
&\qquad \texttt{let}\ x_{n-1}\ \texttt{=}\ x_{n-1}'\ x_n\ \texttt{in} \\
&\qquad \texttt{...} \\
&\qquad \texttt{let}\ x_1\ \texttt{=}\ x_1'\ x_2\ \ldots\ x_n\ \texttt{in}\ e_n \\
&\texttt{in} \\
&\quad \texttt{let}\ x_{n-1} : \tau_{(n-1)1} \rightarrow \tau_{(n-1)2}\ \texttt{=}\ x_{n-1}'\ x_n\ \texttt{in}
\end{aligned}
$$

```
...
let x_2 : τ_21 → τ_22 = x'_2 x_3 ... x_n in
let x_1 : τ_11 → τ_12 = x'_1 x_2 ... x_n in
    e
```

Definition 1.1.3. (**Free and bound variables**) The sets of *free* variables and *variables* in e, are inductively defined by

$$\begin{aligned}
fv(x) &= \{x\} \\
fv(e_1\ e_2) &= fv(e_1) \cup fv(e_2) \\
fv(\lambda x.e) &= fv(e) \setminus \{x\} \\
fv(v) &= \emptyset \\
fv(\text{let } x : \tau = e_1 \text{ in } e_2) &= fv(e_1) \cup fv(e_2) \setminus \{x\} \\
fv(\text{case } \ldots) &= fv(e) \cup \bigcup_{1 \leq i \leq n} fv(e_i) \setminus \{x_1, \ldots, x_{m_i}\}
\end{aligned}
\qquad
\begin{aligned}
var(x) &= \{x\} \\
var(e_1\ e_2) &= var(e_1) \cup var(e_2) \\
var(\lambda x.e) &= var(e) \cup \{x\} \\
var(v) &= \emptyset \\
var(\text{let } x : \tau = e_1 \text{ in } e_2) &= var(e_1) \cup var(e_2) \\
var(\text{case } \ldots) &= var(e) \cup \bigcup_{1 \leq i \leq n} var(e_i)
\end{aligned}$$

- The set of bound variables of e is $bv(e) = var(e) \setminus fv(e)$.

Definition 1.1.4. (**Substitution**) A **substitution** θ is a finite partial function $\theta : \Sigma_{\text{var}} \rightharpoonup \Sigma_e$.

- **Notation**: $\{t_1/x_1, \ldots, t_n/x_n\}$ denotes a substitution θ, where $\theta(x_i) = t_i$ and $t/x \in \theta \iff \theta(x) = t$.

Definition 1.1.5. (α**-equivalence**) The $=_\alpha : \Sigma_e \longleftrightarrow \Sigma_e$ is inductively defined by

$$\frac{}{x =_\alpha x} \qquad \frac{z \notin var(e_1) \cup var(e_2) \qquad \{z/x\}\, e_1 =_\alpha \{z/y\}\, e_2}{\lambda x.e_1 =_\alpha \lambda y.e_2} \qquad \cdots .$$

- $=_\alpha$ introduces a *unique* (canonical) form of the term.
- de Brunjin indexes: IMAGE

Definition 1.1.6. (**Application**) The application of a substitution θ to $M \in \Lambda$, denoted $\theta\, M$, is inductively defined by

$$\begin{aligned}
\theta x &= \begin{cases} \theta(x) & \text{if } x \in \operatorname{dom}\theta \\ x & \text{otherwise} \end{cases} \\
\theta\, \lambda x.e &= \begin{cases} \lambda x.\, [(\theta \setminus \{e'/x\})e] & e'/x \in \theta \\ \lambda x.\theta e & x \notin \operatorname{dom}\theta \wedge x \notin fv(\operatorname{rng}\theta) \end{cases} \\
\theta\, e_1\ e_2 &= (\theta\, e_1)\ (\theta\, e_2) \\
&\vdots
\end{aligned}$$

- The condition $x \notin \operatorname{dom}\theta \wedge x \notin fv(\operatorname{rng}\theta)$ avoids *name capture.* This definition of application is said to be *capture avoiding.*
- $=_\alpha$ is used to "rename" variables e.g. $\{y/x\}\,(\lambda y.x) =_\alpha \{y/x\}\,(\lambda z.x) = \lambda z.y$.

1.1.1 Small-Step Semantics

- Operational semantics define the evaluation behavior using a transition relation $\longrightarrow$.
- Evaluation Strategies:
  - **Call-by-value**: Reduce e_1 to $\lambda x : \tau.e$. Reduce e_2 to v. Evaluate $\{v/x\}\, e$.
  - **Call-by-name**: Reduce e_1 to $\lambda x : \tau.e$. Evaluate $\{e_2/x\}\, e$.
  - **Call-by-need**: Reduce e_2 to $\lambda x : \tau.e$. Substitute x w/ *lazy pointers* to e_2. Evaluate. Semantically, equivalent to call-by-name, but more efficient. Since each argument is evaluated *at most* once.

  Strategies are implemented via *evaluation contexts*.
- $\lambda^{\rightarrow}_{\text{rec}}$ implements call-by-value w/ left-to-right evaluation.

Definition 1.1.7. (**Small-Step Semantics of $\lambda^{\rightarrow}_{\text{rec}}$**) Let us define the evaluation contexts $E \in \Sigma_E$ for $\lambda^{\rightarrow}_{\text{rec}}$:

$$
\begin{aligned}
E \; \texttt{::=} \; & [\cdot] \\
\texttt{|} \; & E \; e \\
\texttt{|} \; & v \; E \\
\texttt{|} \; & \texttt{let} \; x : \tau = E \; \texttt{in} \; e_2 \\
\texttt{|} \; & \texttt{case} \; E \; \texttt{of} \; (\ldots)
\end{aligned}
$$

The small-step semantics of $\lambda^{\rightarrow}_{\text{rec}}$ is defined by the transition relation $\longrightarrow : \Sigma_e \to \Sigma_e$, inductively defined by

$$\text{(Eval)}\ \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

$$\text{(Op }+)\ \frac{n = n_1 + n_2}{n_1 + n_2 \longrightarrow n}$$

$$\text{(Op }\geq)\ \frac{b = n_1 \geq n_2}{n_1 \geq n_2 \longrightarrow b}$$

$$\text{(Fix)}\ \frac{}{\text{fix } v_1 \; v_2 \longrightarrow v_1 \; (\text{fix } v_1) \; v_2}$$

$$(\lambda)\ \frac{}{(\lambda x : \tau . e) \; v \longrightarrow \{v/x\}\, e}$$

$$\text{(Let)}\ \frac{}{\text{let } x : \tau = v \text{ in } e_2 \longrightarrow \{v/x\}\, e_2}$$

$$\text{(Case)}\ \frac{}{\text{case } C_i^{m_i} v_1 \ldots v_{m_i} \text{ of } (\ldots \mid C_i^{m_i} x_1 : \tau_1^{C_i} \ldots x_{m_i} : \tau_{m_i}^{C_i} \to e_i \mid \ldots) \longrightarrow \{v_1/x_1, \ldots, v_{m_i}/x_{m_i}\}\, e_i}$$

- **Notation**:
  - The many-step transition relation $\longrightarrow^*$ is the reflexive transitive closure of $\longrightarrow$.
  - $e \not\longrightarrow$ iff $\neg \exists e' \in \Sigma_e . e \longrightarrow e'$.
- e is *stuck* iff $e \notin \Sigma_v$ and $e \not\longrightarrow$.
- $\longrightarrow^\omega$ denotes a *diverging* (infinite) sequence of $\longrightarrow$ transitions.

Theorem 1.1.1. (**Determininacy for $\lambda^{\rightarrow}_{\text{rec}}$**)

$$
\begin{aligned}
&\forall e_0, e_1, e_2 \in \Sigma_e . \\
&\qquad e_0 \longrightarrow e_1 \wedge e_0 \longrightarrow e_2 \implies e_1 = e_2
\end{aligned}
$$

1.1.2 Big-Step Semantics

- **Problem**: $\longrightarrow$ cannot distinguish between expressions and values
- **Solution**: Big-step semantics

Definition 1.1.8. (**Big-Step Semantics for $\lambda^{\rightarrow}_{\textbf{rec}}$**) The big-step semantics $\Downarrow$ for $\lambda^{\rightarrow}_{\text{rec}}$ is the relation $\Downarrow: \Sigma_e \longleftrightarrow \Sigma_v$, inductively defined by:

$$(\text{Id})\,\frac{}{v \Downarrow v}$$

$$(\text{Op }+)\,\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 + e_2 \Downarrow n}\,[n = n_1 + n_2]$$

$$(\text{Op }\geq)\,\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 \geq e_2 \Downarrow b}\,[b = n_1 \geq n_2]$$

$$(\text{Fix})\,\frac{e_1\ (\text{fix } e_1)\ e_2 \Downarrow v}{\text{fix } e_1\ e_2 \Downarrow v}$$

$$(\text{App})\,\frac{e_1 \Downarrow \lambda x : \tau . e \qquad e_2 \Downarrow v \qquad \{v/x\}\, e \Downarrow v'}{e_1\ e_2 \Downarrow v'}$$

$$(\text{Let})\,\frac{e_1 \Downarrow v_1 \qquad \{v_1/x\}\, e_2 \Downarrow v_2}{\text{let } x : \tau = e_1 \text{ in } e_2 \Downarrow v_2}$$

$$(\text{Case})\,\frac{e \Downarrow C_i^m v_1 \dots v_{m_i} \qquad \{v_1/x_1, \dots, v_{m_i}/x_{m_i}\}\, e_i \Downarrow v}{\text{case } C_i^{m_i} v_1 \dots v_{m_i} \text{ of } (\dots \mid C_i^{m_i} x_1 : \tau_1^{C_i} \dots x_{m_i} : \tau_{m_i}^{C_i} \rightarrow e_i \mid \dots) \Downarrow v}$$

- **Advantages**:
  - Fewer inductive rules. Easier to inspect and define: "*natural semantics*"
  - Useful for *definitional interpreters* (see Compilers).
- **Disadvantages**:
  - Not suitable for concurrency extensions.
  - Doesn't distinguished between $\longrightarrow^{\omega}$ and $\not\longrightarrow$ transitions $\implies$ Not suitable for proving properties e.g. progress

1.1.3 Types

- Types ensure expressions e are "*correct*". Syntax directed, each typing rule corresponds to a abstract syntax rule.

Definition 1.1.9. (**Typing Context**) The typing context Γ in $\lambda^{\rightarrow}_{\text{rec}}$ is a finite partial function $\Gamma : \Sigma_{\text{var}} \rightharpoonup \Sigma_\tau$.

- Γ is the set of assumptions about each type of variables in an expression.
- **Notation**:
  - Context extension: $\Gamma, x : \tau$ denotes the *extension* of Γ. Equivalent to $\Gamma \setminus \{(x, \tau') : (x, \tau') \in \Gamma\} \cup \{(x, \tau)\}$
  - Context membership: $x : \tau \in \Gamma$. Equivalent to $\Gamma(x) \downarrow \wedge \Gamma(x) = \tau$.
  - Empty Context: $\cdot$.

Definition 1.1.10. (**Typing Relation** $\vdash$) The *typing relation* $\vdash \subseteq \Sigma_\Gamma \times \Sigma_e \times \Sigma_\tau$, with infix notation $\Gamma \vdash e : \tau$, defined inductively by:

For constructors $C^n \in \Sigma_{\text{constructor}}$:

$$(\text{Int})\ \frac{}{\Gamma \vdash n : \text{int}} \quad (\text{Bool})\ \frac{}{\Gamma \vdash b : \text{bool}} \quad (\text{Unit})\ \frac{}{\Gamma \vdash () : \text{unit}}$$

For δ-functions $\delta^n \in \Sigma_\delta$:

$$(\text{Op } +)\ \frac{}{\Gamma \vdash + : \text{int} \to (\text{int} \to \text{int})}$$

$$(\text{Op } \geq)\ \frac{}{\Gamma \vdash \geq : \text{int} \to (\text{int} \to \text{bool})}$$

$$(\text{Fix})\ \frac{}{\Gamma \vdash \text{fix} : [(\tau_1 \to \tau_2) \to \tau_1 \to \tau_2] \to \tau_1 \to \tau_2}$$

For expressions $e \in \Sigma_e$, we have:

$$\text{(Var)}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\text{(App)}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$(\lambda)\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2}$$

$$\text{(Let)}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x : \tau_1 = e_1 \text{ in } e_2 : \tau_2}$$

$$\text{(Case)}\ \frac{\Gamma \vdash e : \tau \qquad \forall 1 \leq i \leq n. \Gamma \vdash C_i^{m_i} : \tau_1^{C_i} \to \big(\cdots (\tau_{m_i}^{C_i} \to \tau) \cdots \big) \qquad \forall 1 \leq i \leq n. \Gamma, x_1 : \tau_1^{C_i}, \ldots, x_m : \tau_{m_i}^{C_i} \vdash e_i : \tau'}{\Gamma \vdash \text{case } e \text{ of } (C_1^{m_1} x_1 : \tau_1^{C_1} \ldots x_{m_1} : \tau_{m_1}^{C_1} \to e_1 \mid \ldots \mid C_n^{m_n} x_1 : \tau_1^{C_n} \ldots x_{m_n} : \tau_{m_n}^{C_n} \to e_n) : \tau'}$$

where

$$n = \Big| \big\{ C_i^{m_i} \in \Sigma_{\text{constructors}} : \Gamma \vdash C_i^{m_i} : \tau_1^{C_i} \to (\cdots (\tau_{m_i}^{C_i} \to \tau) \cdots) \big\} \Big| .$$

- Constructors and δ-functions have derived typing rules (denoted w/ $'$) e.g. $\text{(Op +')}\ \frac{e_1 : \text{int} \qquad e_2 : \text{int}}{e_1 + e_2 : \text{int}}$

Theorem 1.1.2. (**Decidability of Typing**) The typing relation $\vdash$ is decidable, that is to say:

$$\begin{aligned} &\forall \Gamma \in \Sigma_\Gamma, e \in \Sigma_e, \tau \in \Sigma_\tau . \\ &\qquad \exists \text{ algorithm } \mathcal{A}. \mathcal{A}(\Gamma, e, \tau) = \text{ true } \iff \Gamma \vdash e : \tau \end{aligned}$$

- Typing Algorithms:
  - *Type checking*: Given Γ, e, τ, determine whether $\Gamma \vdash e : \tau$ is true.
  - *Type inference*: Given Γ, e, determine existence of $\tau \in \Sigma_\tau$ s.t $\Gamma \vdash e : \tau$.

Theorem 1.1.3. (**Uniquness of Typing**)

$$\begin{aligned} &\forall \Gamma \in \Sigma_\Gamma, e \in \Sigma_e, \tau, \tau' \in \Sigma_\tau . \\ &\qquad \Gamma \vdash e : \tau \land \Gamma \vdash e : \tau' \implies \tau = \tau' \end{aligned}$$

Theorem 1.1.4. (**Progress for $\lambda^{\rightarrow}_{\mathbf{rec}}$**)

$$\begin{aligned}&\forall e \in \Sigma_e, \tau \in \Sigma_\tau. \\ &\qquad \cdot \vdash e : \tau \implies e \in \Sigma_v \vee (\exists e' \in \Sigma_e . e \longrightarrow e')\end{aligned}$$

Lemma 1.1.1. (**Weakening**)

$$\begin{gathered}\forall \Gamma \vdash e : \tau . \forall x \notin \operatorname{dom} \Gamma, \tau' \in \Sigma_\tau \\ \Gamma, x : \tau' \vdash e : \tau\end{gathered}$$

Lemma 1.1.2. (**Value Inversion Lemma of Typing**) For all $\Gamma \vdash v : \tau$,

- If $\tau = \text{unit}$, then $v = ()$.
- If $\tau = \text{int}$, then $\exists n \in \mathbb{Z}.v = n$.
- If $\tau = \text{bool}$, then $\exists b \in \{\text{true}, \text{false}\}.v = b$.

Lemma 1.1.3. (**Substitution Lemma**)

$$\begin{aligned}&\forall \Gamma \in \Sigma_\Gamma, x \in \Sigma_{\text{var}}, e, e' \in \Sigma_e, \tau, \tau' \in \Sigma_\tau. \\ &\qquad \Gamma \vdash e : \tau \wedge \Gamma, x : \tau \vdash e' : \tau' \implies \Gamma \vdash \{e/x\}\, e' : \tau'\end{aligned}$$

Theorem 1.1.5. (**Preservation for $\lambda^{\rightarrow}_{\mathbf{rec}}$**)

$$\begin{aligned}&\forall e, e' \in \Sigma_e, \tau \in \Sigma_\tau. \\ &\qquad \cdot \vdash e : \tau \wedge e \longrightarrow e' \implies \cdot \vdash e' : \tau\end{aligned}$$

- Progress and type preservation $\implies$ Type safety

Theorem 1.1.6. (**Type Safety for $\lambda^{\rightarrow}_{\mathbf{rec}}$**)

$$\begin{aligned}&\forall e, e' \in \Sigma_e, \tau \in \Sigma_\tau. \\ &\qquad \cdot \vdash e : \tau \wedge e \longrightarrow^* e' \implies e' \in \Sigma_v \vee (\exists e'' \in \Sigma_e . e' \longrightarrow e'')\end{aligned}$$

1.2 Mutability

- **Problem**: $\lambda^{\to}_{\text{rec}}$ is *purely function*. Add side effects via *mutable store*

1.2.1 Store $\lambda^{\to}_{\textbf{rec + ref}}$

Definition 1.2.1. (**Store**) Let Σ_{loc} be a countably infinite set of *locations*. A *store* s is a finite partial function $s : \Sigma_{\text{loc}} \rightharpoonup \Sigma_v$. The set of stores is denoted Σ_s.

- **Notation**: $s, \ell \to v$ denotes the *extension* of s. Equivalent to $s \setminus \{(\ell, v') : (\ell, v') \in \Gamma\} \cup \{(x, v)\}$

Definition 1.2.2. ($\lambda^{\to}_{\textbf{rec + ref}}$ **Syntax**) Let Σ_δ be the extended set of δ-functions:

$$\begin{aligned}\Sigma_\delta = &\cdots \\ &\cup \left\{\cdot;^2 \cdot\right\} \cup \left\{\text{ref}^2 \cdot, !^1 \cdot, \cdot :=^2 \cdot\right\}\end{aligned}$$

and $\Sigma_{\text{constructor}}$ be the extended set of constructors:

$$\begin{aligned}\Sigma_{\text{constructor}} = &\cdots \\ &\cup \left\{\ell^0 : \ell \in \Sigma_{\text{loc}}\right\}\end{aligned}$$

The simply typed lambda calculus with recursion and *references*, denoted $\lambda^{\to}_{\text{rec + ref}}$, is defined as:

$$\begin{aligned}
v \ \texttt{::=}\ & \lambda x : \tau . e \\
\texttt{|}\ & \underbrace{P^n\, v_1 \ \ldots\ v_n}_{\texttt{constructed values w/ arity } n} \\
\texttt{|}\ & \underbrace{p^n\, v_1 \ \ldots\ v_k}_{\texttt{partailly constructed values w/ arity } n} \qquad k < n
\end{aligned}$$

$$\begin{aligned}
e \ \texttt{::=}\ & x \in \Sigma_{\texttt{var}} \\
\texttt{|}\ & e_1 \ e_2 \\
\texttt{|}\ & \lambda x : \tau . e \\
\texttt{|}\ & v \\
\texttt{|}\ & \texttt{let } x : \tau \texttt{ = } e_1 \texttt{ in } e_2 \\
\texttt{|}\ & \texttt{case } e \texttt{ of } (C_1^{m_1} x_1 : \tau_1^{C_1} \ldots x_{m_1} : \tau_n^{C_1} \to e_1 \mid \ldots \mid C_n^{m_n} x_1 : \tau_1^{C_n} \ldots x_{m_n} : \tau_m^{C_n} \to e_n)
\end{aligned}$$

- **Design Choices of References**:
  - Explicit dereferencing and assignment, initialization
  - Garbage collection required (since store grows)
  - No reference arithmetic (unlike C) or reference equality.

Definition 1.2.3. ($\lambda^{\rightarrow}_{\textbf{rec + ref}}$ **Types**) The set of types $\tau \in \Sigma_\tau$ for $\lambda^{\rightarrow}_{\text{rec + ref}}$ is defined by

$$
\begin{aligned}
\tau \; ::= & \; \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \\
\mid & \; \tau_1 \to \tau_2 \\
\mid & \; \tau \; \texttt{ref}
\end{aligned}
$$

Definition 1.2.4. (**Small-Step Semantics of** $\lambda^{\rightarrow}_{\textbf{rec + ref}}$) A *config* is defined as the pair $\langle e, s\rangle$, where $e \in \Sigma_e, s \in \Sigma_s$. The set of configs is defined $\Sigma_{\text{config}} = \Sigma_e \times \Sigma_s$.

The small-step semantics of $\lambda^{\rightarrow}_{\text{rec + ref}}$, $\longrightarrow$: $\Sigma_{\text{config}} \longleftrightarrow \Sigma_{\text{config}}$, is defined by

$$\vdots$$

$$(\text{Seq}) \; \frac{}{\langle (); v, s\rangle \longrightarrow \langle v, s\rangle}$$

$$(\text{Ref}) \; \frac{\ell \notin \operatorname{dom} s}{\langle \text{ref } v, s\rangle \longrightarrow \langle \ell, (s, \ell \to v)\rangle}$$

$$(\text{Deref}) \; \frac{(\ell, v) \in s}{\langle !\ell, s\rangle \longrightarrow \langle v, s\rangle}$$

$$(\text{Assign}) \; \frac{\ell \in \operatorname{dom} s}{\langle \ell := v, s\rangle \longrightarrow \langle (), (s, \ell \to v)\rangle}$$

$$\vdots$$

- **Problem**: (Ref) rule $\implies$ non-determinism. Since new locations are arbitrary.
- **Solution**: α-equivalence on locations, denoted ℓ-equivalence.

Definition 1.2.5. (**Reference Substitution**) A reference substitution σ is a finite partial function $\sigma : \Sigma_{\text{loc}} \rightharpoonup \Sigma_{\text{loc}}$.

- Application on expression, denoted $\sigma(e)$, is defined inductively, w/ base case on location values $\ell \in \Sigma$.
- **Notation**: $\sigma(s) = \{(\sigma(\ell), v) : (\ell, v) \in s\}$

Definition 1.2.6. (ℓ **Equivalence**) The ℓ-equivalence on expressions $=_\ell$: $\Sigma_e \longleftrightarrow \Sigma_e$ is defined by

$$e_1 =_\ell e_2 \iff \exists \sigma \in \Sigma_\sigma . e_1 = \sigma(e_2).$$

Similarly, for $=_\ell$: $\Sigma_s \longleftrightarrow \Sigma_s$ on stores,

$$s_1 =_\ell s_2 \iff \exists \sigma \in \Sigma_\sigma . s_1 = \sigma(s_2).$$

Theorem 1.2.1. (**Determinancy for $\lambda^{\rightarrow}_{\textbf{rec + ref}}$**)

$$\begin{aligned}
&\forall e_0, e_1, e_2 \in \Sigma_e, s_0, s_1, s_2 \in \Sigma_s . \\
&\qquad \langle e_0, s_0 \rangle \longrightarrow \langle e_1, s_1 \rangle \wedge \langle e_0, s_0 \rangle \longrightarrow \langle e_2, s_2 \rangle \\
&\qquad \implies \langle e_1, s_1 \rangle =_\ell \langle e_2, s_2 \rangle
\end{aligned}$$

Definition 1.2.7. (**Store Typing Context**) The store typing context Σ in $\lambda^{\rightarrow}_{\text{rec + ref}}$ is a finite partial function $\Sigma : \Sigma_{\text{loc}} \rightharpoonup \Sigma_\tau$. The set of store typing contexts is denoted Σ_Σ.

- The typing context in $\lambda^{\rightarrow}_{\text{rec + ref}}$ is denoted $\Sigma; \Gamma$.

Definition 1.2.8. (**Typing Relation**) The *typing relation* $\vdash \subseteq \Sigma_\Sigma \times \Sigma_\Gamma \times \Sigma_e \times \Sigma_\tau$, with infix notation $\Sigma; \Gamma \vdash e : \tau$, defined inductively by:

For constructors $C^n \in \Sigma_{\text{constructor}}$:

$$\cdots \quad (\text{Loc}) \, \frac{\ell : \tau \in \Sigma}{\Sigma; \Gamma \vdash \ell : \tau \text{ref}}$$

For δ-functions $\delta^n \in \Sigma_\delta$:

$$\vdots$$

$$\text{(Seq)}\ \frac{}{\Sigma; \Gamma \vdash \cdot ; \cdot : \text{unit} \to (\tau \to \tau)}$$

$$\text{(Ref)}\ \frac{}{\Sigma; \Gamma \vdash \text{ref}\cdot : \tau \to \tau\ \text{ref}}$$

$$\text{(Deref)}\ \frac{}{\Sigma; \Gamma \vdash !\cdot : \tau\ \text{ref} \to \tau}$$

$$\text{(Assign)}\ \frac{}{\Sigma; \Gamma \vdash \cdot := \cdot : \tau\ \text{ref} \to (\tau \to \text{unit})}$$

For expressions $e \in \Sigma_e$, we have:

$$\vdots$$

Definition 1.2.9. (**Well-typed store**) A store s is well typed in the context Σ, denoted $\Sigma \vdash s$, iff

$$\operatorname{dom} s = \operatorname{dom} \Sigma \wedge \forall (\ell, v) \in s. \ell : \tau \in \Sigma \implies \Sigma; \cdot \vdash v : \tau.$$

Theorem 1.2.2. (**Progress for $\lambda^{\to}_{\mathbf{rec\ +\ ref}}$**)

$$\begin{aligned}&\forall \Sigma \in \Sigma_\Sigma, e \in \Sigma_e, \tau \in \Sigma_\tau, s \in \Sigma_s.\\ &\qquad \Sigma \vdash s \wedge \Sigma; \cdot \vdash e : \tau\\ &\qquad\quad \implies e \in \Sigma_v \vee (\exists e' \in \Sigma_e, s' \in \Sigma_s. \langle e, s\rangle \longrightarrow \langle e', s'\rangle)\end{aligned}$$

Theorem 1.2.3. (**Preservation for $\lambda^{\to}_{\mathbf{rec\ +\ ref}}$**)

$$\begin{aligned}&\forall \Sigma \in \Sigma_\Sigma, e, e' \in \Sigma_e, \tau \in \Sigma_\tau, s, s' \in \Sigma_s.\\ &\qquad \Sigma \vdash s \wedge \Sigma; \cdot \vdash e : \tau \wedge \langle e, s\rangle \longrightarrow \langle e', s'\rangle\\ &\qquad\quad \implies \exists \Sigma' \in \Sigma_\Sigma. \operatorname{dom} \Sigma \cap \Sigma = \emptyset \wedge \Sigma, \Sigma' \vdash s \wedge \Sigma, \Sigma'; \cdot \vdash e : \tau\end{aligned}$$

Theorem 1.2.4. (**Type Safety for $\lambda^{\to}_{\mathbf{rec\ +\ ref}}$**)

$$\begin{aligned}&\forall \Sigma \in \Sigma_\Sigma, e, e' \in \Sigma_e, s, s' \in \Sigma_s, \tau \in \Sigma_\tau.\\ &\qquad \Sigma \vdash s \wedge \Sigma; \cdot \vdash e : \tau \wedge \langle e, s\rangle \longrightarrow^* \langle e', s'\rangle\\ &\qquad\quad \implies e' \in \Sigma_v \vee (\exists e'' \in \Sigma_e, s'' \in \Sigma_s. \langle e', s'\rangle \longrightarrow \langle e'', s''\rangle)\end{aligned}$$

1.3 Structured Data

1.3.1 Product and Sum Types $\lambda^{\rightarrow}_{\textbf{rec + ref} + (\times/+)}$

- **Idea**: Add constructors for *product* $\tau_1 \times \tau_2$ and *sum* $\tau_1 + \tau_2$ types.

Definition 1.3.1. ($\lambda^{\rightarrow}_{\textbf{rec + ref} + (\times/+)}$ **Syntax**) Let $\Sigma_{\text{constructor}}$ be the extended set of constructors:

$$\begin{aligned}\Sigma_{\text{constructor}} &= \cdots \\ &\cup \left\{(\cdot,\cdot)^2, \text{inl}^1 : \tau_1 + \tau_2, \text{inr}^1 \cdot : \tau_1 + \tau_2\right\}\end{aligned}$$

The simply typed lambda calculus with recursion, references and product / sum types, denoted $\lambda^{\rightarrow}_{\text{rec + ref} + (\times/+)}$, is defined as:

$$\begin{aligned} v \ \texttt{::=}\ & \lambda x : \tau . e \\ \texttt{|}\ & \underbrace{P^n\, v_1\, \ldots\, v_n}_{\texttt{constructed values w/ arity } n} \\ \texttt{|}\ & \underbrace{p^n\, v_1\, \ldots\, v_k}_{\texttt{partailly constructed values w/ arity } n} \qquad k < n \end{aligned}$$

$$\begin{aligned} e \ \texttt{::=}\ & x \in \Sigma_{\texttt{var}} \\ \texttt{|}\ & e_1\ \ e_2 \\ \texttt{|}\ & \lambda x : \tau . e \\ \texttt{|}\ & v \\ \texttt{|}\ & \texttt{let}\ x : \tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \\ \texttt{|}\ & \texttt{case}\ e\ \texttt{of}\ (C_1^{m_1} x_1 : \tau_1^{C_1} \ldots x_{m_1} : \tau_n^{C_1} \rightarrow e_1 \mid \ldots \mid C_n^{m_n} x_1 : \tau_1^{C_n} \ldots x_{m_n} : \tau_m^{C_n} \rightarrow e_n) \end{aligned}$$

- **Syntactic Sugar**:

$$\begin{aligned} \texttt{\#1}\ e &\triangleq \texttt{case}\ e\ \texttt{of}\ \texttt{((}x_1 : \tau_1\texttt{,}\ x_2 : \tau_2\texttt{)}\ \rightarrow\ x_1\texttt{)} \\ \texttt{\#2}\ e &\triangleq \texttt{case}\ e\ \texttt{of}\ \texttt{((}x_1 : \tau_1\texttt{,}\ x_2 : \tau_2\texttt{)}\ \rightarrow\ x_2\texttt{)} \end{aligned}$$

  given $\Sigma; \Gamma \vdash e : \tau_1 \times \tau_2$.

- Tuples implemented using the syntactic sugar:

$$\texttt{(}e_1\texttt{,}\ \ldots\texttt{,}\ e_n\texttt{)} \triangleq \texttt{(}e_1\texttt{,}\ \texttt{(}e_2\texttt{,}\ \texttt{(}\ldots\ \texttt{(}e_{n-1}\texttt{,}\ e_n\texttt{)}\ \ldots\texttt{)))}$$

Definition 1.3.2. ($\lambda^{\rightarrow}_{\textbf{rec + ref}}$ **Types**) The set of types $\tau \in \Sigma_\tau$ for $\lambda^{\rightarrow}_{\text{rec + ref}}$ is defined by

$$\begin{aligned}\tau \ \texttt{::=}\ & \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \\ & \mid \tau_1 \rightarrow \tau_2 \mid \tau\ \texttt{ref} \\ & \mid \tau_1\ \texttt{+}\ \tau_2 \mid \tau_1 \times \tau_2\end{aligned}$$

- Product types are not-associative: $\tau_1 \times (\tau_2 \times \tau_3) \neq (\tau_1 \times \tau_2) \times \tau_3$.
- Sum type constructors require annotations, due to lack of *polymorphism* and type inference.

Definition 1.3.3. (**Small-Step Semantics of** $\lambda^{\rightarrow}_{\textbf{rec + ref + data}}$) See definition ??. (*No additional transition rules required*).

Definition 1.3.4. (**Typing Relation**) The *typing relation* $\vdash \subseteq \Sigma_\Sigma \times \Sigma_\Gamma \times \Sigma_e \times \Sigma_\tau$, with infix notation $\Sigma; \Gamma \vdash e : \tau$, defined inductively by:

For constructors $C^n \in \Sigma_{\text{constructor}}$:

$$\vdots$$

$$(\text{Product})\ \frac{}{\Sigma; \Gamma \vdash (\cdot, \cdot) : \tau_1 \rightarrow (\tau_2 \rightarrow \tau_1 \times \tau_2)}$$

$$(\text{Inl})\ \frac{}{\Sigma; \Gamma \vdash (\text{inl}\cdot : \tau_1 + \tau_2) : \tau_1 \rightarrow \tau_1 + \tau_2}$$

$$(\text{Inr})\ \frac{}{\Sigma; \Gamma \vdash (\text{inr}\cdot : \tau_1 + \tau_2) : \tau_2 \rightarrow \tau_1 + \tau_2}$$

For δ-functions $\delta^n \in \Sigma_\delta$:

$$\vdots$$

For expressions $e \in \Sigma_e$, we have:

$$\vdots$$

- **Theorem**: Progress, preservation, determinism are identical. See section ??.

1.3.2 Records $\lambda^{\rightarrow}_{\mathbf{rec}\,+\,\mathbf{ref}\,+\,(\times/+/\{\})}$

- **Idea**: Extend product types w/ records

Definition 1.3.5. ($\lambda^{\rightarrow}_{\mathbf{rec}\,+\,\mathbf{ref}\,+\,(\times/+/\{\})}$ **Syntax**) Let Σ_{lab} be a countably infinite set of *labels*. Let $\Sigma_{\text{constructor}}$ be the extended set of constructors:

$$\begin{aligned}\Sigma_{\text{constructor}} &= \cdots \\ &\cup \left\{ \{\text{lab}_1 : \tau_1, \ldots, \text{lab}_n : \tau_n\}^n : \text{lab}_i \in \Sigma_{\text{lab}}, \tau_i \in \Sigma_\tau, n \in \mathbb{Z}^+ \right\}\end{aligned}$$

The simply typed lambda calculus with recursion, references, product / sum types and records, denoted $\lambda^{\rightarrow}_{\text{rec}\,+\,\text{ref}\,+\,(\times/+/\{\})}$, is defined as:

$$\begin{aligned}v \ \texttt{::=}\ & \lambda x : \tau . e \\ \texttt{|}\ & \underbrace{P^n\, v_1\ \ldots\ v_n}_{\texttt{constructed values w/ arity } n} \\ \texttt{|}\ & \underbrace{p^n\, v_1\ \ldots\ v_k}_{\texttt{partailly constructed values w/ arity } n} \qquad k < n\end{aligned}$$

$$\begin{aligned}e \ \texttt{::=}\ & x \in \Sigma_{\texttt{var}} \\ \texttt{|}\ & e_1\ e_2 \\ \texttt{|}\ & \lambda x : \tau . e \\ \texttt{|}\ & v \\ \texttt{|}\ & \texttt{let}\ x : \tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \\ \texttt{|}\ & \texttt{case}\ e\ \texttt{of}\ (C_1^{m_1} x_1 : \tau_1^{C_1} \ldots x_{m_1} : \tau_n^{C_1} \to e_1 \mid \ldots \mid C_n^{m_n} x_1 : \tau_1^{C_n} \ldots x_{m_n} : \tau_m^{C_n} \to e_n)\end{aligned}$$

- **Syntactic Sugar**:

$$\begin{aligned}&\texttt{\{ lab}_1\ \texttt{=}\ e_1\texttt{, ..., lab}_n\ \texttt{=}\ e_n\ \texttt{\}} \triangleq \texttt{\{ lab}_1 : \tau_1\texttt{, ..., lab}_n : \tau_n\ \texttt{\}}\ e_1\ \texttt{...}\ e_n \\ &\texttt{\#lab}\ e \triangleq \texttt{case}\ e\ \texttt{of (\{ ..., lab} : \tau_i\texttt{, ... \} ...}\ x_i : \tau_i\ \texttt{...} \to x_i\texttt{)}\end{aligned}$$

  given $\Sigma; \Gamma \vdash e_i : \tau_i$ and $\Sigma; \Gamma \vdash e : \{\ldots, \text{lab} : \tau_i, \ldots\}$

Definition 1.3.6. ($\lambda^{\rightarrow}_{\mathbf{rec}\,+\,\mathbf{ref}\,+\,(\times/+/\{\})}$ **Types**) The set of types $\tau \in \Sigma_\tau$ for $\lambda^{\rightarrow}_{\text{rec}\,+\,\text{ref}\,+\,(\times/+/\{\})}$ is defined by

$$\begin{aligned}\tau \ \texttt{::=}\ & \texttt{int}\ \texttt{|}\ \texttt{bool}\ \texttt{|}\ \texttt{unit} \\ \texttt{|}\ & \tau_1 \to \tau_2\ \texttt{|}\ \tau\ \texttt{ref} \\ \texttt{|}\ & \tau_1\ \texttt{+}\ \tau_2\ \texttt{|}\ \tau_1 \times \tau_2 \\ \texttt{|}\ & \texttt{\{ lab}_1 : \tau_1\texttt{, ..., lab}_n : \tau_n\ \texttt{\}}\end{aligned}$$

Definition 1.3.7. (**Small-Step Semantics of $\lambda^{\rightarrow}_{\mathbf{rec}\ +\ \mathbf{ref}\ +\ (\times/+/\{\})}$**) See definition ??. (*No additional transition rules required*).

Definition 1.3.8. (**Typing Relation**) The *typing relation* $\vdash \subseteq \Sigma_\Sigma \times \Sigma_\Gamma \times \Sigma_e \times \Sigma_\tau$, with infix notation $\Sigma; \Gamma \vdash e : \tau$, defined inductively by:

For constructors $C^n \in \Sigma_{\text{constructor}}$:

$$\vdots$$

$$\text{(Record)}\ \frac{}{\Sigma; \Gamma \vdash \{\text{lab}_1 : \tau_1, \ldots, \text{lab}_n : \tau_n\}^n : \tau_1 \to \cdots \to \tau_n \to \{\text{lab}_1 : \tau_1, \ldots, \text{lab}_n : \tau_n\}}$$

For δ-functions $\delta^n \in \Sigma_\delta$:

$$\vdots$$

For expressions $e \in \Sigma_e$, we have:

$$\vdots$$

- **Theorem**: Progress, preservation, determinism are identical. See section ??.

2 Concepts

2.1 Curry-Howard Correspondence

Theorem 2.1.1. (**Curry-Howard Correspondence**) The Curry-Howard correspondence defines a equivalence relation $\cong: \Sigma_\tau \longleftrightarrow \mathcal{L}_0(\Omega_0 \setminus \{\neg\})$

Types	Propositions
$\tau_1 \to \tau_2$	$\psi \to \phi$
$\tau_1 \times \tau_2$	$\psi \wedge \phi$
$\tau_1 + \tau_2$	$\psi \vee \phi$

If $\Gamma' \vdash_{\mathscr{P}} \psi$ in proof system $\mathscr{P}$ and $\tau \cong \psi$, and there exists $e \in \Sigma_e$ s.t $\Gamma \vdash e : \tau$ w/ $\operatorname{rng}\Gamma \cong \Gamma'$, then e is the *corresponding proof* of ψ in $\lambda^{\to}_{(\times/+)}$.

- Proofs are *constructive.*
- $\implies$ Type system of $\lambda^{\to}_{(\times/+)}$ is a *institutional* proof system for $\mathcal{L}_0(\Omega_0 \setminus \{\neg\})$:

Type System	Gentzen's Natural Deduction $\mathscr{G}_0$
$\dfrac{}{\Gamma, x : \tau \vdash x : \tau}$	$\dfrac{}{\Gamma, \psi \vdash \psi}$
$\dfrac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2}$	$\dfrac{\Gamma, \psi \vdash \phi}{\Gamma \vdash \psi \to \phi}$
$\dfrac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$	$\dfrac{\Gamma \vdash \psi \qquad \Gamma \vdash \psi \to \phi}{\Gamma \vdash \phi}$
$\dfrac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$	$\dfrac{\Gamma \vdash \psi \qquad \Gamma \vdash \phi}{\Gamma \vdash \psi \wedge \phi}$
$\dfrac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \#1\ e : \tau_1} \qquad \dfrac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \#2\ e : \tau_2}$	$\dfrac{\Gamma \vdash \psi \wedge \phi}{\Gamma \vdash \psi} \qquad \dfrac{\Gamma \vdash \psi \wedge \phi}{\Gamma \vdash \phi}$
$\dfrac{\Gamma \vdash e : \tau_1}{\Gamma \vdash (\text{inl } e : \tau_1 + \tau_2) : \tau_1 + \tau_2}$	$\dfrac{\Gamma \vdash \psi}{\Gamma \vdash \psi \vee \phi} \qquad \dfrac{\Gamma \vdash \phi}{\Gamma \vdash \psi \vee \phi}$
$\dfrac{\Gamma \vdash e : \tau_2}{\Gamma \vdash (\text{inr } e : \tau_1 + \tau_2) : \tau_1 + \tau_2}$	
$\dfrac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 \vdash e_1 : \tau \qquad \Gamma, y : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \text{case ...} : \tau}$	$\dfrac{\Gamma \vdash \psi \vee \phi \qquad \Gamma, \psi \vdash \chi \qquad \Gamma, \phi \vdash \chi}{\Gamma \vdash \chi}$

2.2 Subtyping

- **Problem**: Type system lacks polymorphism
- Types of polymorphism:
  - *Ad-hoc polymorphism*: operator overloading. e.g. Haskell type classes
  - *Parametric Polymorphism*: types contain type variables α w/ universal quantification $\overrightarrow{\forall}$. See System F, ML, etc.
  - *Subtype Polymorphism*: Polymorphism via subtype-relation $\leq$.

2.2.1 Subtype Polymorphism

- **Motivation**: Substitution principle. $\tau \leq \tau'$ iff "substitute" $e' : \tau'$ w/ $e : \tau \implies$ values of τ is a *subset* of τ'.

Definition 2.2.1. (**Denotation of Types in $\lambda^{\rightarrow}_{(\times/+/\{\})}$**) Let the universe, or *domain*, $\mathscr{U}$ be the defined by:

$$\begin{aligned} d \texttt{ ::= } & n \in \mathbb{Z} \texttt{ | true | false | ()} \\ & \texttt{| (1, } d\texttt{) | (2, } d\texttt{)} \\ & \texttt{| } (d_1, d_2) \\ & \texttt{| \{(}d_1\texttt{, } d'_1\texttt{), ..., (}d_n\texttt{, } d'_n\texttt{)\}} \\ & \texttt{| \{(lab}_1\texttt{, } d_1\texttt{), ..., (lab}_m\texttt{, } d_m\texttt{)\}} \end{aligned}$$

where $\text{lab}_i \in \Sigma_{\text{lab}}$, and $n \geq 0, m \geq 1$.

The denotation function of types, denoted $[\![\cdot]\!] : \Sigma_\tau \to \mathcal{P}(\mathscr{U})$, is inductively defined by

$$\begin{aligned} [\![\text{int}]\!] &= \mathbb{Z} \\ [\![\text{bool}]\!] &= \{\text{true}, \text{false}\} \\ [\![\text{unit}]\!] &= \{()\} \\ [\![\tau_1 \times \tau_2]\!] &= [\![\tau_1]\!] \times [\![\tau_2]\!] \\ [\![\tau_1 + \tau_2]\!] &= [\![\tau_1]\!] \uplus [\![\tau_2]\!] \\ [\![\tau_1 \to \tau_2]\!] &= \{\{(d_1, d'_1), \ldots, (d_n, d'_n)\} : d_i \in [\![\tau_1]\!] \implies d'_i \in [\![\tau_2]\!]\} \\ &= \mathcal{P}\left(\overline{[\![\tau_1]\!] \times \overline{[\![\tau_2]\!]}}\right) \\ [\![\{\text{lab}_1 : \tau_1, \ldots, \text{lab}_n : \tau_n\}]\!] &= \{\{(\text{lab}_1, d_1), \ldots, (\text{lab}_k, d_k)\} : n \leq k \wedge d_i \in [\![\tau_i]\!]\} \end{aligned}$$

- The denotation of Σ_τ defines the set of values of types w/ functions and records represented as binary relations $\mathscr{U} \longleftrightarrow \mathscr{U}, \Sigma_{\text{lab}} \longleftrightarrow \mathscr{U}$.

Definition 2.2.2. (**Subtype**) The subtyping relation $\leq : \Sigma_\tau \longleftrightarrow \Sigma_\tau$ is defined as

$$\tau_1 \leq \tau_2 \iff [\![\tau_1]\!] \subseteq [\![\tau_2]\!].$$

Lemma 2.2.1. $\leq$ is reflexive and transitive.

Proof. Follows from reflexivity and transitivity of $\subseteq$. □

Theorem 2.2.1. (**Record Subtyping**) The following hold:

$$\frac{}{\{\text{lab}_1 : \tau_1, \dots, \text{lab}_k : \tau_k, \dots, \text{lab}_n : \tau_n\} \leq \{\text{lab}_1 : \tau_1, \dots, \text{lab}_k : \tau_k\}}$$

$$\frac{\tau_1 \leq \tau_1' \quad \dots \quad \tau_n \leq \tau_n'}{\{\text{lab}_1 : \tau_1, \dots, \text{lab}_n : \tau_n\} \leq \{\text{lab}_1 : \tau_1', \dots, \text{lab}_n : \tau_n'\}}$$

$$\frac{\pi \text{ is a permutation on } [1, n]}{\{\text{lab}_1 : \tau_1, \dots, \text{lab}_n : \tau_n\} \leq \left\{\text{lab}_{\pi(1)} : \tau_{\pi(1)}, \dots, \text{lab}_{\pi(n)} : \tau_{\pi(n)}\right\}}$$

Definition 2.2.3. (**Covariance and Contravaraince**) τ_1, τ_2 are *covariant* iff $\tau_1 \leq \tau_2$. Similarly, τ_1, τ_2 are *contravariant* iff $\tau_2 \leq \tau_1$.

- Covariance: traverse down the subtype tree
- Contravariance: traverse up the subtype tree

Theorem 2.2.2. (**Subtyping of Functions**) For all $\tau_1, \tau_2, \tau_3, \tau_4 \in \Sigma_\tau$,

$$\frac{\tau_3 \leq \tau_1 \qquad \tau_2 \leq \tau_4}{\tau_1 \to \tau_2 \leq \tau_3 \to \tau_4}$$

Proof.

$$\begin{aligned}
&\tau_1 \to \tau_2 \leq \tau_3 \to \tau_4 \\
&\iff [\![\tau_1 \to \tau_2]\!] \subseteq [\![\tau_3 \to \tau_4]\!] \\
&\iff \mathcal{P}\left(\overline{[\![\tau_1]\!] \times \overline{[\![\tau_2]\!]}}\right) \subseteq \mathcal{P}\left(\overline{[\![\tau_3]\!] \times \overline{[\![\tau_4]\!]}}\right) \\
&\iff \overline{[\![\tau_1]\!] \times \overline{[\![\tau_2]\!]}} \subseteq \overline{[\![\tau_3]\!] \times \overline{[\![\tau_4]\!]}} \\
&\iff [\![\tau_1]\!] \times \overline{[\![\tau_2]\!]} \supset [\![\tau_3]\!] \times \overline{[\![\tau_4]\!]} \\
&\iff [\![\tau_3]\!] \subseteq [\![\tau_1]\!] \land [\![\tau_2]\!] \subseteq [\![\tau_4]\!]
\end{aligned}$$

□

- Function arguments are *contravariant* and return types are *covariant*.

Theorem 2.2.3. (**Subtyping of Product and Sums**)

$$\frac{\tau_1 \leq \tau_1' \qquad \tau_2 \leq \tau_2'}{\tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'} \quad \frac{\tau_1 \leq \tau_1' \qquad \tau_2 \leq \tau_2'}{\tau_1 + \tau_2 \leq \tau_1' + \tau_2'}$$

Proof. Follows from distributivity of $\subseteq$ of $\times$ and $\uplus$. $\square$

- Implement subtype polymorphism w/ subsumption rule:

$$(\text{Sub})\ \frac{\Gamma \vdash e : \tau \qquad \tau \leq \tau'}{\Gamma \vdash e : \tau'}$$

2.2.2 Objects

- **Idea**: Using references and record subtyping, we can implement *objects*.
- Split class for `counter` definition into

```
type counter_state = { mutable count: int }
let new_counter_state () = { count = 0 }

class counter () =
  object (self)
    (* state *)
    val mutable state : counter_state = new_counter_state ()

    (* behavior / methods *)
    method get () = state.count
    method inc () = state.count <- state.count + 1
  end
```

Example 2.2.1. (**Counter**)

1. Define a *state constructor*:

```
let new_counter_state : unit → counter_state
  = λx : unit. { count = ref 0 }
```

2. Define *method constructor* that implements the *methods* given a state:

```
let new_counter : counter_state → counter
    = λs : counter_state.
        { get = λx : unit. ! (#count s)
        , inc = λx : unit. (#count s) := ! (#count s) + 1
        }
```

3. The *object constructor* is the composition of the state and method constructors:

```
let counter : unit → counter = new_counter ∘ new_counter_state
```

- Inheritance is implemented using re-use of *method constructors* e.g. Reset Counter object:

```
let new_reset_counter : counter_state → reset_counter
    = λs : counter_state.
        let super : counter = new_counter s in
            { get = #get super
            , inc = #inc super
            , reset = λx : unit. (#count s) := 0
            }
```

- **Problem**: Code duplication when copying fields from super object to sub object.
- **Solution**: Extensible records.

2.3 Concurrency

- **Idea**: Extend $\lambda_{\text{rec + ref + } (\times/+/\{\})}$ w/ locks (mutexes) and concurrency.

Definition 2.3.1. ($\lambda^{\rightarrow}_{\textbf{rec + ref + } (\times/+/\{\}) \textbf{ + lock + con}}$ **Syntax**) Let Σ_{lock} be a countably infinite set of mutex symbols. Let $\Sigma_{\text{constructor}}$ be the extended set of constructors:

$$\begin{aligned}\Sigma_{\text{constructor}} = \cdots& \\ &\cup \left\{ m^0 : m \in \Sigma_{\text{lock}} \right\}\end{aligned}$$

and Σ_δ be the extended set of δ-functions:

$$\begin{aligned}\Sigma_\delta &= \cdots \\ &\cup \left\{\text{lock}^1\cdot, \text{unlock}^1\cdot\right\}\end{aligned}$$

The simply typed lambda calculus with recursion, references, product / sum types / records, locks and concurrency, denoted $\lambda^{\rightarrow}_{\text{rec + ref + }(\times/+/\{\})\text{ + lock + con}}$, is defined as:

$$\begin{aligned}
v \; \texttt{::=} \;& \lambda x : \tau . e \\
\texttt{|} \;& \underbrace{P^n \; v_1 \; \ldots \; v_n}_{\texttt{constructed values w/ arity } n} \\
\texttt{|} \;& \underbrace{p^n \; v_1 \; \ldots \; v_k}_{\texttt{partailly constructed values w/ arity } n} \qquad k < n
\end{aligned}$$

$$\begin{aligned}
e \; \texttt{::=} \;& x \in \Sigma_{\texttt{var}} \\
\texttt{|} \;& e_1 \;\; e_2 \\
\texttt{|} \;& \lambda x : \tau . e \\
\texttt{|} \;& v \\
\texttt{|} \;& \texttt{let } x : \tau \texttt{ = } e_1 \texttt{ in } e_2 \\
\texttt{|} \;& \texttt{case } e \texttt{ of } (C_1^{m_1} x_1 : \tau_1^{C_1} \ldots x_{m_1} : \tau_n^{C_1} \to e_1 \mid \ldots \mid C_n^{m_n} x_1 : \tau_1^{C_n} \ldots x_{m_n} : \tau_m^{C_n} \to e_n) \\
\texttt{|} \;& (e_1 \;\|\; e_2)
\end{aligned}$$

Definition 2.3.2. (**$\lambda^{\rightarrow}_{\text{rec + ref + }(\times/+/\{\})\text{ + lock + con}}$ Types**) The set of types $\tau \in \Sigma_\tau$ for $\lambda^{\rightarrow}_{\text{rec + ref + }(\times/+/\{\})\text{ + lock + con}}$ is defined by

$$\begin{aligned}
\tau \; \texttt{::=} \;& \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \mid \texttt{lock} \\
\texttt{|} \;& \tau_1 \to \tau_2 \mid \tau \texttt{ ref} \\
\texttt{|} \;& \tau_1 \texttt{ + } \tau_2 \mid \tau_1 \times \tau_2 \\
\texttt{|} \;& \texttt{\{ lab}_1 : \tau_1 \texttt{, ..., lab}_n : \tau_n \texttt{ \}}
\end{aligned}$$

Definition 2.3.3. (**Locking Context**) We define a locking context M as a finite partial function $M : \Sigma_{\text{lock}} \to |\mathbf{B}|$. Σ_M denotes the set of locking contexts.

Definition 2.3.4. (**Small-Step Semantics of $\lambda^{\rightarrow}_{\text{rec + ref + }(\times/+/\{\})\text{ + lock + con}}$**) A *config* is defined as the tuple $\langle e, s, M \rangle$, where $e \in \Sigma_e, s \in \Sigma_s, M \in \Sigma_M$. The set of configs is defined $\Sigma_{\text{config}} = \Sigma_e \times \Sigma_s \times \Sigma_M$.

The small-step semantics of $\lambda^{\rightarrow}_{\text{rec + ref + }(\times/+/\{\})\text{ + lock + con}}$, $\longrightarrow : \Sigma_{\text{config}} \rightharpoonup \Sigma_{\text{config}}$, is defined by

$$\vdots$$

$$\text{(Lock)}\ \frac{M(m) = 0}{\langle \text{lock } m, s, M\rangle \longrightarrow \langle (), s, (M, m \to 1)\rangle}$$

$$\text{(Unlock)}\ \frac{M(m) = 1}{\langle \text{unlock } m, s, M\rangle \longrightarrow \langle (), s, (M, m \to 0)\rangle}$$

$$\vdots$$

$$\text{(Parallel 1)}\ \frac{\langle e_1, s, M\rangle \longrightarrow \langle e_1', s', M'\rangle}{\langle (e_1 \parallel e_2), s, M\rangle \longrightarrow \langle (e_1' \parallel e_2), s', M'\rangle}$$

$$\text{(Parallel 2)}\ \frac{\langle e_2, s, M\rangle \longrightarrow \langle e_2', s', M'\rangle}{\langle (e_1 \parallel e_2), s, M\rangle \longrightarrow \langle (e_1 \parallel e_2'), s', M'\rangle}$$

$$\text{(Parallel 3)}\ \frac{}{\langle (v_1 \parallel v_2), s, M\rangle \longrightarrow \langle (v_1, v_2), s, M\rangle}$$

- **Consequences**:
  - State-space explosion: n threads w/ m states $\implies$ m^n states.
  - Non-determinism. (Parallel 1) and (Parallel 2) ensure non-determinism.
  - Deadlock. Locks may result in deadlock (See CDS) $\implies$ enforced *locking disciplines*.

Definition 2.3.5. (**Typing Relation**) The *typing relation* $\vdash \subseteq \Sigma_\Sigma \times \Sigma_\Gamma \times \Sigma_e \times \Sigma_\tau$, with infix notation $\Sigma; \Gamma \vdash e : \tau$, defined inductively by:

For constructors $C^n \in \Sigma_{\text{constructor}}$:

$$\ldots \quad \text{(Mutex)}\ \frac{}{\Sigma; \Gamma \vdash m : \text{lock}}$$

For δ-functions $\delta^n \in \Sigma_\delta$:

$$\vdots$$

$$\text{(Lock)}\ \frac{}{\Sigma; \Gamma \vdash \text{lock}\ \cdot : \text{lock} \to \text{unit}}$$

$$\text{(Unlock)}\ \frac{}{\Sigma; \Gamma \vdash \text{unlock}\ \cdot : \text{lock} \to \text{unit}}$$

For expressions $e \in \Sigma_e$, we have:

$$\vdots$$

$$\text{(Parallel)}\ \frac{\Sigma; \Gamma \vdash e_1 : \tau_1 \qquad \Sigma; \Gamma \vdash e_2 : \tau_2}{\Sigma; \Gamma \vdash (e_1 \parallel e_2) : \tau_1 \times \tau_2}$$

2.3.1 Thread Local Semantics

- **Problem**: Locking disciplines require an effect system on locks and references.
- **Solution**: Thread-local semantics / Type and Effect Systems.

Definition 2.3.6. (**Effects**) Let Σ_γ be the set of labels, defined by

$$\begin{aligned} \kappa \ &::= \ + \ \mid \ - \\ \gamma \ &::= \ \ell := v \ \mid \ !\ell = v \ \mid \ m_\kappa \end{aligned}$$

where $!\ell$ is the dereference effect, $!\ell :=$ is the assign effect, and m^κ is the effect of performing operation κ on lock m. The set of effects $\Sigma_\mathcal{E}$, is then defined by

$$\mathcal{E} \ ::= \ \emptyset \ \mid \ \gamma$$

where $\emptyset$ is the empty effect.

- Effects may be used to define $e \xrightarrow{\mathcal{E}} e'$ transitions, or *thread-local semantics*

Definition 2.3.7. (**Thread-Local Semantics**) The thread-local small-step semantics of $\lambda^{\to}_{\text{rec + ref + (×/+/\{\}) + lock + con}}$, defined by the transition relation $\xrightarrow{\mathcal{E}}: \Sigma_e \longrightarrow \Sigma_e$, is inductively defined by

$$\text{(Eval)}\ \frac{e \xrightarrow{\mathcal{E}} e'}{E[e] \xrightarrow{\mathcal{E}} E[e']}$$

$$\text{(Op }+\text{)}\ \frac{n = n_1 + n_2}{n_1 + n_2 \xrightarrow{\emptyset} n}$$

$$\text{(Op }\geq\text{)}\ \frac{b = n_1 \geq n_2}{n_1 \geq n_2 \xrightarrow{\emptyset} b}$$

$$\text{(Fix)}\ \frac{}{\text{fix } v_1\ v_2 \xrightarrow{\emptyset} v_1\ (\text{fix } v_1)\ v_2}$$

$$\text{(Seq)}\ \frac{}{(); v \xrightarrow{\emptyset} v}$$

$$\text{(Ref)}\ \frac{}{\text{ref } v \xrightarrow{\ell := v} \ell}$$

$$\text{(Deref)}\ \frac{}{!\ell \xrightarrow{!\ell = v} v}$$

$$\text{(Assign)}\ \frac{}{\ell := v \xrightarrow{\ell := v} ()}$$

$$\text{(Lock)}\ \frac{}{\text{lock } m \xrightarrow{m^+} ()}$$

$$\text{(Unlock)}\ \frac{}{\text{unlock } m \xrightarrow{m^-} ()}$$

$$(\lambda)\ \frac{}{(\lambda x : \tau . e)\ v \xrightarrow{\emptyset} \{v/x\}\, e}$$

$$\text{(Let)}\ \frac{}{\text{let } x : \tau = v \text{ in } e_2 \xrightarrow{\emptyset} \{v/x\}\, e_2}$$

$$\text{(Case)}\ \frac{}{\text{case } C_i^{m_i} v_1 \dots v_{m_i} \text{ of } (\dots \mid C_i^{m_i} x_1 : \tau_1^{C_i} \dots x_{m_i} : \tau_{m_i}^{C_i} \to e_i \mid \dots) \xrightarrow{\emptyset} \{v_1/x_1, \dots, v_{m_i}/x_{m_i}\}\, e_i}$$

$$\text{(Parallel)}\ \frac{}{(v_1 \parallel v_2) \xrightarrow{\emptyset} (v_1, v_2)}$$

where the evaluation contexts for $\lambda^{\rightarrow}_{\text{rec + ref + }(\times/+/\{\})\text{ + lock + con}}$ are defined by

$$
\begin{aligned}
E \; \texttt{::=} \; & [\cdot] \\
\texttt{|} \; & E \; e \\
\texttt{|} \; & v \; E \\
\texttt{|} \; & \texttt{let } x : \tau = E \texttt{ in } e_2 \\
\texttt{|} \; & \texttt{case } E \texttt{ of } (\ldots) \\
\texttt{|} \; & \texttt{(}E \parallel e\texttt{)} \; \texttt{|} \; \texttt{(}e \parallel E\texttt{)}
\end{aligned}
$$

Definition 2.3.8. (**Thread-Global Semantics**) The thread-global small-step semantics of $\lambda^{\rightarrow}_{\text{rec + ref + }(\times/+/\{\})\text{ + lock + con}}$, defined by the transition relation $\longrightarrow : \Sigma_{\text{config}} \nrightarrow \Sigma_{\text{config}}$ is defined by

$$\frac{e \xrightarrow{\mathcal{E}} e'}{\langle e, s, M\rangle \longrightarrow \langle e', s, M\rangle}$$

$$\frac{e \xrightarrow{\ell := v} e' \qquad \ell \in \operatorname{dom} s}{\langle e, s, M\rangle \longrightarrow \langle e', (s, \ell \to v), M\rangle}$$

$$\frac{e \xrightarrow{!\ell = v} e' \qquad (\ell, v) \in s}{\langle e, s, M\rangle \longrightarrow \langle e', s, M\rangle}$$

$$\frac{e \xrightarrow{m^+} e' \qquad M(m) = 0}{\langle e, s, M\rangle \longrightarrow \langle e', s, (M, m \to 1)\rangle}$$

$$\frac{e \xrightarrow{m^-} e' \qquad M(m) = 1}{\langle e, s, M\rangle \longrightarrow \langle e', s, (M, m \to 0)\rangle}$$

Theorem 2.3.1. The small-step operational semantics and thread-global semantics are equivalent for $\lambda^{\rightarrow}_{\text{rec + ref + }(\times/+/\{\})\text{ + lock + con}}$.

- **Consequences**: May now reason locally about the effects of each thread using transition sequences.
- For Type and Effect Systems, see supervision work.

2.3.2 Two Phase Locking

Definition 2.3.9. (**Ordered Two Phase Locking**) Let $m_\ell \in \Sigma_{\text{lock}}$ denote that lock m is *associated* w/ location ℓ. Let $\sqsubseteq : \Sigma_{\text{lock}} \longleftrightarrow \Sigma_{\text{lock}}$ be a total order on Σ_{lock}, used to define the order that locks are acquired in.

An expression $e \in \Sigma_e$ satisfies O2PL discipline iff for any sequence:

$$e \xrightarrow{\mathcal{E}_1} e_1 \xrightarrow{\mathcal{E}_2} e_2 \xrightarrow{\mathcal{E}_3} e_3 \xrightarrow{\mathcal{E}_4} \cdots ,$$

(i) For all $\mathcal{E}_i = (\ell := v)$ or $\mathcal{E}_i = (!\ell = v)$, then there exists $1 \leq j < i$ s.t $\mathcal{E}_j = m_\ell^+$.

(ii) For all $1 \leq i < j$, if $\mathcal{E}_i = m_\ell^+$ and $\mathcal{E}_j = (m')_{\ell'}^+$, then $m \sqsubset m'$.

(iii) For all $i \geq 1$, if $\mathcal{E}_i = m_\ell^+$, then there exists $i < j$ s.t $\mathcal{E}_j = m_\ell^-$

(iv) For all $i \geq 1$, if $\mathcal{E}_i = m_\ell^-$, then there does not exist $i < j$ s.t $\mathcal{E}_j = (m')_{\ell'}^+$.

- Informally:
  (i) Acquire mutex m_ℓ before accessing ℓ
  (ii) Acquire locks in order
  (iii) All locks acquired must be released
  (iv) Once a lock has been released, we cannot acquire any more

Definition 2.3.10. (**Serializable**) The expressions $e_1, \ldots, e_n \in \Sigma_e$ are *serializable* iff

$$\forall s, s' \in \Sigma_s, M, M' \in \Sigma_M.$$
$$\langle (e_1 \| \ldots \| e_n), s, M \rangle \longrightarrow^* \langle e', s', M' \rangle \not\longrightarrow$$
$$\implies \exists \text{permutation } \pi \text{ on } [1, n]. e'' \in \Sigma_e. \left\langle e_{\pi(1)}; \ldots; e_{\pi(n)}, s, M \right\rangle \longrightarrow^* \langle e'', s', M' \rangle$$

- Informally: concurrent threads are serializable iff there exists a serial execution of the expressions w/ equivalent effects.

Definition 2.3.11. (**Deadlock-Free**) The expressions $e_1, \ldots, e_n \in \Sigma_e$ are *deadlock-free* iff

$$\forall s, s' \in \Sigma_s, M, M' \in \Sigma_M.$$
$$\langle (e_1 \| \ldots \| e_n), s, M \rangle \longrightarrow^* \langle e', s', M' \rangle \not\longrightarrow$$
$$\implies \neg \exists e'' \in \Sigma_e, m \in \Sigma_{\text{lock}}. e' \xrightarrow{m^+} e''$$

- Informally: blocked concurrent threads are deadlock-free iff concurrent execution is not blocked by a waiting lock.

2.4 Semantic Equivalence

- **Motivation**: Proving equivalence of expressions $\implies$ optimizations.

Definition 2.4.1. (**Store Extension**) A store s' is an extension of s, denoted $s \rhd s'$, iff

$$\operatorname{dom} s \subseteq \operatorname{dom} s' \land \forall \ell \in \operatorname{dom} s . s(\ell) = s'(\ell).$$

- Define $s_1 \bowtie s_2 \iff s_1 \rhd s_2 \lor s_2 \rhd s_1$

Definition 2.4.2. (**Semantic Equivalence**) We define equivlance $\simeq^{\tau}_{\Sigma} : \Sigma_e \longleftrightarrow \Sigma_e$ to be $e_1 \simeq^{\tau}_{\Sigma} e_2$ iff

$$\begin{aligned}\forall s \in \Sigma_s . \Sigma \vdash s \implies & (\Sigma; \cdot \vdash e_1 : \tau \land \Sigma; \cdot \vdash e_2 : \tau) \\ & \land \Big[\big(\langle e_1, s \rangle \longrightarrow^{\omega} \land \langle e_2, s \rangle \longrightarrow^{\omega} \big) \\ & \quad \lor \big(\exists v \in \Sigma_v, s_1, s_2 \in \Sigma_s . s_1 \bowtie s_2 \land \langle e_1, s \rangle \longrightarrow^* \langle v, s_1 \rangle \land \langle e_2, s \rangle \longrightarrow^* \langle v, s_2 \rangle \big) \Big]\end{aligned}$$

where $\longrightarrow^{\omega}$ denotes a diverging sequence of $\longrightarrow$ transitions.

Lemma 2.4.1. (**Equivalence Relation $\simeq^{\tau}_{\Sigma}$**) $\simeq^{\tau}_{\Sigma}$ is an equivalence relation. For all $e_1, e_2, e_3 \in \Sigma_e$

(i) *Reflexivity*: $e_1 \simeq^{\tau}_{\Sigma} e_1$

(ii) *Symmetry*: $e_1 \simeq^{\tau}_{\Sigma} e_2 \implies e_2 \simeq^{\tau}_{\Sigma} e_1$

(iii) *Transitivity*: $e_1 \simeq^{\tau}_{\Sigma} e_2 \land e_2 \simeq^{\tau}_{\Sigma} e_3 \implies e_1 \simeq^{\tau}_{\Sigma} e_3$

Lemma 2.4.2. (**Congruence relation $\simeq^{\tau}_{\Sigma}$**) $\simeq^{\tau}_{\Sigma} : \Sigma_e \longleftrightarrow \Sigma_e$ is a congruence relation on $\lambda^{\rightarrow}_{\text{rec + ref + }(\times/+)}$, that is

$$\forall e_1, e_2 \in \Sigma_e . e_1 \simeq^{\tau}_{\Sigma} e_2 \implies (\forall C \in \Sigma_C . C[e_1] \equiv_{\Sigma} C[e_2]),$$

where $C \in \Sigma_C$ is the set of $\lambda^{\rightarrow}_{\text{rec + ref + }(\times/+/\{\})}$ contexts:

$$\begin{aligned} C \; \texttt{::=} \; & [\cdot] \\ & \texttt{|} \; C \; e \; \texttt{|} \; e \; C \\ & \texttt{|} \; \lambda x : \tau . C \\ & \texttt{|} \; \texttt{let} \; x : \tau = C \; \texttt{in} \; e \; \texttt{|} \; \texttt{let} \; x : \tau = e \; \texttt{in} \; C \\ & \texttt{|} \; \texttt{case} \; C \; \texttt{of} \; \texttt{(...)} \; \texttt{|} \; \texttt{case} \; e \; \texttt{of} \; \texttt{(...} \; \texttt{|} \; C_i^{m_i} x_1 : \tau_1^{C_i} \ldots x_{m_i} : \tau_{m_i}^{C_i} \to C \; \texttt{|} \; \texttt{...)} \end{aligned}$$

and *contextual equivalence* $e_1 \equiv_\Sigma e_2$ is defined as

$$\forall C, \tau' \in \Sigma_\tau . \Sigma ; \cdot \vdash C[e_1] : \tau' \wedge \Sigma ; \cdot \vdash C[e_2] : \tau' \implies C[e_1] \simeq_\Sigma^{\tau'} C[e_2]$$

Proof. See notes. *Induction on C w/ case analysis on $\longrightarrow^\omega$* □

- Contextual equivalence proof strategy: case analysis on $\longrightarrow^\omega$ w/ following useful lemmas.

Lemma 2.4.3. (**Store-Weakening Lemma**) The store-weakening lemma states that

$$\begin{aligned} & \forall \Sigma \in \Sigma_\Sigma, \Gamma \in \Sigma_\Gamma, e \in \Sigma_e, \tau \in \Sigma_\tau . \\ & \qquad \Sigma ; \Gamma \vdash e : \tau \implies \forall \ell \notin \operatorname{dom} \Sigma, \tau' \in \Sigma_\tau . \Sigma, \ell : \tau' ; \Gamma \vdash e : \tau \end{aligned}$$

Proof. (By rule induction on $\vdash$)

Corollary 2.4.0.1. The store-weakening corollary states that

$$\begin{aligned} & \forall \Sigma \in \Sigma_\Sigma, \Gamma \in \Sigma_\Gamma, e \in \Sigma_e, \tau \in \Sigma_\tau . \\ & \qquad \Sigma ; \Gamma \vdash e : \tau \implies (\forall \Sigma' \in \Sigma_\Sigma . \operatorname{dom} \Sigma \cap \operatorname{dom} \Sigma' = \emptyset \implies \Sigma \cup \Sigma' ; \Gamma \vdash e : \tau) \end{aligned}$$

Proof. (By rule induction on Σ')

Lemma 2.4.4. (**Store-Extension Lemma**) The store-extension lemma states that

$$\begin{aligned} & \forall e, e' \in \Sigma_e, s, s' \in \Sigma_s . \\ & \qquad \langle e, s \rangle \longrightarrow \langle e', s' \rangle \implies \forall \ell \notin \operatorname{dom} s, v \in \Sigma_v . \\ & \qquad\quad \exists s'' \in \Sigma_s . \ell \notin \operatorname{dom} s'' \wedge \langle e, (s, \ell \to v) \rangle \longrightarrow \langle e', (s'', \ell \to v) \rangle \end{aligned}$$

Proof. (By rule induction on $\longrightarrow$)

Corollary 2.4.0.2. The store-extension corollary states that

$$\begin{aligned} & \forall e, e' \in \Sigma_e, s, s' \in \Sigma_s . \\ & \qquad \langle e, s \rangle \longrightarrow \langle e', s' \rangle \implies \Big(\forall s'' \in \Sigma_s . \operatorname{dom} s \cap \operatorname{dom} s'' = \emptyset \implies \\ & \qquad\quad \exists s''' \in \Sigma_s . \operatorname{dom} s''' \cap \operatorname{dom} s'' = \emptyset \wedge \langle e, (s, s'') \rangle \longrightarrow \langle e', (s''', s'') \rangle \Big) \end{aligned}$$

Proof. (By rule induction on s'')