A Examples in Scripting Languages

This appendix presents translations of the examples in section 3 into real-world scripting languages.

Example 1

Prototype inheritance:

```
let Rect = { "area": func(self:?) . self["x"] * self["y"], "parent": null } in
let Cuboid = { "parent": Rect,
               "vol": func(self) . self["area"](self) * self["z"] } in
let shape = { "x": 2, "y": 5, "z": 10: "parent": Cuboid } in
let vol = shape["vol"](shape) // vol is 100
```

JavaScript Here is the equivalent program in JavaScript. Note that the `this` argument is implicit:

```
var Rect =   { area: function() { return this.x * this.y; },
               __proto__: null };
var Cuboid = { __proto__: Rect,
               vol: function() { return this.area() * this.z; } };
var shape = { x: 2, y: 5, z: 10, __proto__: Cuboid };
var vol = shape.vol(); // vol is 100
var f = vol;
var vol = f(); // ERROR: this.area is undefined
```

Lua This program can be written in Lua using *metatables*, which allow assigning a `parent`-like field:

```
Rect =   { area = function(self) return self.x * self.y end }
Cuboid = { vol = function(self) return self.area(self) * self.z end }
setmetatable(Cuboid, {__index = Rect})
shape = { x=2, y=5, z=10 }
setmetatable(shape, {__index = Cuboid})
vol = shape.vol(shape)
f = shape.vol
vol2 = f() // ERROR: attempt to index local self (a nil value)
```

Example 2

Extracting methods:

```
let ArrParent = { "slice": func(self:?,begin:?,end:?). ··· , ··· } in
let arr1 = { "0": 3, "1": 20, "2": 59, "length": 3, "parent": ArrParent }
let nodeList = { "0": htmlElementA, "1": htmlElementB, "2": htmlElementC,
                 "len": 3, "parent": HTMLNodeListParent } in
let eltArray = ArrParent["slice"](nodeList,0,1)
// returns an array containing htmlElementA and htmlElementB
```

JavaScript This version of slice is built-in. We use DOM-manipulation functions to fetch array-like objects.

```
var ArrParent = Array.prototype;
var arr1 = [3, 20, 59]; // JavaScript desugars to an Array object
// Get all the links on a page:
var nodeList = document.getElementsByTagName("a");
// Using .call on a function allows us to provide the this arg
var eltArray = ArrParent.slice.call(nodeList, 0, 1)
// eltArray contains the first two elements in the list
```

Lua Lua objects allow trivial method extraction—Lua has similar array behavior to JavaScript as well, and any number-indexed dictionary can be used by library methods.

```
arr = {123, 45, 6}
table.sort(arr)
-- arr is now {1 = 6, 2 = 45, 3 = 123}
not_arr = {foo = "bar"}
not_arr[1] = 6
not_arr[2] = 5
not_arr[3] = 4
table.sort(not_arr)
-- not_arr is now {1 = 4, 2 = 5, 3 = 6, foo = "bar"}
```

Example 3

Bound methods:

```
let Rect = { "area": func(self:?) . self["x"] * self["y"], "parent": null } in
let Cuboid = { "parent": Rect,
               "vol": func(self) . self["area"](self) * self["z"] } in
let rec shape2 = {
  "x": 2, "y": 5, "z": 10",
  "_class_": Cuboid,
  "parent": {
    "vol": func() . shape2["_class_"]["vol"](shape2)
  }
} in
let f = shape2["vol"]
let vol2 = f() // vol2 is still 100, f closes over shape2
```

Python In Python, we can see this effect with classes:

```
class Rect(object):
  def area(self): return self.x * self.y
class Cuboid(Rect):
```

```
  def vol(self): return self.area() * self.z
shape2 = Cuboid()
shape2.x = 2; shape2.y = 5; shape2.z = 10
f = shape2.vol
vol2 = f() # vol2 is 100
```

Ruby In Ruby, we use `obj.method(:methname)` to access the method, and `method.call` to invoke it:

```
class Rect
  def area; self.x * self.y; end
end
class Cuboid < Rect
  def vol; self.area() * self.z; end
end
shape2 = Cuboid.new
def shape2.x; 2; end
def shape2.y; 5; end
def shape2.z; 10; end
f = shape2.method(:vol)
vol2 = f.call() # vol2 is 100
```

Example 4

Ad hoc private fields:

```
let safeGetField = Λα <: ?.func(obj:?,fieldName:?,default:?).
  if (fieldName matches "_.*_") default
  else if (obj hasfield fieldName) obj[fieldName] else default in
safeGetField(?)({ "_private_": 42, "pub": 23, "parent": null },
              "_private_", 0) // returns 0
```

JavaScript In JavaScript, this check could be performed with a regex. For a real-world example, see `reject_name` in ADsafe at `https://github.com/douglascrockford/ADsafe/blob/master/adsafe.js#L254`.

```
function safeGetField(obj, field, default) {
  if(/_(.*)_/.test(field))        return default
  else {
    if(obj.hasOwnProperty(field)) return obj[field];
    else                          return default;
  }
}
```

Python A similar check works in Python. Note that variations on this pattern are found in production code inside Django, for example: `https://github.com/django/django/blob/master/django/db/models/base.py#L157`.

```
def safeGetField(obj, field, default):
  rx = re.compile(r"_(.*)_")
  if rx.match(field) is not None: return default
  else:
    if hasattr(obj, field):       return getattr(obj, field)
    else:                         return default
```

Ruby Note that several variations on this pattern are found in production code inside Ruby on Rails, for example: `https://github.com/rails/rails/blob/master/activerecord/lib/active_record/base.rb#L1725`.

```
def safeGetField(obj, field, default)
  return default unless /_(.*)_/.match(field).nil?
  return default unless obj.respond_to?(field)
  return obj.send(field.intern)
end
```

Example 5

Safe dictionary lookup:

```
let safeAssign = Λα <: ?.func(dict:?,word:Str,value:?).dict["w_" + word = value]
let safeLookup = Λα <: ?.func(dict:?,world:Str,default:?).
  let lookup = "w_" + word in
  if (dict hasfield lookup) dict[lookup]
  else default
```

JavaScript While JavaScript does not have type abstraction, implementation of the core functionality is trivial:

```
function safeAssign(dict, word, value) { dict["w_" + word] = value; }
function safeLookup(dict, word, default) {
  var lookup = "w_" + word;
  if(dict.hasOwnProperty(word)) return dict[lookup];
  return default;
}
```

Such an implementation is necessary in JavaScript when objects are used as dicitonaries, because of the presence of the `__proto__` field in major browsers.

Python Python and Ruby both support dictionary-like objects natively, and don't need to use this pattern.

B Definitions

Notation In these proofs, we write $\mathsf{func}(x{:}T)\ \texttt{\{}\ e\ \texttt{\}}$ instead of $\mathsf{func}(x{:}T)\ \texttt{.}\ e$.

Definition 1 (Type Equivalence) *We define a relation on types* $=_T$.

$$\textit{Equiv-Obj}\;\dfrac{\begin{array}{c} M_A \subseteq L_A \qquad L_A \subseteq M_A \qquad \forall i,j.L_i \cap M_j \neq \emptyset \Rightarrow S_i =_T T_j \wedge p_i = q_j \\ \forall i.\exists(j_1,\cdots,j_k).L_i \subseteq \bigcup_{l\in(j_1,\cdots,j_k)} M_l \qquad \forall j.\exists(i_1,\cdots,i_k).M_i \subseteq \bigcup_{l\in(i_1,\cdots,i_k)} L_l \end{array}}{\{L_1^{p_1} : S_1, \cdots, L_n^{p_n} : S_n, L_A : \textbf{abs}\} =_T \{M_1^{q_1} : T_1, \cdots, M_m^{q_m} : T_m, M_A : \textbf{abs}\}}$$

The other cases of $=_T$ *are trivial; to define them we lift Equiv-Obj in the natural way over the other types.* $=_T$ *describes an equivalence class of types—we use* $T_1 =_T T_2$ *and* $T_1 = T_2$ *interchangeably in this document, and types represented by the same letter are assumed to be related by* $=_T$.

Definition 2 (Subtyping) *The generating function,*

$$\mathcal{ST} : \mathcal{P}(\Gamma \times \mathcal{T} \times \mathcal{T} + p \times p) \to \mathcal{P}(\Gamma \times \mathcal{T} \times \mathcal{T} + p \times p)$$

is defined co-inductively by the subtyping judgments. We define subtyping as $\Gamma \vdash S <: T$, *iff* $(\Gamma, S, T) \in \nu\mathcal{ST}$ *and* $p <: q$, *iff* $(p, q) \in \nu\mathcal{ST}$.

Definition 3 (Transitivity) *For* $R \subseteq \mathcal{P}(\Gamma \times \mathcal{T} \times \mathcal{T} + p \times p)$

$$\begin{aligned} TR(R) = {} & \{(\Gamma, x, z) \mid \forall x, z \in \mathcal{T}, \exists y \in \mathcal{T}, (\Gamma, x, y), (\Gamma, y, z) \in R\} \\ & \cup \{(x, z) \mid \forall x, z \in p, \exists y \in p, (x, y), (y, z) \in R\} \end{aligned}$$

Definition 4 (Top-down Subexpressions) S *is a top-down subexpression of* T, *written* $S \sqsubseteq T$, *if* (S, T) *is in* μTD, *defined as follows:*

$$\begin{aligned} TD(R) = {} & \{(T,T) \mid T \textit{ is a finite type }\} \\ & \cup \{(S, \{L^p : T, rest\cdots\}) \mid (S,T) \in R\} \\ & \cup \{(S, \{L^p : T, rest\cdots\} \mid (S, \{rest\}) \in R\} \\ & \cup \{(S, \{L^p : T\}) \mid (S,T) \in R\} \\ & \cup \{(S, \mathsf{Ref}\, T) \mid (S,T) \in R\} \\ & \cup \{(S, \mu x.T) \mid (S, T[x/\mu x.T]) \in R\} \\ & \cup \{(S, T_1 \to T_2) \mid (S, T_1) \in R\} \\ & \cup \{(S, T_1 \to T_2) \mid (S, T_2) \in R\} \\ & \cup \{(S, \forall\alpha <: U.T) \mid (S, U) \in R\} \\ & \cup \{(S, \forall\alpha <: U.T) \mid (S, T) \in R\} \end{aligned}$$

Definition 5 (Bottom-Up Subexpressions) *S is a bottom-up subexpression of T, written $S \preceq T$, if (S, T) is in μBU, defined as follows:*

$$\begin{aligned}
BU(R) = {} & \{(T, T) \mid T \text{ is a finite type } \} \\
& \cup \{(S, \{L^p : T, rest \cdots\}) \mid (S, T) \in R\} \\
& \cup \{(S, \{L^p : T, rest \cdots\} \mid (S, \{rest\}) \in R\} \\
& \cup \{(S, \{L^p : T\}) \mid (S, T) \in R\} \\
& \cup \{(S, \mathit{Ref}\, T) \mid (S, T) \in R\} \\
& \cup \{(S, T_1 \to T_2) \mid (S, T_1) \in R\} \\
& \cup \{(S, T_1 \to T_2) \mid (S, T_2) \in R\} \\
& \cup \{(S, \forall \alpha <: U.T) \mid (S, U) \in R\} \\
& \cup \{(S, \forall \alpha <: U.T) \mid (S, T) \in R\} \\
& \cup \{(S[x/\mu x.T], \mu x.T) \mid (S, T) \in R\}
\end{aligned}$$

Definition 6 (Active Expressions) *Active expressions, ae, are defined as follows:*

$$\begin{aligned}
ae = {} & v_1\texttt{(}v_2\texttt{)} \\
\mid {} & \textsf{\textbf{fix}}\ \texttt{(}f\texttt{:}T\texttt{)}\texttt{.}e \\
\mid {} & v_1\texttt{+}v_2 \\
\mid {} & \textsf{\textbf{ref}}\ v \\
\mid {} & \textsf{\textbf{deref}}\ v \\
\mid {} & v_1\ \texttt{=}\ v_2 \\
\mid {} & v_1\texttt{[}v_2\texttt{]} \\
\mid {} & v_1\texttt{[}v_2\ \texttt{=}\ v_3\texttt{]} \\
\mid {} & \textsf{\textbf{delete}}\ v_1\texttt{[}v_2\texttt{]} \\
\mid {} & \textsf{\textbf{if}}\ \texttt{(}v_1\texttt{)}\ \texttt{\{}\ e_2\ \texttt{\}}\ \textsf{\textbf{else}}\ \texttt{\{}\ e_3\ \texttt{\}} \\
\mid {} & v_1\ \textsf{\textbf{hasfield}}\ v_2 \\
\mid {} & v_1\ \textsf{\textbf{matches}}\ v_2 \\
\mid {} & \textsf{\textbf{fieldin}}\ \texttt{\{}\ s_1 : v_1\texttt{,}\ s_2 : v_2\ \cdots\texttt{\}}\ \textsf{\textbf{init}}\ v_{acc}\ \textsf{\textbf{do}}\ v_f
\end{aligned}$$

C Auxilliary Lemmas

Lemma 2 *For all T, $\mathcal{T}_T^{\downarrow} \subseteq \mathcal{T}_T^{\uparrow}$*

Proof This is exactly the same argument as Pierce [19]. ■

Lemma 3 *The set of top-down subexpressions, $\mathcal{T}_T^{\downarrow} = \{S \mid (S, T) \in \mu TD\}$, is finite for all T.*

Proof By lemma 2, $\mathcal{T}_T^{\downarrow} \subseteq \mathcal{T}_T^{\uparrow}$ for all T, and by lemma 4, $\mathcal{T}_T^{\uparrow}$ is finite for all T, so $\mathcal{T}_T^{\downarrow}$ is finite for all T.

Lemma 4 *The set of bottom-up subexpressions, $\mathcal{T}_T^{\uparrow} = \{S \mid (S, T) \in \mu BU\}$ is finite for all T.*

Proof The second position in the each right-hand clause in BU is smaller than the left.

Lemma 5 *If* $(S, T[x/U]) \in \mathcal{T}^{\uparrow}_{T[x/U]}$*, then either* $(S, U) \in \mathcal{T}^{\uparrow}_{U}$ *or* $S = S'[x/U]$ *for some* $(S', T) \in \mathcal{T}^{\uparrow}_{T}$*.*

Proof By case analysis on T.

D Subtyping

Lemma 6 *If:*

$$\mathcal{S} : \mathcal{P}(\mathit{\Gamma} \times \mathcal{T} \times \mathcal{T} + p \times p) \to \mathcal{P}(\mathit{\Gamma} \times \mathcal{T} \times \mathcal{T} + p \times p)$$

is a monotone function, and for all R*,* $TR(\mathcal{S}(R)) \subseteq \mathcal{S}(TR(R))$*, then* $\nu\mathcal{S}$ *is transitive.*

Proof: This definition is from Gapayev, et al., and reproduced in Pierce's text. Its relation to transitivity is discussed there—we use it as a goal and defer to their explanation to complete the proof [19, Lemma 21.3.6].

Lemma 7 (Subtyping is Transitive) $TR(\nu\mathcal{ST}) \subseteq \nu\mathcal{ST}$

Proof:

For arbitrary R, we consider both field annotations in (p, q), and subtyping judgments $(\mathit{\Gamma}, S, T)$:

Let $(p, q) \in TR(\mathcal{ST}(R))$. By definition of TR, there exists a p' such that $(p, p'), (p', q) \in \mathcal{ST}(R)$. By case analysis of the $p <: p$ rules, it follows trivially that $(p, q) \in \mathcal{ST}(TR(R))$.

Let $(\mathit{\Gamma}, S, T) \in TR(\mathcal{ST}(R))$. By definition of TR, there exists a U such that $(\mathit{\Gamma}, S, U), (\mathit{\Gamma}, U, T) \in \mathcal{ST}(R)$. We will show that $(\mathit{\Gamma}, S, T) \in \mathcal{ST}(TR(R))$, so that by lemma 6, $\nu\mathcal{ST}$ is transitive.

By case-analysis on the possible shapes of U (eliding the trivial cases where $T = \top$):

- $U = \{L_1^{p_1} : U_1, \cdots, L_n^{p_n} : U_n, L_A : \mathbf{abs}\}$.
  Since $(\mathit{\Gamma}, S, U), (\mathit{\Gamma}, U, T) \in \mathcal{ST}(R)$, by cases of $\mathcal{ST}$,
  (1) $S = \{K_1^{o_1} : S_1, \cdots, K_l^{o_l} : S_l, K_A : \mathbf{abs}\}$, and
  (2) $T = \{M_1^{q_1} : T_1, \cdots, M_m^{q_m} : T_m, M_A : \mathbf{abs}\}$.
  By hypothesis, $(\mathit{\Gamma}, S, U) \in \mathcal{ST}(R)$, which must be by S-Object. By hypothesis of S-Object:
  (3) $\forall i, j.$ if $K_i \cap L_j \neq \emptyset$ then $(S_i, U_j) \in R$ and $(o_i, p_j) \in R$,
  (4) $\bigcup_i^{1 \cdots n} L_i \subseteq \bigcup_h^{1 \cdots l} K_h \cup K_A$,
  (4') $L_A \subseteq K_A$,
  (5) $\forall i.$ if $L_i \cap K_A \neq \emptyset$ then ($p_j = \circ$ or $p_j = \uparrow$),
  (6) $\forall i.$ if $p_i = \uparrow$ then $(\mathit{\Gamma}, \mathit{inherit}(S, L_i), U_i) \in R$.
  Similarly,

(7) $\forall i, j.$ if $L_i \cap M_j \neq \emptyset$ then $(U_i, T_j) \in R$ and $(p_i, q_j) \in R$,
(8) $\bigcup_j^{1 \cdots m} M_j \subseteq \bigcup_i^{1 \cdots n} L_i \cup L_A$,
(8') $M_A \subseteq L_A$,
(9) $\forall j.$ if $M_j \cap L_A \neq \emptyset$ then ($q_j = \circ$ or $q_j = \uparrow$),
(10) $\forall j.$ if $q_j = \uparrow$ then $(\Gamma, \mathit{inherit}(U, M_j), T_j) \in R$.

Our goal is to show that $(\Gamma, S, T) \in \mathcal{ST}(TR(R))$, by constructing a proof of S-Object using the above hypotheses and the definition of $\mathcal{ST}$ and TR. Informally, we need to find support for the hypotheses of S-Object for S and T among the elements of $TR(R)$. In particular, we show that:

a. $\bigcup_j^{1 \cdots m} M_j \subseteq \bigcup_h^{1 \cdots l} K_h \cup K_A$.
   Proof: By (4), (4'), (8), (8'), and transitivity of subset inclusion.

b. $\forall h, j.$ if $K_h \cap M_j \neq \emptyset$ then $(o_h, q_j) \in TR(R)$ and $(\Gamma, S, T) \in TR(R)$
   Proof: Let $x \in K_h \cap M_j$, thus $x \in K_h$ and $x \in M_j$. By (8), there are two cases:
   i. $x \in L_A$ — In this case, $M_j \cap L_A \neq \emptyset$. Since $L_A \subseteq K_A$ by (8'), $x \in K_A$. But by the well-formedness of object types, object types' fields are partitioned, and this is a contradiction, since $x \in K_h$. This case cannot occur.
   ii. $x \in L_i$ for some i — In this case, we have that $x \in L_i$ and $x \in M_j$, so by (7), $(U_i, T_j) \in R$ and $(p_i, q_j) \in R$. We also have that $x \in L_i$ and $x \in K_h$, so by (3), $(S_h, U_i) \in R$ and $(o_h, p_i) \in R$. This completes item b., as by definition of TR, $(o_h, p_i) \in R, (p_i, q_j) \in R \Rightarrow (o_h, q_j) \in TR(R)$, and $(S_h, U_i) \in R, (U_i, T_j) \in R \Rightarrow (S_h, T_j) \in TR(R)$.

c. $\forall j. M_j \cap K_A \neq \emptyset \Rightarrow (q_j = \circ$ or $q_j = \uparrow)$
   Proof: For each non-vacuous case of j, there is some x with $x \in M_j$ and $x \in K_A$. By (8), there are two cases:
   i. $x \in M_j \cap L_i$ for some i — In this case, $L_i \cap K_A \neq \emptyset$, so by (5) $p_i = \circ$ or $p_i = \uparrow$. Only $p-$Refl applies, so item c. is complete.
   ii. $x \in M_j \cap L_A$ — This case follows directly from (9).

d. $\forall j.$ if $q_j = \uparrow$ then $(\Gamma, \mathit{inherit}(S, M_j), T_j) \in TR(R)$.
   Proof: By (10), $\forall j.$ if $q_j = \uparrow$ then $(\Gamma, \mathit{inherit}(U, M_j), T_j) \in R$. By assumption, we have that $(\Gamma, S, U) \in R$ (or, equivalently, $\Gamma \vdash S <: U$). Recall from (1) that $S = \{K_1^{o_1} : S_1, \cdots, K_l^{o_l} : S_n, K_A : \mathbf{abs}\}$. By lemma 12, since $\Gamma \vdash S <: U, M_j \subseteq M_j$, it must be that $\Gamma \vdash \mathit{inherit}(S, M_j) <: \mathit{inherit}(U, M_j)$ for each j. Now we have that $(\Gamma, \mathit{inherit}(S, M_j), \mathit{inherit}(U, M_j)) \in R$ and $(\Gamma, \mathit{inherit}(U, M_j), T_j) \in R$ for each j, which is sufficient to show that $(\Gamma, \mathit{inherit}(S, M_j), T_j) \in TR(R)$ for each j, which completes the proof.

- $U = b$ Only S-b applies, so S and T must both be b, and are therefore in R.
- **Case $U = L_u$.** Only case S-Str applies for $S <: U$ and $U <: T$. Thus, $S = L_s$ and $T = L_T$, with $L_s \subseteq L_u \subseteq L_T$. Thus $S <: T$ follows by transitivity of the subset relation.
- **Case $U = \textsf{Ref}\ U'$.** Only case S-Ref applies, thus $S = \textsf{Ref}\ S'$ and $T = \textsf{Ref}\ T'$, with $(S', U'), (U', S'), (U', T'), (T', U') \in R$. By definition of TR, $(S', T'), (T', S') \in TR(R)$. Thus, $(\textsf{Ref}\ S', \textsf{Ref}\ T') \in \mathcal{ST}(TR(R))$.

- **Case $U = \alpha$.** There are three possibilities, depending on uses of S-VR and S-VTR:
  - $S = \alpha$ and $T = \alpha$, so $(\Gamma, S, T) \in TR(R)$ by S-VR.
  - $S = \beta$, $\beta \neq \alpha$, $(\beta <: \alpha) \in \Gamma$, and $(\alpha <: T) \in \Gamma$. In this case, $(\Gamma, S, T) \in TR(R)$ by S-VTR.
  - $S = \alpha$ and $(\alpha <: T) \in \Gamma$. In this case, $(\Gamma, S, T) \in TR(R)$ by S-VTR.
- **Case** $U = \forall\alpha <: U_1.U_2$. The only rule that applies is S-Kern, so it must be that:
  - $S = \forall\alpha <: U_1.S_2$,
  - $((\Gamma, \alpha <: U_1), S_2, U_2) \in \mathcal{ST}(R)$,
  - $T = \forall\alpha <: U_1.T_2$,
  - $((\Gamma, \alpha <: U_1), U_2, T_2) \in \mathcal{ST}(R)$.

  By the definition of TR, $((\Gamma, \alpha <: U_1), S, T) \in TR(R)$.
- $U = \mu\alpha.T$ This case is addressed in [19, chapter 21].
- $U = U_1 \to U_2$ See [19, page 288]

■

Lemma 8 (Subtyping is Reflexive) *For all $T \in \mathcal{T}$, $(T, T) \in \nu\mathcal{ST}$, and for all $F \in \mathcal{F}$, $(F, F) \in \nu\mathcal{ST}$.*

Proof: By case analysis on the subtyping rules.

Lemma 9 *$\mathcal{ST}$ is Invertible*

Proof: The corresponding support function is well-defined. By inspection of the subtyping rules, for a given pair of expressions, only one typing rule applies.

Theorem 3 *For all types S and T, $S <: T$ is decidable.*

Proof: Since S and T are finite μ-types, the set $\mathit{reachables}_{\mathcal{ST}}(S, T)$ is finite [19, Proposition 21.9.11]. Thus, the algorithm $\mathit{gfp}_{\mathcal{ST}}$ [19, Definition 21.5.5] terminates [19, Theorem 21.5.12]. ■

Lemma 10 *For all Γ, T, L, M, $\mathit{inherit}_\Gamma(T, L) \sqcup \mathit{inherit}_\Gamma(T, M) = \mathit{inherit}_\Gamma(T, L \sqcup M)$.*

Proof: By induction on the syntactic size of T and by definition of the join operator.

Note that in the definition of *inherit*, the condition in both cases requires that for some L_C, $L_Q \sqsubseteq L_C$. $L \sqsubseteq L_C$ and $M \sqsubseteq L_C$ hold if the left-hand side of the equality are defined. However, $L \sqcup M \subseteq L_C$ holds only because $L \sqcup M = L \cup M$.

■

Lemma 11 *If $L_Q \subseteq M_Q \subseteq \bigcup_j^{1\cdots m} M_j$ and $\forall i, j. L_i \cap M_j \neq \emptyset \Rightarrow \Gamma \vdash S'_i <: T'_j$ then*

$$\Gamma \vdash \bigsqcup_{i}^{1\cdots n} \{S'_i \mid \Gamma \vdash L_Q \cap L_i \neq \emptyset\} <: \bigsqcup_{j}^{1\cdots m} \{T'_i \mid \Gamma \vdash M_Q \cap M_j \neq \emptyset\}$$

Proof: It is sufficient to show that for all S'_i on the left-hand side, there exists a T'_j such that $\mathit{\Gamma} \vdash S'_i <: T'_j$.

For any S'_i, since $L_i \cap L_Q \neq \emptyset$, $\exists \mathit{str}.\mathit{str} \in L_i \cap L_Q$. Since $L_Q \subseteq \bigcup M_j$, intersecting on the left-hand side we have $L_i \cap L_Q \subseteq \bigcup M_j$. Thus $\mathit{str} \in \bigcup \{M_j \mid q_j \neq^{\circ}\}$. Therefore, $\exists M_j.\mathit{str} \in M_j$, hence $L_i \cap M_j \neq \emptyset$ and so $\mathit{\Gamma} \vdash S'_i <: T'_j$. ■

Lemma 12 *For all $\mathit{\Gamma}, S, T, L_Q, M_Q$, if:*

H1. $\mathit{\Gamma} \vdash S <: T$,
H2. $\mathit{\Gamma} \vdash L_Q \subseteq M_Q$,
H3. $\mathit{inherit}_{\mathit{\Gamma}}(S, L_Q) = S'$, *and*
H4. $\mathit{inherit}_{\mathit{\Gamma}}(T, M_Q) = T'$,

then $\mathit{\Gamma} \vdash S' <: T'$.

Proof: By double induction on syntactic size of S' and T', followed by case analysis of *inherit* in H3 and H4. We thus have four cases. In all cases, by inversion of (H1) and the definition of *inherit*, we have:

$$S = \{L_1^{p_1} : S'_1 \cdots L_n^{p_n} : S'_n, L_A : \textbf{abs}\}$$
$$T = \{M_1^{q_1} : T'_1 \cdots M_m^{q_m} : T'_m, M_A : \textbf{abs}\}$$

We thus have available the hypotheses of S-Object:

I1. $\forall i, j. L_i \cap M_j \neq \emptyset \Rightarrow p_i <: q_j \wedge \mathit{\Gamma} \vdash S'_i <: T'_j$,
I2. $\bigcup_i^{1 \cdots m} M_i \subseteq \bigcup_j^{1 \cdots n} L_j \cup L_A$,
I3. $M_A \subseteq L_A$,
I4. $\forall j$.if $q_j = \uparrow$ then $q_j = \uparrow \wedge \mathit{\Gamma} \vdash \mathit{inherit}(S, M_j) <: T'_j$, and
I5. $\forall j$.if $M_j \cap L_A \neq \emptyset$ then $q_j =^{\circ}$ or $q_j = \uparrow$

Case 1. Base case, where `"parent"` of both S and T are elided.

By definition of *inherit*, the goal is:

$$\mathit{\Gamma} \vdash \bigsqcup_{i}^{1 \cdots n} \{S'_i \mid \mathit{\Gamma} \vdash L_Q \cap L_i \neq \emptyset\} <: \bigsqcup_{j}^{1 \cdots m} \{T'_i \mid \mathit{\Gamma} \vdash M_Q \cap M_j \neq \emptyset\}$$

The condition on both applications of *inherit* are $L_Q \subseteq \bigcup \{L_i \mid p_i \neq \circ\}$ and $M_Q \subseteq \bigcup \{M_j \mid q_j \neq \circ\}$. Therefore, $M_Q \subseteq \bigcup M_j$ and lemma 11 applies.

Case 2. Inductive case, where `"parent"` of both S and T are references to objects.

H5. $\exists L_i.$`"parent"` $\in L_i \wedge S'_i = \mathsf{Ref}\ S_P$,
H5'. $L_Q \subseteq \bigcup_i^{1 \cdots n} L_i \cup L_A$,
H6. $\exists M_i.$`"parent"` $\in M_i \wedge T'_i = \mathsf{Ref}\ T_P$, and
H6'. $M_Q \subseteq \bigcup_i^{1 \cdots m} M_i \cup M_A$.
HInd. $\forall L'_Q, M'_Q, S_P, T_P. |S_P| < |S| \wedge |T_P| < |T| \wedge \mathit{\Gamma} \vdash L'_Q \subseteq M'_Q \wedge \mathit{\Gamma} \vdash S_P <: T_P \Rightarrow \big(\mathit{inherit}_{\mathit{\Gamma}}(S_P, L'_Q) = S'_P \wedge \mathit{inherit}_{\mathit{\Gamma}}(T_P, M'_Q) = T'_P \Rightarrow \mathit{\Gamma} \vdash S'_P <: T'_P\big)$.

The goal thus reduces to:

$$\mathit{\Gamma} \vdash \bigsqcup\{S'_i \mid \mathit{\Gamma} \vdash L_Q \cap L_i \neq \emptyset\} \sqcup \mathit{inherit}(S_p, \mathcal{L}) <: \bigsqcup\{T'_i \mid \mathit{\Gamma} \vdash M_Q \cap M_i \neq \emptyset\} \sqcup \mathit{inherit}(T_p, \mathcal{M})$$
$$\text{where } \mathcal{L} = L_Q \cap (L_A \cup \bigcup\{L_i | p_i = \circ\}) \text{ and } \mathcal{M} = M_Q \cap (M_A \cup \bigcup\{M_j | q_j = \circ\})$$

We define the set of inherited fields of T that are looked up by M_Q and are absent on S:

$$\mathcal{N} = \{M_j \mid M_Q \cap M_j \cap L_A \neq \emptyset \wedge q_j = \uparrow\}$$
$$\mathcal{L}^+ = \mathcal{L} \cap \bigcup \mathcal{N}$$
$$\mathcal{L}^- = \mathcal{L} \cap \overline{\bigcup \mathcal{N}}$$

Using lemma 10, we rewrite the goal to:

$$\mathit{\Gamma} \vdash \bigsqcup\{S'_i \mid \mathit{\Gamma} \vdash L_Q \cap L_i \neq \emptyset\} \sqcup \mathit{inherit}(S_p, \mathcal{L} \cap \overline{\bigcup \mathcal{N}}) \sqcup \mathit{inherit}(S_p, \mathcal{L} \cap \bigcup \mathcal{N})$$
$$<: \bigsqcup\{T'_i \mid \mathit{\Gamma} \vdash M_Q \cap M_i \neq \emptyset\} \sqcup \mathit{inherit}(T_p, \mathcal{M})$$

We prove the goal by breaking it into the following subcases:

a. $\mathit{\Gamma} \vdash \bigsqcup\{S'_i \mid \mathit{\Gamma} \vdash L_Q \cap L_i \neq \emptyset\} <: \bigsqcup\{T'_j \mid \mathit{\Gamma} \vdash M_Q \cap M_j \neq \emptyset\}$
   We cannot apply lemma 11 directly because $M_Q \subseteq \bigcup M_j \cup M_A$, whereas the hypothesis of the lemma requires $M_Q \subseteq \bigcup M_j$.
   However, note that since $L_A \cap L_i = \emptyset$ (by well-formedness of types), we have $L_Q \cap L_i \neq \emptyset$ iff $L_Q \backslash L_A \cap L_i$. Similarly, $M_Q \cap M_j \neq \emptyset$ iff $M_Q \backslash M_A \cap M_J$. We can therefore rewrite the subgoal to $\mathit{\Gamma} \vdash \bigsqcup\{S'_i \mid \mathit{\Gamma} \vdash L_Q \backslash L_A \cap L_i \neq \emptyset\} <: \bigsqcup\{T'_j \mid \mathit{\Gamma} \vdash M_Q \backslash M_A \cap M_j \neq \emptyset\}$. Lemma 11 now applies.

b. $\mathit{\Gamma} \vdash \mathit{inherit}(S_p, \mathcal{L} \cap \overline{\bigcup \mathcal{N}}) <: \mathit{inherit}(T_p, \mathcal{M})$
   By induction (HInd). The following cases allow us to apply HInd.
   - $|S_P| < |S|$ and $|T_P| < |T|$ are trivial.
   - We show that $\mathit{\Gamma} \vdash S_P <: T_P$. By H5 and H6, `"parent"` $\in L_P, M_P$ thus $L_P \cap M_P \neq \emptyset$. Therefore, $\mathit{\Gamma} \vdash \mathsf{Ref}\ S_P <: \mathsf{Ref}\ T_P$ by I1. By (S-Ref), it follows that $\mathit{\Gamma} \vdash S_P <: T_P$.
   - We show that $\mathit{\Gamma} \vdash \mathcal{L} \cap \overline{\bigcup \mathcal{N}} \subseteq \mathcal{M}$. It is sufficient to show that $\forall x. \mathit{\Gamma} \vdash x \in \mathcal{L} \cap \overline{\bigcup \mathcal{N}} \Rightarrow \mathit{\Gamma} \vdash x \in \mathcal{M}$.
     By definition of $\mathcal{L}$, $x \in L_Q$, thus by (H2), $x \in M_Q$.
     By (H6'), $x \in M_A \cup \bigcup_j^{j \cdots m} M_j$. If $x \in M_A$, we are done. So, consider the case where $\exists j. x \in M_j$.
     By definition of $\mathcal{L}$, either $x \in L_A$ or $x \in \bigcup\{L_i | p_i = \circ\}$. If $x \in L_A$, then by I5 $q_j = \circ$. If $x \in \bigcup\{L_i | p_i = \circ\}$, since $x \in L_i \cap M_j$, $p_i <: q_j$, and by p-Refl, $q_j = \circ$.
     Therefore, $x \in \mathcal{M}$ since it is in both sets.

c. $\mathit{\Gamma} \vdash \mathit{inherit}(S_p, \mathcal{L} \cap \bigcup \mathcal{N}) <: \bigsqcup\{T'_j \mid \mathit{\Gamma} \vdash M_Q \cap M_j \neq \emptyset\}$.
   Rewrite the left-hand side by expanding the definition of $\mathcal{N}$, distributing the intersection over the union, and applying lemma 10:

$$\mathit{\Gamma} \vdash \bigsqcup\{\mathit{inherit}(S_p, \mathcal{L} \cap M_j) \mid M_Q \cap M_j \cap L_A \neq \emptyset \wedge q_j = \uparrow\} <: \bigsqcup\{T'_j \mid \mathit{\Gamma} \vdash M_Q \cap M_j \neq \emptyset\}$$

It is sufficient to show that for all elements of the left-hand side, there exists a supertype on the right-hand side. For each $\mathit{inherit}(S_p, \mathcal{L} \cap M_j)$ on the left-hand side, the associated T'_j is on the right-hand side by definition of $\mathcal{N}$. We now show that

$$\mathit{\Gamma} \vdash \mathit{inherit}(S_p, \mathcal{L} \cap M_j) <: T'_j$$

Since $M_j \cap L_A \neq \emptyset$ and $q_j =^{\uparrow}$, I4 applies and $\mathit{\Gamma} \vdash \mathit{inherit}(S, M_j) <: T'_j$. By lemma 10 $\mathit{inherit}(S, M_j) = \mathit{inherit}(S, \mathcal{L} \cap M_j) \sqcup \mathit{inherit}(S, \overline{\mathcal{L}} \cap M_j)$, so $\mathit{\Gamma} \vdash \mathit{inherit}(S, \mathcal{L} \cap M_j) <: T_j$ by the definition of joins. Further, by the definition of *inherit*,

$$\mathit{\Gamma} \vdash \mathit{inherit}(S, \mathcal{L} \cap M_j) = \mathit{inherit}(S_P, \mathcal{L} \cap M_j \cap L_A \bigcup \{L_i \mid p_i = \circ\}) \sqcup \dots$$

and by the definition of $\mathcal{L}$, $\mathcal{L} \cap M_j \cap L_A \bigcup \{L_i \mid p_i = \circ\}$ is the same as $\mathcal{L} \cap M_j$. By the definition of joins:

$$\mathit{\Gamma} \vdash \mathit{inherit}_{\mathit{\Gamma}}(S_P, \mathcal{L} \cap M_j) <: T_j$$

which, when applied for each j, completes Case 2.

Case 3. Impossible case, where the `"parent"` field is on the right-hand side type, but is elided on the left-hand side type.

By well-formedness of types, if $\exists i.\texttt{"parent"} \in M_i$ then $q_i =^{\downarrow}$. But by I2, $\texttt{"parent"} \in \bigcup_j^{1\cdots n} L_j \cup L_A$, which is a contradiction.

Case 4. Inductive case, where the `"parent"` field is on the left-hand side type, but is elided on the right-hand side type.

HLP. $\exists L_P.\texttt{"parent"} \in L_P \wedge S'_i = \mathsf{Ref}\ S_P$,
H6. $L_Q \subseteq \bigcup_i^{1\cdots n} L_i \cup L_A$,
HRP. If $\neg\exists M_P.\texttt{"parent"} \in M_P$, and
H8. $M_Q \subseteq \bigcup_i^{1\cdots m} \{M_i \mid q_i \neq^{\circ}\}$.

By HRP, since $\neg\exists M_P.\texttt{"parent"} \in M_P$. The goal is therefore:

$$\mathit{\Gamma} \vdash \bigsqcup_i^{1\cdots n} \{S'_i \mid \mathit{\Gamma} \vdash L_Q \cap L_i \neq \emptyset\} \sqcup \mathit{inherit}(S_P, L_Q \cap (L_A \cup \bigcup \{L_i | p_i = \circ\})) <: \bigsqcup_j^{1\cdots m} \{T'_j \mid \mathit{\Gamma} \vdash M_Q \cap M_j \neq \emptyset\}$$

The first part of the join is satisfied by lemma 11. For part 2, using lemma 10 rewrite:

$$\mathit{\Gamma} \vdash \mathit{inherit}(S_P, L_Q \cap (L_A \cup \bigcup \{L_i | p_i = \circ\})) <: \bigsqcup_j^{1\cdots m} \{T'_j \mid \mathit{\Gamma} \vdash M_Q \cap M_j \neq \emptyset\}$$

to:

$$\mathit{\Gamma} \vdash \mathit{inherit}(S_P, L_Q \cap L_A) \sqcup \mathit{inherit}(S_P, L_Q \cap \bigcup \{L_i | p_i = \circ\}) <: \bigsqcup_j^{1\cdots m} \{T'_j \mid \mathit{\Gamma} \vdash M_Q \cap M_j \neq \emptyset\}$$

There are two cases.

- Consider $str \in L_Q \cap L_A$. This case follows by the same argument as in item c. of Case 2.
- Consider $str \in L_Q \cap L_i$, where $p_i = \circ$. By H2, $str \in M_Q$. By H8, there exists an M_j, such that $str \in M_j$ and $q_j \neq^\circ$. By I1, since $str \in L_i, M_j$, it must be that $p_i <: q_j$. By inspection of the definition of the $p <: q$ relation, we have a contradiction.

■

Lemma 13 *If* $\{L_1 : F_1 \cdots L_n : F_n\} <: \{M_1 : G_1 \cdots M_m : G_m\}$, *then* $\bigcup_j^{1 \cdots m} \{M_j \mid G_j = T^\downarrow\} \subseteq \bigcup_i^{1 \cdots n} \{L_i \mid F_i = T^\downarrow\}$.

Proof: By contradiction.

Assume there exists some string x with $x \in \bigcup_j^{1 \cdots j} \{M_j \mid G_j = T^\downarrow\}$ and $x \notin \bigcup_i^{1 \cdots n} \{L_i \mid F_i = T^\downarrow\}$. By assumption of S-Object, $\bigcup_j^{1 \cdots m} M_j \subseteq \bigcup_i^{1 \cdots n} L_i$, so it must be that there is some L_i that contains x, but either has type T° or **abs**. That is, there must be an M_j with $x \in M_j$ and an L_i with $x \in L_i$ with $G_j = T^\downarrow$ and either $F_i =$ **abs** or $F_i = T'^\circ$. This violates S-Object, which asserts that since $L_i \cap M_j \neq \emptyset$, it must be that $F_i <: G_j$, which cannot happen since possibly absent and definitely absent fields cannot subtype definitely present fields.

■

E Typing

$$\text{T-IfHasField1}\ \frac{\Gamma \vdash v : S \cdots \qquad \Gamma(f) = L \qquad \Sigma; \Gamma, \alpha <: L, f : \alpha \vdash e_2 : T \qquad L' = L \cap \overline{\alpha} \qquad \Gamma \vdash e_3 : T}{\Sigma; \Gamma \vdash \textsf{\textbf{if}}\ (\{str : v \cdots\ \}\ \textsf{\textbf{hasfield}}\ f)\ e_2\ \textsf{\textbf{else}}\ e_3 : T}$$

$$\text{T-IfHasField2}\ \frac{\begin{array}{c}\Gamma(o) = \{\cdots L^\circ : S \cdots\} \qquad \Sigma; \Gamma, o : \{\cdots str^\downarrow : S, L'^\circ : S \cdots\} \vdash e_2 : T \\ L' = L \cap \overline{\{str\}} \qquad \Gamma \vdash e_3 : T\end{array}}{\Sigma; \Gamma \vdash \textsf{\textbf{if}}\ (o\ \textsf{\textbf{hasfield}}\ str)\ e_2\ \textsf{\textbf{else}}\ e_3 : T}$$

$$\text{T-IfHasFieldFalse}\ \frac{\Gamma \vdash v : S \cdots \qquad \Sigma; \Gamma \vdash e_3 : T \qquad str_2 \notin str \cdots}{\Sigma; \Gamma \vdash \textsf{\textbf{if}}\ (\{str : v \cdots\ \}\ \textsf{\textbf{hasfield}}\ str_2)\ e_2\ \textsf{\textbf{else}}\ e_3 : T}$$

$$\text{T-IfFalse}\ \frac{\Sigma; \Gamma \vdash e_3 : T}{\Sigma; \Gamma \vdash \textsf{\textbf{if}}\ (\textsf{\textbf{false}})\ e_2\ \textsf{\textbf{else}}\ e_3 : T}$$

Fig. 6. Auxiliary Typing Rules for If-Splitting

Lemma 14 (Canonical Forms) *If* $\Sigma; \Gamma \vdash v : T$ *and if* T *is:*

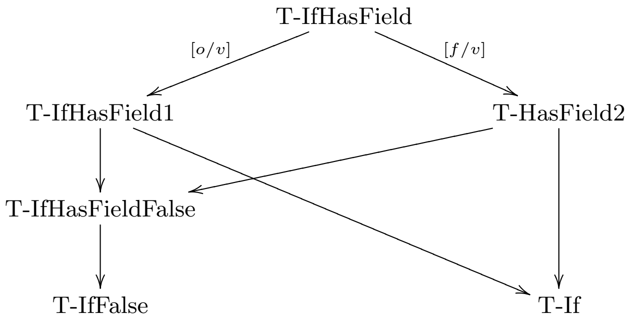

Fig. 7. Usage of Auxiliary Typing Rules by Substitution

- $\{L_1^{p_1} : S_1 \cdots L_m^{p_m} : S_m\}$ *then* $v =$ *{str*$_1$*: v*$_1\cdots$*}*, $\Sigma;\Gamma \vdash v_1 \cdots v_n : U_1 \cdots U_n$, $\Sigma;\Gamma \vdash v : \{str_1^{\downarrow} : U_1 \cdots str_n^{\downarrow} : U_n, \overline{\{str_1 \cdots str_n\}} : \mathbf{abs}\}$ *and* $\{str_1^{\downarrow} : U_1 \cdots str_n^{\downarrow} : U_n, \overline{\{str_1 \cdots str_n\}} : \mathbf{abs}\} <: \{L_1^{p_m} : S_1 \cdots L_m^{p_m} : S_m\}$,
- *Ref* S, *then* $v = loc$ *and* $\Sigma(loc) <: S$,
- $S \to T$, *then* $v =$ **func** *(x) { e }*,
- L *then* $v = str$ *and* $\Gamma \vdash str <: L$

Proof: By induction on the typing derivation.

Lemma 15 (Inversion) *If:*

- $\Sigma;\Gamma \vdash$ *{str :v*$\cdots$*}* $: T$ *then* $\Sigma;\Gamma \vdash v : S \cdots$ *and* $\Gamma \vdash \{str^{\downarrow} : S \cdots\} <: T$
- $\Sigma;\Gamma \vdash e_1$*+*$e_2 : T$ *then* $\Sigma;\Gamma \vdash e_1, e_2 <:$ *Str and* $\Gamma \vdash T <:$ *Str.*
- $\Sigma;\Gamma \vdash$ **fix** *(f :S).e* $: T$ *then* $\Sigma;\Gamma, f : S \vdash e <: S$ *and* $\Gamma \vdash S <: T$.
- $\Sigma;\Gamma \vdash$ **ref** $e : T$ *then* $\Sigma;\Gamma \vdash e : S$ *and Ref* $S = T$
- $\Sigma;\Gamma \vdash l : T$ *then* $\Sigma(l) = T$
- $\Sigma;\Gamma \vdash$ **deref** $e : T$, *then* $\Sigma;\Gamma \vdash e :$ *Ref* S *with* $S <: T$,
- $\Sigma;\Gamma \vdash e_1$ = $e_2 : T$, *then* $\Sigma;\Gamma \vdash e_1 :$ *Ref* S, $\Sigma;\Gamma \vdash e_2 : U$, $U <: S$, *and Ref* $S <: T$,
- $\Sigma;\Gamma \vdash e_f$*(e*$\cdots$*)* $: T$, *then* $\Sigma;\Gamma \vdash e_f : S \cdots \to T'$, $\Sigma;\Gamma \vdash e : S \cdots$, *and* $T' <: T$.
- $\Sigma;\Gamma \vdash e_o$*[*e_f*]* $: T$, *then* $\Sigma;\Gamma \vdash e_o : \{L_1^{p_1} : S_1 \cdots\}$, $\Sigma;\Gamma \vdash e_f : L$, *inherit*$(\{L_1^{p_1} : S_1 \cdots\}, L) = T'$, *and* $T' <: T$.
- $\Sigma;\Gamma \vdash e_o$*[*e_f = e_v*]* $: T$ *then* $\Sigma;\Gamma \vdash e_o : \{L^p : S \cdots, L_A : \mathbf{abs}\}$, $\Sigma;\Gamma \vdash e_f : L'$, $\Sigma;\Gamma \vdash e_v : U$, $\forall L.$*if* $L \cap L' \neq \emptyset$ *then* $U <: S$, *and* $\Gamma \vdash \{L^p : S \cdots, L_A : \mathbf{abs}\} <: T$.
- $\Sigma;\Gamma \vdash$ **delete** e_o*[*e_f*]* $: T$, *then* $\Sigma;\Gamma \vdash e_o : \{L_1^{p_1} : S_1 \cdots\}$, $\Sigma;\Gamma \vdash e_f : L$, $\forall L \cap L_i \neq \emptyset . F_i \neq T^{\downarrow}$, $\Gamma \vdash \{L_1^{p_1} : S_1 \cdots\} <: T$
- $\Sigma;\Gamma \vdash e_1$ **hasfield** $e_2 :$ *Bool*, *then* $\Sigma;\Gamma \vdash e_1 : \{L_1 : F_1 \cdots L_n : F_n\}$, $\Sigma;\Gamma \vdash e_2 : L$.
- $\Sigma;\Gamma \vdash e$ **matches** $P :$ *Bool*, *then* $\Sigma;\Gamma \vdash e : L$.
- $\Sigma;\Gamma \vdash$ **if** *(*v_1*) {* e_2 *}* **else** *{* e_3 *}* $: T$, *then* $\Sigma;\Gamma \vdash v_1 :$ *Bool*, $\Sigma;\Gamma \vdash e_2 : T$, *and* $\Sigma;\Gamma \vdash e_3 : T$.

– $\Sigma; \Gamma \vdash$ **fieldin** v_{obj} **init** v_{acc} **do** $v_f : T$, *then* $\Sigma; \Gamma \vdash v_{obj} : \{L^p : S \cdots\}$, $\Sigma; \Gamma \vdash v_{acc} : T$, *and* $\Sigma; \Gamma \vdash v_f : (\mathsf{Str} \to T) \to T$

Lemma 16 (Type Substitution) *If* $\Sigma; \alpha <: S, \Gamma \vdash e : T$ *and* $\Gamma \vdash U <: S$ *then* $\Sigma; \Gamma[\alpha/U] \vdash e[\alpha/U] : T[\alpha/U]$.

Proof: By induction on the typing derivation. ■

Lemma 17 (Substitution) *If* $\Sigma; x : S, \Gamma \vdash e : T$ *and* $\Sigma; \Gamma \vdash v : S$, *then* $\Sigma; \Gamma \vdash e[x/v] : T$.

Proof: By induction on the typing derivation. The only interesting case is substituting the identifiers in **if** `(o` **hasfield** `f)` e_2 **else** e_3 when it is typed by T-IfHasField. The resulting expressions require the auxiliary typing rules in figure 6.

- The expression is typed by T-HasField and $x = o$. The resulting expression is typable by T-IfHasField1 as follows. By canonical forms, $v = \{ str : v \cdots \}$ and $\Sigma; \Gamma \vdash v' : T'$. By induction, $\Sigma; \Gamma, \alpha <: L, f : \alpha e_2[x/v]$ and $\Sigma; \Gamma \vdash e_3[x/v]$. The remaining antecents of T-IfHasField1 are those of T-HasField.
- The expression is typed by T-HasField and $x = f$. The resulting expression is typable by T-IfHasField2 as follows. By canonical forms, $v = str$, $\Sigma; \Gamma \vdash v : str$, and $\Gamma \vdash str <: L$. By type substitution followed by induction, $\Sigma; \Gamma, o : \{\cdots str^{\downarrow} : S, L'^{\circ} : S \cdots\} \vdash e_2 : T$. The remaining antecedents of T-IfHasField2 are those of T-HasField.
- The expression is typed by T-IfHasField1 and $x = f$. The resulting expression has the form:

  if `({` str`:` $v' \cdots$ `}` **hasfield** str'`)` e_2 **else** e_3

  There are two cases.
  - If $str' \in str \cdots$ then by type substitution and induction, $\Sigma; \Gamma, \alpha <: L, f : \alpha \vdash e_2 : T[\alpha/str][f/v] = \Sigma; \Gamma \vdash e_2[f/v] : T$. By induction, $\Sigma; \Gamma \vdash e_3[f/v] : T$. Finally, the conditional has type Bool. Thus the expression is typable by T-If.
  - If $str' \notin str \cdots$ then the term is trivally typable by T-IfHasFieldFalse.
- The expression is typed by T-IfHasField2 and $x = o$. The resulting expression has the form:

  if `({` str`:` $v' \cdots$ `}` **hasfield** str'`)` e_2 **else** e_3

  There are two cases.
  - If $str' \in str \cdots$ then the result is trivially typable by T-If.
  - If $str' \notin str \cdots$ then the term is trivally typable by T-IfHasFieldFalse.

■

Lemma 18 (Main Preservation) *If* $\Sigma_1; \cdot \vdash ae : T$, $\Sigma_1 \vdash \sigma_1$, *and* $\sigma_1 E\langle ae \rangle \to \sigma_2 E\langle e_2 \rangle$ *then there exists a* Σ_2, *such that:*

i. $\Sigma_2 \supseteq \Sigma_1$,
ii. $\Sigma_2 \vdash \sigma_2$, *and*
iii. $\Sigma_2; \cdot \vdash e_2 : T$.

Proof: By case-analysis on ae, using inversion (lemma 15) where specified:

- $\sigma_1 E\langle \texttt{(}\mathsf{\textbf{func}}\ \texttt{(}x : S'\texttt{)}\ \texttt{\{}\ e\ \texttt{\})(}v\texttt{)}\rangle \to \sigma_1 E\langle e[x/v]\rangle$.
  By inversion, $\Sigma_1; \cdot \vdash v : S$, $\cdot \vdash S <: S'$, $\Sigma_1; x : S \vdash e : T'$, and $\cdot \vdash T' <: T$. By substitution (lemma 17), $\Sigma_1; \cdot \vdash e[x/v] : T$.
- $\sigma_1 E\langle \mathsf{\textbf{fix}}\ \texttt{(}x : S\texttt{.}\ e\rangle \to \sigma_1 E\langle e[x/\mathsf{\textbf{fix}}\ \texttt{(}x : S\texttt{).}e]\rangle$.
  By inversion, $\Sigma_1; \cdot \vdash e : S$, $\Sigma_1; x : S \vdash e : S$, and $\cdot \vdash S <: T$. By substitution (lemma 17), $\Sigma_1; \cdot \vdash e[x/\mathsf{\textbf{fix}}\ \texttt{(}x{:}S\texttt{).}\ \mathtt{e}] : T$.
- $\sigma E\langle (\lambda\alpha <: S.e)(U)\rangle \to \sigma E\langle e[\alpha/U]\rangle$
  By type substitution (lemma 16).
- $\sigma E\langle \mathsf{\textbf{ref}}\ v\rangle \to \sigma, (l, v)E\langle l\rangle$ where $l \notin dom(\sigma)$. By inversion, $\Sigma_1; \cdot \vdash v : S$ and $\mathsf{Ref}\ S <: T$. Let $\Sigma_2 = l : S, \Sigma_1$. By T-Loc, $\Sigma_2; \cdot \vdash l : \mathsf{Ref}\ S$.
- $\sigma E\langle \mathsf{\textbf{deref}}\ \mathsf{l}\rangle \to \sigma E\langle \sigma(l)\rangle$
  By inversion, $\Sigma; \Gamma \vdash l : \mathsf{Ref}\ S$ and $\Gamma \vdash S <: T$. By inversion, $\Sigma(l) = S$. Thus by T-Loc and T-Sub $\Sigma; \Gamma \vdash l : T$.
- $\sigma E\langle \mathsf{setref}\ l\ \ v\rangle \to \sigma[l := v]E\langle l\rangle$ where $l \in dom(\sigma)$.
  By inversion (T-SetRef), $\Sigma; \Gamma \vdash l : \mathsf{Ref}\ \ S$, $\Sigma; \Gamma \vdash v : S$, and $\mathsf{Ref}\ \ S = T$. By inversion (T-Loc), $\Sigma(l) = T$ thus $\Sigma \vdash \sigma[l := v]$. by T-Loc, $\Sigma; \Gamma \vdash l : T$.
- $\sigma E\langle \texttt{\{}\ \cdots str\texttt{:}\ v \cdots\ \texttt{\}[}str\texttt{]}\rangle \to \sigma E\langle v\rangle$. By inversion, $\Sigma_1; \cdot \vdash str : L$, $\Sigma_1; \cdot \vdash \{\cdots str_1 : v_1 \cdots\} : S$, $\Sigma_1; \cdot \vdash v : T', T' <: inherit.(S, L)$, and $inherit.(S, L) <: T$. By inversion, $\{\cdots str^{\downarrow} : T' \cdots\} <: S$. Thus $\cdot \vdash T' <: T$.
  By lemma 12, $inherit.(\{\cdots str^{\downarrow} : T' \cdots\}, str) <: inherit.(S, L)$. By [REF], $inherit.(\{\cdots str^{\downarrow} : T' \cdots\}, str) = T'$.
- $\sigma E\langle \texttt{\{}\ \cdots\ \texttt{"parent":}\ l\ \texttt{\}[}str\texttt{]}\rangle \to \sigma E\langle \texttt{(}\mathsf{\textbf{deref}}\ l\texttt{)[}str\texttt{]}\rangle$, where $str \notin \cdots$.
  By inversion of the left-hand side, $\Sigma_1; \cdot \vdash str : L$, $\Sigma_1; \cdot \vdash \{\cdots \texttt{"parent"} : T_P\} : S$, and $inherit.(S, L) <: T$. By lemma 12, $inherit.(\{\cdots \texttt{"parent"}^{\downarrow} : T_P\}, str) <: inherit.(S, L)$. By inversion, $\Sigma_1; \cdot \vdash l : \mathsf{Ref}\ S_P = T_P$. Since $str \notin \cdots$ and by definition of $inherit$, $inherit.(\{\cdots \texttt{"parent"}^{\downarrow} : \mathsf{Ref}\ S_P\}, str) = inherit_{\ldots}(S_P, str)$, which is a subtype of T.
  Type right-hand side with T-Sub and T-GetField, using $\Sigma_1; \cdot \vdash str : str$, $inherit_{\ldots}(S_P, str) <: T$, and $\Sigma_1; \cdot \vdash \texttt{(}\mathsf{\textbf{deref}}\ l\texttt{)}: S_P$. This holds since $\Sigma_1 : \cdot \vdash l : \mathsf{Ref}\ S_P$ above.
- $\sigma E\langle \texttt{\{}\ \cdots str : v \cdots\ \texttt{\}[}str\ \texttt{=}\ v'\texttt{]}\rangle \to \sigma E\langle \texttt{\{}\ \cdots str : v' \cdots\ \texttt{\}}\rangle$
  By inversion (T-Update), $\Sigma; \Gamma \vdash \texttt{\{}\cdots\texttt{\}} : \{L : S \cdots\}$, $\Gamma \vdash \{L : S \cdots\} <: T$, and $\Sigma; \Gamma \vdash v' : U'$. By inversion (T-Object), $\Sigma; \Gamma \vdash \texttt{\{}\cdots str{:}v \cdots\texttt{\}} : \{\cdots str : U \cdots\}$ and $\Gamma \vdash \{\cdots str : U \cdots\} <: \{L : S \cdots\}$. Thus by inversion (T-Update), $\Gamma \vdash U' <: U$. The resulting expression is typable by S-Object, thus by S-Sub, $\Gamma \vdash \{\cdots str : U' \cdots\} <: \{\cdots str : U \cdots\} <: T$.
- $\sigma E\langle \texttt{\{}\ \cdots\ \texttt{\}[}str\ \texttt{=}\ v'\texttt{]}\rangle \to \sigma E\langle \texttt{\{}\ \cdots\ \texttt{\}}\rangle$ where $str \notin \cdots$
  By inversion of T-Update, $\Sigma; \Gamma \vdash \texttt{\{}\cdots str{:}v \cdots\texttt{\}} : S$ and $\Gamma \vdash S <: T$.
- $\sigma E\langle \mathsf{\textbf{delete}}\ \texttt{\{}\ \cdots str : v \cdots\ \texttt{\}[}str\texttt{]}\rangle \to \sigma E\langle \texttt{\{}\ \cdots\cdots\ \texttt{\}}\rangle$
  Similar to to E-UpdateField case above.
- $\sigma E\langle \mathsf{\textbf{delete}}\ \texttt{\{}\ \cdots\ \texttt{\}[}str\texttt{]}\rangle \to \sigma E\langle \mathsf{\textbf{delete}}\ \texttt{\{}\ \cdots\ \texttt{\}}\rangle$ where $str \notin \cdots$
  Similar to to E-UpdateField case above.
- $\sigma E\langle \mathsf{\textbf{fieldin}}\ \texttt{\{}\ s{:}v\texttt{,}rest \cdots\ \texttt{\}}\ \mathsf{\textbf{init}}\ v_{acc}\ \mathsf{\textbf{do}}\ v_f\rangle \to \sigma E\langle \mathsf{\textbf{fieldin}}\ \texttt{\{}\ str_2 : v_2\ \ \cdots\ \texttt{\}}\ \mathsf{\textbf{init}}\ v_f\texttt{(}str_1\texttt{)(}v_{acc}\texttt{)}\ \mathsf{\textbf{do}}\ v_f\rangle$
  By inversion, $\Sigma; \Gamma \vdash v_{acc} : T$ and $\Sigma; \Gamma \vdash v_f : (\mathsf{Str} \to T) \to T$. The double application can by typed by T-App, and the resulting expression will by typable by T-FieldIn.

- $\sigma E\langle \textbf{fieldin}\ \texttt{\{}\ s\texttt{:}v\texttt{\}}\ \textbf{init}\ v_{acc}\ \textbf{do}\ v_f \rangle \to \sigma E\langle v_f\texttt{(}s\texttt{)(}v_{acc}\texttt{)}\rangle$
  Similar to E-FieldIn above.
  The remaining cases are conventional and straightforward. (**if** is standard, and in the rest, both the left-hand side and the right-hand side have type Bool.)
- $\sigma E\langle \textbf{if}\texttt{(}\textbf{true}\texttt{)}\ \texttt{\{}\ e_1\ \texttt{\}}\ \textbf{else}\ \texttt{\{}\ e_2\ \texttt{\}}\rangle \to \sigma E\langle e_1\rangle$
- $\sigma E\langle \textbf{if}\texttt{(}\textbf{false}\texttt{)}\ \texttt{\{}\ e_1\ \texttt{\}}\ \textbf{else}\ \texttt{\{}\ e_2\ \texttt{\}}\rangle \to \sigma E\langle e_2\rangle$
- $\sigma E\langle \texttt{\{}\ \cdots str\texttt{:}v\cdots\ \texttt{\}}\ \textbf{hasfield}\ str\rangle \to \sigma E\langle \textbf{true}\rangle$
- $\sigma E\langle \texttt{\{}\ \cdots\ \texttt{\}}\ \textbf{hasfield}\ str\rangle \to \sigma E\langle \textbf{false}\rangle$ where $str \notin \cdots$
- $\sigma E\langle str\ \textbf{matches}\ P\rangle \to \sigma E\langle \textbf{true}\rangle$
- $\sigma E\langle str\ \textbf{matches}\ P\rangle \to \sigma E\langle \textbf{false}\rangle$
- $\sigma E\langle str_1\ \texttt{+}\ str_2\rangle \to \sigma E\langle str_1 str_2\rangle$

■

Lemma 19 (Preservation) *If $\Sigma_1 \vdash e_1 : T$, $\Sigma_1 \vdash \sigma_1$, and $\sigma_1 e_1 \to \sigma_1 e_2$, then there exists a Σ_2, such that:*

i. $\Sigma_2; \cdot \vdash e_2 : T$,
ii. $\Sigma_2 \vdash \sigma_2$, *and*
iii. $\Sigma_1 \subseteq \Sigma_2$.

Proof: By case-analysis of the reduction rules, there exists an evaluation context, E, an active expression, ae, and an expression, e', such that $e_1 = E\langle ae\rangle$ and $e_2 = E\langle e'\rangle$. There thus exists a subdeduction $\Sigma_1; \cdot \vdash ae : S$ of the original typing derivation. Lemma 18 now applies, so we have $\Sigma_2 \subseteq \Sigma_1$, $\Sigma_2 \vdash \sigma_2$, and $\Sigma_2; \cdot \vdash e' : S$. Replacing the original subdeduction, we have $\Sigma_2; \cdot \vdash E\langle e'\rangle : T$. ■

Theorem 4 (Typed Progress) *If $\Sigma \vdash \sigma$ and $\Sigma; \cdot \vdash e : T$ then either $e \in v$ or there exist σ' and e' such that $\sigma e \to \sigma' e'$.*

Proof: By case-analysis of the reduction rules, either $e \in v$, $e = E\langle ae\rangle$, or $e = E\langle \textbf{err}\rangle$. By inspection of the typing relation, **err** is untypable. We therefore consider the case where $e = E\langle ae\rangle$ by case-analysis on the definition of active expressions.

- The cases where ae is of the form $v_1\texttt{(}v_2\texttt{)}$, $\textbf{ref}\ v$, $\textbf{deref}\ v$, $v_1\ \texttt{=}\ v_2$, and $\textbf{if}\ \texttt{(}v_1\texttt{)}\ \texttt{\{}\ e_2\ \texttt{\}}\ \textbf{else}\ \texttt{\{}\ e_3\ \texttt{\}}$ are routine.
- Consider $ae = v_1\texttt{[}v_2\texttt{]}$. By inversion, $\Sigma; \cdot \vdash v_1 : \{L_1^{p_1} : T_1 \cdots L_n^{p_n} : T_n\}$ and $\Sigma; \cdot \vdash v_2 : L_Q$. By canonical forms, $v_1 = \{str_1 : w_1 \cdots str_m : w_m\}$ and $\{str_1^{\downarrow} : S_1 \cdots str_m^{\downarrow} : S_m, \overline{\{str_1 \cdots str_m\}}\} <: \{L_1^{p_1} : T_1 \cdots L_n^{p_n} : T_n\}$. Also by canonical forms, $v_2 = str_Q$ and $str_Q <: L_Q$.
  If $str_Q \in \{str_1 \cdots str_m\}$, then E-GetField applies.
  If $str_Q \notin \{str_1 \cdots str_m\}$, then we show that `"parent"` $\in \{str_1 \cdots str_m\}$ so that E-Inherit applies. This holds by the 2nd case of *inherit*, which requires that `"parent"` exist if $str_Q \notin \{str_1 \cdots str_m\}$.

- Consider $ae = v_1$`[`v_2 `=` v_3`]`. By inversion, $\Sigma; \cdot \vdash v_1 : \{L^p : S \cdots\}$ and $\Sigma; \cdot \vdash v_2 : L_Q$. By canonical forms, $v_1 = \{str : w \cdots\}$ and $v_2 = str_Q$. Thus either E-Create or E-Update apply.
- Consider $ae =$ **delete** v_1`[`v_2`]`. By inversion, $\Sigma; \cdot \vdash v_1 : \{L^p : S \cdots\}$ and $\Sigma; \cdot \vdash v_2 : L_Q$. By canonical forms, $v_1 = \{str : w \cdots\}$ and $v_2 = str_Q$. Thus either E-Delete or E-Delete-None apply.
- Consider $ae = v_1$ **hasfield** v_2. By inversion, $\Sigma; \cdot \vdash v_1 : \{L^p : S \cdots\}$ and $\Sigma; \cdot \vdash v_2 : L_Q$. By canonical forms, $v_1 = \{str : w \cdots\}$ and $v_2 = str_Q$. Thus either E-HasField or E-HasNotField apply.
- Consider $ae = v$ **matches** P. By inversion and canonical forms, $v = str$. Thus either E-Matches or E-NoMatch apply.
- Consider $ae =$ **fieldin** `{` $s_1 : v_1$ `,` $s_2 : v_2$ $\cdots$`}` **init** v_{acc} **do** v_f. By inversion, $\Sigma; \cdot \vdash v_1 : \{L^p : S \cdots\}$. By canonical forms, $v_1 = \{str : w \cdots\}$. Thus either E-FieldIn or E-FieldIn-End apply.
- Consider $ae = v_1$`+` v_2. By inversion and canonical forms, $v_1 = str_1$ and $v_2 = str_2$. Thus E-Str+ applies.
- Consider $ae =$ **fix** `(`f`:`S`)`. `e`. E-Fix applies trivially.