Operational Semantics $\boxed{e \hookrightarrow e}$

$$\frac{op\ \overline{v} \equiv v_y}{\texttt{let}\ y = op\ \overline{v}\ \texttt{in}\ e \hookrightarrow e[y \mapsto v_y]}\ \textsc{StAppOp}$$

$$\frac{e_1 \hookrightarrow e_1'}{\texttt{let}\ y = e_1\ \texttt{in}\ e_2 \hookrightarrow \texttt{let}\ y = e_1'\ \texttt{in}\ e_2}\ \textsc{StLetE1} \qquad \frac{}{\texttt{let}\ y = v\ \texttt{in}\ e \hookrightarrow e[y \mapsto v]}\ \textsc{StLetE2}$$

$$\frac{}{\texttt{let}\ y = \lambda x{:}t.e_1\ v_x\ \texttt{in}\ e_2 \hookrightarrow \texttt{let}\ y = e_1[x \mapsto v_x]\ \texttt{in}\ e_2}\ \textsc{StLetAppLam}$$

$$\frac{}{\texttt{let}\ y = \texttt{fix} f{:}t.\lambda x{:}t_x.e_1\ v_x\ \texttt{in}\ e_2 \hookrightarrow \texttt{let}\ y = (\lambda f{:}t.e_1[x \mapsto v_x])\ (\texttt{fix} f{:}t.\lambda x{:}t_x.e_1)\ \texttt{in}\ e_2}\ \textsc{StLetAppFix}$$

$$\frac{}{\texttt{match}\ d_i\ \overline{v_j}\ \texttt{with}\ \overline{d_i\ \overline{y_j} \to e_i} \hookrightarrow e_i[\overline{y_j \mapsto v_j}]}\ \textsc{StMatch}$$

Fig. 11. Small Step Operational Semantics

Basic Typing $\boxed{\Gamma \vdash_{\mathsf{t}} e : t}$

$$\frac{}{\Gamma \vdash_{\mathsf{t}} \texttt{err} : t}\ \textsc{BtErr} \qquad \frac{}{\Gamma \vdash_{\mathsf{t}} c : \mathsf{Ty}(c)}\ \textsc{BtConst} \qquad \frac{}{\Gamma \vdash_{\mathsf{t}} op : \mathsf{Ty}(op)}\ \textsc{BtOp} \qquad \frac{\Gamma(x) = t}{\Gamma \vdash_{\mathsf{t}} x : t}\ \textsc{BtVar}$$

$$\frac{\Gamma, x{:}t_1 \vdash_{\mathsf{t}} e : t_2}{\Gamma \vdash_{\mathsf{t}} \lambda x{:}t_1.e : t_1 {\to} t_2}\ \textsc{BtFun} \qquad \frac{\Gamma, f{:}t_1{\to}t_2 \vdash_{\mathsf{t}} \lambda x{:}t_1.e : t_1{\to}t_2}{\Gamma \vdash_{\mathsf{t}} \texttt{fix} f{:}(t_1{\to}t_2)\lambda x{:}t_1.e : t_1{\to}t_2}\ \textsc{BtFix}$$

$$\frac{\emptyset \vdash_{\mathsf{t}} e_1 : t_x \quad \Gamma, x{:}t_x \vdash_{\mathsf{t}} e_2 : t}{\Gamma \vdash_{\mathsf{t}} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t}\ \textsc{BtLetE} \qquad \frac{\mathsf{Ty}(op) = \overline{t_i}{\to}t_x \quad \Gamma \vdash_{\mathsf{t}} v_i : t_i \quad \Gamma, x{:}t_x \vdash_{\mathsf{t}} e : t}{\Gamma \vdash_{\mathsf{t}} \texttt{let}\ x = op\ \overline{v_i}\ \texttt{in}\ e : t}\ \textsc{BtAppOp}$$

$$\frac{\Gamma \vdash_{\mathsf{t}} v_1 : t_2{\to}t_x \quad \Gamma \vdash_{\mathsf{t}} v_2 : t_2 \quad \Gamma, x{:}t_x \vdash_{\mathsf{t}} e : t}{\Gamma \vdash_{\mathsf{t}} \texttt{let}\ x = v_1\ v_2\ \texttt{in}\ e : t}\ \textsc{BtApp}$$

$$\frac{\Gamma \vdash_{\mathsf{t}} v : t_v \quad \forall i, \mathsf{Ty}(d_i) = \overline{t_j}{\to}t_v \quad \Gamma, \overline{y_j{:}t_j} \vdash_{\mathsf{t}} e_i : t}{\Gamma \vdash_{\mathsf{t}} \texttt{match}\ v\ \texttt{with}\ \overline{d_i\ \overline{y_j} \to e_i} : t}\ \textsc{BtMatch}$$

Fig. 12. Basic Typing Rules

A OPERATIONAL SEMANTICS

The operational semantics of our core language is shown in Figure 11, which is a standard small step semantics.

B BASIC TYPING RULES

The basic typing rules of our core language is shown in Figure 12.

Typing $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash^{\mathbf{WF}} [v{:}b \mid \bot]}{\Gamma \vdash \texttt{err} : [v{:}b \mid \bot]}\ \textsc{TErr} \qquad \frac{\Gamma \vdash^{\mathbf{WF}} \mathsf{Ty}(c)}{\Gamma \vdash c : \mathsf{Ty}(c)}\ \textsc{TConst} \qquad \frac{\Gamma \vdash^{\mathbf{WF}} \mathsf{Ty}(op)}{\Gamma \vdash op : \mathsf{Ty}(op)}\ \textsc{TOp}$$

$$\frac{\Gamma \vdash^{\mathbf{WF}} [v{:}b \mid v = x]}{\Gamma \vdash x : [v{:}b \mid v = x]}\ \textsc{TVarBase} \qquad \frac{\Gamma(x) = (a{:}\tau_a \rightarrow \tau_b) \quad \Gamma \vdash^{\mathbf{WF}} a{:}\tau_a \rightarrow \tau_b}{\Gamma \vdash x : (a{:}\tau_a \rightarrow \tau_b)}\ \textsc{TVarFun}$$

$$\frac{\Gamma, x{:}\tau_x \vdash e : \tau \quad \Gamma \vdash^{\mathbf{WF}} x{:}\tau_x \rightarrow \tau}{\Gamma \vdash \lambda x{:}\lfloor \tau_x \rfloor.e : (x{:}\tau_x \rightarrow \tau)}\ \textsc{TFun}$$

$$\frac{\Gamma \vdash \lambda x{:}b.\lambda f{:}(b \rightarrow \lfloor \tau \rfloor).e : (x{:}\{v{:}b \mid \phi\} \rightarrow f{:}(x{:}\{v{:}b \mid v {<} x \land \phi\} \rightarrow \tau) \rightarrow \tau) \quad \Gamma \vdash^{\mathbf{WF}} x{:}\{v{:}b \mid \phi\} \rightarrow \tau}{\Gamma \vdash \mathsf{fix} f{:}(b \rightarrow \lfloor \tau \rfloor).\lambda x{:}b.e : (x{:}\{v{:}b \mid \phi\} \rightarrow \tau)}\ \textsc{TFix}$$

$$\frac{\begin{array}{c}\emptyset \vdash \tau <: \tau' \quad \emptyset \vdash e : \tau \\ \Gamma \vdash^{\mathbf{WF}} \tau'\end{array}}{\Gamma \vdash e : \tau'}\ \textsc{TSub} \qquad \frac{\begin{array}{c}\Gamma \vdash \tau' <: \tau \quad \Gamma \vdash \tau <: \tau' \\ \Gamma \vdash e : \tau \quad \Gamma \vdash^{\mathbf{WF}} \tau'\end{array}}{\Gamma \vdash e : \tau'}\ \textsc{TEq}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_1 \quad \Gamma \vdash e : \tau_2 \\ \Gamma \vdash \tau_1 \lor \tau_2 = \tau \quad \Gamma \vdash^{\mathbf{WF}} \tau\end{array}}{\Gamma \vdash e : \tau}\ \textsc{TMerge} \qquad \frac{\begin{array}{c}\Gamma \vdash e_x : \tau_x \quad \Gamma, x{:}\tau_x \vdash e : \tau \\ \Gamma \vdash^{\mathbf{WF}} \tau\end{array}}{\Gamma \vdash \texttt{let}\ x = e_x\ \texttt{in}\ e : \tau}\ \textsc{TLetE}$$

$$\frac{\begin{array}{c}\Gamma \vdash op : \overline{a_i{:}\{v{:}b_i \mid \phi_i\}} \rightarrow \tau_x \\ \forall i, \Gamma \vdash v_i : [v{:}b_i \mid [\phi_i]] \\ \Gamma, x{:}\tau_x[\overline{a_i \mapsto v_i}] \vdash e : \tau \\ \Gamma \vdash^{\mathbf{WF}} \tau\end{array}}{\Gamma \vdash \texttt{let}\ x = op\ \overline{v_i}\ \texttt{in}\ e : \tau}\ \textsc{TAppOp} \qquad \frac{\begin{array}{c}\Gamma \vdash v_1 : (\tau_1 \rightarrow \tau_2) \rightarrow \tau_x \\ \Gamma \vdash v_2 : \tau_1 \rightarrow \tau_2 \quad \Gamma, x{:}\tau_x \vdash e : \tau \\ \Gamma \vdash^{\mathbf{WF}} \tau\end{array}}{\Gamma \vdash \texttt{let}\ x = v_1\ v_2\ \texttt{in}\ e : \tau}\ \textsc{TAppFun}$$

$$\frac{\begin{array}{c}\Gamma \vdash v_1 : a{:}\{v{:}b \mid \phi\} \rightarrow \tau_x \\ \Gamma \vdash v_2 : [v{:}b \mid \phi] \\ \Gamma, x{:}\tau_x[a \mapsto v_2] \vdash e : \tau \\ \Gamma \vdash^{\mathbf{WF}} \tau\end{array}}{\Gamma \vdash \texttt{let}\ x = v_1\ v_2\ \texttt{in}\ e : \tau}\ \textsc{TApp} \qquad \frac{\begin{array}{c}\Gamma \vdash v : \tau_v \quad \Gamma \vdash^{\mathbf{WF}} \tau \quad \Gamma, \overline{y{:}\tau_y} \vdash d_i(\overline{y}) : \tau_v \\ \Gamma, \overline{y{:}\tau_y} \vdash e_i : \tau\end{array}}{\Gamma \vdash (\texttt{match}\ v\ \texttt{with}\ \overline{d_i\ \overline{y} \rightarrow e_i}) : \tau}\ \textsc{TMatch}$$

Fig. 13. Full Typing Rules

C COVERAGE TYPING RULES

The full set of coverage typing rules of our core language is shown in Figure 13. The rule TOp (which is similar with TConst), TAppFun and TAppOp (which is similar with TApp) are not shown in Section 4.

Algorithm 2: Disjunction and Conjunction

i **Procedure** $\mathsf{Disj}(\tau_1, \tau_2)$:=
  match τ_1, τ_2:
    case $[v{:}t \mid \phi_1], [v{:}t \mid \phi_2]$ **do**
      return $[v{:}t \mid \phi_1 \lor \phi_2]$;
    case $\{v{:}t \mid \phi_1\}, \{v{:}t \mid \phi_2\}$ **do**
      return $[v{:}t \mid \phi_1 \land \phi_2]$;
    case $a{:}\tau_{a_1} \to \tau_1, a{:}\tau_{a_2} \to \tau_2$ **do**
      $\tau_a \leftarrow \mathsf{Conj}(\tau_{a_1}, \tau_{a_2})$;
      return $a{:}\tau_a \to \mathsf{Disj}(\tau_1, \tau_2)$;

Procedure $\mathsf{Conj}(\tau_1, \tau_2)$:=
  match τ_1, τ_2:
    case $[v{:}t \mid \phi_1], [v{:}t \mid \phi_2]$ **do**
      return $[v{:}t \mid \phi_1 \land \phi_2]$;
    case $\{v{:}t \mid \phi_1\}, \{v{:}t \mid \phi_2\}$ **do**
      return $[v{:}t \mid \phi_1 \lor \phi_2]$;
    case $a{:}\tau_{a_1} \to \tau_1, a{:}\tau_{a_2} \to \tau_2$ **do**
      $\tau_a \leftarrow \mathsf{Disj}(\tau_{a_1}, \tau_{a_2})$;
      return $a{:}\tau_a \to \mathsf{Conj}(\tau_1, \tau_2)$;

D SUBSET RELATION OF DENOTATION UNDER TYPE CONTEXT

The subset relation between the denotation of two refinement types τ_1 and τ_2 under a type context Γ (written $[\![\tau_1]\!]_\Gamma \subseteq [\![\tau_1]\!]_\Gamma$) is:

$$
\begin{aligned}
[\![\tau_1]\!]_\emptyset \subseteq [\![\tau_2]\!]_\emptyset &\doteq [\![\tau_1]\!] \subseteq [\![\tau_2]\!] \\
[\![\tau_1]\!]_{x:\tau_x,\Gamma} \subseteq [\![\tau_1]\!]_{x:\tau_x,\Gamma} &\doteq \forall v_x \in [\![\tau_x]\!], \\
&\quad [\![\tau_1[x \mapsto v_x]]\!]_{\Gamma[x\mapsto v_x]} \subseteq [\![\tau_2[x \mapsto v_x]]\!]_{\Gamma[x\mapsto v_x]} \quad \text{if } \tau \equiv \{v{:}b \mid \phi\} \\
[\![\tau_1]\!]_{x:\tau_x\Gamma} \subseteq [\![\tau_2]\!]_{x:\tau_x\Gamma} &\doteq \exists \hat{e}_x \in [\![\tau_x]\!], \forall v_x, \hat{e}_x \hookrightarrow^* v_x \implies \\
&\quad [\![\tau_1[x \mapsto v_x]]\!]_{\Gamma[x\mapsto v_x]} \subseteq [\![\tau_2[x \mapsto v_x]]\!]_{\Gamma[x\mapsto v_x]} \quad \text{otherwise}
\end{aligned}
$$

The way we interpret the type context Γ here is the same as the definition of the type denotation under the type context, but we keep the denotation of τ_1 and τ_2 as the subset relation under the same interpretation of Γ, that is under the *same* substitution $[x \mapsto v_x]$. This constraint is also required by other refinement type systems, which define the denotation of the type context Γ as a set of substitutions, with the subset relation of the denotation of two types holding under the *same* substitution. However, our type context is more complicated, since it has both under- and overapproximate types that are interpreted via existential and universal quantifiers, and cannot simply be denoted as a set of substitution. Thus, we define a subset relation over denotations under a type context to ensure the ame substitution is applied to both types.

E BIDIRECTIONAL TYPING RULES

The full set of bidirectional typing rules of our core language is shown in Figure 14 and Figure 15. Similar to other refinement type systems, there are no synthesis rules for functions which require synthesis of a refinement type for the input argument. The user can only type check functions against given types (ChkFun and ChkFix).

F ALGORITHM DETAILS

Disjunction Function. We implement our disjunction function Disj as a function with type $\mathsf{Disj} : \tau \to \tau \to \tau$. The disjunction of multiple types is equal to defined compositionally:

$$\mathsf{Disj}(\tau_1, \tau_2, ..., \tau_{n-1}, \tau_n) \doteq \mathsf{Disj}(\tau_1, \mathsf{Disj}(\tau_2, ..., \mathsf{Disj}(\tau_{n-1}, \tau_n)))$$

As shown in Algorithm 2, the Disj and Conj functions call each other recursively. As discussed in Section 4, the disjunction of two base coverage type (underapproximate type) $[v{:}t \mid v = 1]$

Type Synthesis

$$\boxed{\Gamma \vdash e \Rightarrow \tau}$$

$$\frac{\Gamma \vdash^{\mathbf{WF}} \mathsf{Ty}(c)}{\Gamma \vdash c \Rightarrow \mathsf{Ty}(c)}\ \textsc{SynConst} \qquad \frac{\Gamma \vdash^{\mathbf{WF}} \mathsf{Ty}(op)}{\Gamma \vdash op \Rightarrow \mathsf{Ty}(op)}\ \textsc{SynOp} \qquad \frac{\Gamma \vdash^{\mathbf{WF}} [v{:}b \mid \bot]}{\Gamma \vdash \mathtt{err} \Rightarrow [v{:}b \mid \bot]}\ \textsc{SynErr}$$

$$\frac{\Gamma \vdash^{\mathbf{WF}} [v{:}b \mid v = x]}{\Gamma \vdash x \Rightarrow [v{:}b \mid v = x]}\ \textsc{SynVarBase} \qquad \frac{\Gamma(x) = (a{:}\tau_a{\to}\tau_b) \quad \Gamma \vdash^{\mathbf{WF}} a{:}\tau_a{\to}\tau_b}{\Gamma \vdash x \Rightarrow (a{:}\tau_a{\to}\tau_b)}\ \textsc{SynVarFun}$$

$$\frac{\begin{array}{c}\Gamma \vdash v_1 \Rightarrow (a{:}\tau_a{\to}\tau_b){\to}\tau_x \\ \Gamma \vdash v_2 \Leftarrow a{:}\tau_a{\to}\tau_b \quad \Gamma' = x{:}\tau_x \\ \Gamma, \Gamma' \vdash e \Rightarrow \tau \quad \tau' = \mathsf{Ex}(\Gamma', \tau) \\ \Gamma \vdash^{\mathbf{WF}} \tau'\end{array}}{\Gamma \vdash \mathtt{let}\ x = v_1\ v_2\ \mathtt{in}\ e \Rightarrow \tau'}\ \textsc{SynAppFun} \qquad \frac{\begin{array}{c}\Gamma \vdash v_1 \Rightarrow a{:}\{v{:}b \mid \phi\} \to \tau_x \\ \Gamma' = a{:}[v{:}b \mid v = v_2 \wedge \phi], x{:}\tau_x \\ \Gamma, \Gamma' \vdash e \Rightarrow \tau \quad \tau' = \mathsf{Ex}(\Gamma', \tau) \\ \Gamma \vdash^{\mathbf{WF}} \tau'\end{array}}{\Gamma \vdash \mathtt{let}\ x = v_1\ v_2\ \mathtt{in}\ e \Rightarrow \tau'}\ \textsc{SynAppBase}$$

$$\frac{\begin{array}{c}\Gamma \vdash op \Rightarrow \overline{a_i{:}\{v{:}b_i \mid \phi_i\}} \to \tau_x \\ \Gamma' = \overline{a_i{:}[v{:}b_i \mid v = v_i \wedge \phi_i]}, x{:}\tau_x \\ \Gamma, \Gamma' \vdash e \Rightarrow \tau \quad \tau' = \mathsf{Ex}(\Gamma', \tau) \\ \Gamma \vdash^{\mathbf{WF}} \tau'\end{array}}{\Gamma \vdash \mathtt{let}\ x = op\ \overline{v_i}\ \mathtt{in}\ e \Rightarrow \tau'}\ \textsc{SynAppOp} \qquad \frac{\begin{array}{c}\Gamma \vdash e_x \Rightarrow \tau_x \quad \Gamma' = x{:}\tau_x \\ \Gamma, \Gamma' \vdash e \Rightarrow \tau \quad \tau' = \mathsf{Ex}(\Gamma', \tau) \\ \Gamma \vdash^{\mathbf{WF}} \tau'\end{array}}{\Gamma \vdash \mathtt{let}\ x = e_x\ \mathtt{in}\ e \Rightarrow \tau}\ \textsc{SynLetE}$$

$$\frac{\begin{array}{c}\forall i, \mathsf{Ty}(d_i) = \overline{y{:}\{v{:}b_y \mid \theta_y\}}{\to}[v{:}b \mid \psi_i] \quad \Gamma'_i = \overline{y{:}[v{:}b_y \mid \theta_y]}, a{:}[v{:}b \mid v = v_a \wedge \psi_i] \\ \Gamma, \Gamma'_i \vdash e_i \Rightarrow \tau_i \quad \tau'_i = \mathsf{Ex}(\Gamma'_i, \tau_i) \quad \Gamma \vdash^{\mathbf{WF}} \mathsf{Disj}(\overline{\tau'_i})\end{array}}{\Gamma \vdash \mathtt{match}\ v_a\ \mathtt{with}\ \overline{d_i\ \overline{y} \longrightarrow e_i} \Rightarrow \mathsf{Disj}(\overline{\tau'_i})}\ \textsc{SynMatch}$$

Fig. 14. Typing Synthesis Rules

Type Check

$$\boxed{\Gamma \vdash e \Leftarrow \tau}$$

$$\frac{\emptyset \vdash e \Rightarrow \tau \quad \Gamma \vdash \tau <: \tau' \quad \Gamma \vdash^{\mathbf{WF}} \tau'}{\Gamma \vdash e \Leftarrow \tau'}\ \textsc{ChkSub} \qquad \frac{\Gamma, x{:}\tau_x \vdash e \Leftarrow \tau \quad \Gamma \vdash^{\mathbf{WF}} x{:}\tau_x{\to}\tau}{\Gamma \vdash \lambda x{:}\lfloor \tau_x \rfloor . e \Leftarrow (x{:}\tau_x{\to}\tau)}\ \textsc{ChkFun}$$

$$\frac{\begin{array}{c}\forall i, \mathsf{Ty}(d_i) = \overline{y{:}\{v{:}b_y \mid \theta_y\}}{\to}[v{:}b \mid \psi_i] \quad \Gamma'_i = \overline{y{:}[v{:}b_y \mid \theta_y]}, a{:}[v{:}b \mid v = v_a \wedge \psi_i] \\ \Gamma, \Gamma'_i \vdash e_i \Rightarrow \tau_i \quad \tau'_i = \mathsf{Ex}(\Gamma'_i, \tau_i) \quad \Gamma \vdash \mathsf{Disj}(\overline{\tau'_i}) <: \tau' \quad \Gamma \vdash^{\mathbf{WF}} \tau'\end{array}}{\Gamma \vdash \mathtt{match}\ v_a\ \mathtt{with}\ \overline{d_i\ \overline{y} \longrightarrow e_i} \Leftarrow \tau'}\ \textsc{ChkMatch}$$

$$\frac{\Gamma \vdash \lambda x{:}b.\lambda f{:}(b{\to}\lfloor \tau \rfloor).e \Leftarrow (x{:}\{v{:}b \mid \phi\}{\to}f{:}(x{:}\{v{:}b \mid v{<}x \wedge \phi\} \to \tau){\to}\tau) \quad \Gamma \vdash^{\mathbf{WF}} x{:}\{v{:}b \mid \phi\} \to \tau}{\Gamma \vdash \mathtt{fix} f{:}(b{\to}\lfloor \tau \rfloor).\lambda x{:}b.e \Leftarrow (x{:}\{v{:}b \mid \phi\} \to \tau)}\ \textsc{ChkFix}$$

Fig. 15. Typing Synthesis Rules

and $[v{:}t \mid v = 2]$ takes the disjunction of their qualifiers: $[v{:}t \mid v = 1 \vee v = 2]$. On the other hand, the disjunction of normal refinement types (overapproximate types) is the conjunction of their corresponding qualifiers. The disjunction of function types conjuncts their argument type and disjuncts their return type.

Algorithm 3: Exists and Forall

```
Procedure Ex(x, [v:t | φ_x], τ) :=
    match τ:
        case [v:t | φ] do
            return [v:t | ∃x:t, φ_x[v ↦ x] ∧ φ];
        case {v:t | φ} do
            return {v:t | ∀x:t, φ_x[v ↦ x] ⟹ φ};
        case a:τ_a→τ do
            τ'_a ← Fa(x, [v:t | φ_x], τ_a);
            return a:τ'_a→Ex(x, [v:t | φ_x], τ);

Procedure Fa(x, [v:t | φ_x], τ) :=
    match τ:
        case [v:t | φ] do
            return [v:t | ∀x:t, φ_x[v ↦ x] ⟹ φ];
        case {v:t | φ} do
            return {v:t | ∃x:t, φ_x[v ↦ x] ∧ φ};
        case a:τ_a→τ do
            τ'_a ← Ex(x, [v:t | φ_x], τ_a);
            return a:τ'_a→Fa(x, [v:t | φ_x], τ);
```

"Exists" Function. We implement our "Exists" function Ex as a function with type $\text{Ex}(x, \tau_x, \tau) : Var \to \tau \to \tau \to \tau$, where x and τ_x is a variable and corresponding binding type that we want to existentialize into the type τ, thus it can also be notated as $\text{Ex}(x{:}\tau_x,\ \tau)$. Existentializing a type context $x_1{:}\tau_1, x_2{:}\tau_2, ...x_n{:}\tau_n$ into a type τ is equal to existentializing each binding consecutively:

$$\text{Ex}(x_1{:}\tau_1, x_2{:}\tau_2, ...x_n{:}\tau_n,\ \tau) \doteq \text{Ex}(x_1{:}\tau_1,\ \text{Ex}(x_1{:}\tau_1,\ ...,\ \text{Ex}(x_n{:}\tau_n,\ \tau)))$$

As shown in Algorithm 2, the Ex function relys on the Fa function. More specifically, as we mentioned in Section 5, existentializing a binding $x{:}[v{:}nat \mid v > 0]$ into type $[v{:}nat \mid v = x + 1]$ will derive the type $[v{:}nat \mid \exists x, x > 0 \land v = x + 1]$ which has an existentially-quantified qualifier; the function type is contravariant in its argument types and covariant in its return types.

SMT Query Encoding for data types. In order to reason over data types, we allow the user to specify refinement types with method predicates (e.g., mem) and quantifiers ($e.g.$, $\forall u, \neg mem(v, u)$). These method predicates are encoded as uninterpreted functions. In order to ensure the query is an EPR sentence, we require that a normal refinement type (overapproximate types) can only use universal quantifiers. In addition, as shown in Figure 3, we disallow nested method predicate application (e.g., $mem(v, mem(v, u))$) and can only apply a method predicate over constants $mem(v, 3)$ (it can be encoded as $\forall u, u = 3 \implies mem(v, u)$).