A Examples in Scripting Languages

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

Example 1

Prototype inheritance:

```javascript
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:

```javascript
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:

```lua
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:

```javascript
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.

```javascript
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.

```lua
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:

```lua
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:

```python
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:

```ruby
let safeGetField = Aα <= 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.

```javascript
function safeGetField(obj, field, default) {
  if(/\_\.*\_/i.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.

```python
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.

```ruby
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:

```javascript
let safeAssign = Lambda(dict, word, value).dict["w_" + word] = value
let safeLookup = Lambda(dict, word, default).dict[lookup]
if (dict hasfield lookup) dict[lookup]
else default
```

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

```javascript
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 dictionaries, 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 \( \text{func}(x:T) \{ e \} \) instead of \( \text{func}(x:T).e \).

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

\[
M_A \subseteq L_A \quad L_A \subseteq M_A \quad \forall i, j. L_i \cap M_j \neq \emptyset \Rightarrow S_i =_T T_j \land p_i = q_j
\]

\[
\{ L_1^{p_1} : S_1, \ldots, L_n^{p_n} : S_n, L_A : \text{abs} \} =_T \{ M_1^{q_1} : T_1, \ldots, M_n^{q_n} : T_m, M_A : \text{abs} \}
\]

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,

\[
ST : \mathcal{P}(\Gamma \times T \times T + p \times p) \rightarrow \mathcal{P}(\Gamma \times T \times T + p \times p)
\]

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

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

\[
TR(R) = \{(\Gamma, x, z) \mid \forall x, z \in T, \exists y \in T, (\Gamma, x, y), (\Gamma, y, z) \in R \}
\]

\[
\cup \{(z, z) \mid \forall x, z \in p, \exists y \in p, (x, y), (y, z) \in R \}
\]

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

\[
TD(R) = \{(T, T) \mid T \text{ is a finite type } \}
\]

\[
\cup \{(S, \{L^p : T, \text{rest} \ldots\}) \mid (S, T) \in R\}
\]

\[
\cup \{(S, \{L^p : T, \text{rest} \ldots\}) \mid (S, \{\text{rest}\}) \in R\}
\]

\[
\cup \{(S, \{L^p : T\}) \mid (S, T) \in R\}
\]

\[
\cup \{(S, \text{Ref } T) \mid (S, T) \in R\}
\]

\[
\cup \{(S, \mu x.T) \mid (S, T[x/\mu x.T]) \in R\}
\]

\[
\cup \{(S, T_1 \rightarrow T_2) \mid (S, T_1) \in R\}
\]

\[
\cup \{(S, T_1 \rightarrow T_2) \mid (S, T_2) \in R\}
\]

\[
\cup \{(S, \forall \alpha < U.T) \mid (S, U) \in R\}
\]

\[
\cup \{(S, \forall \alpha < U.T) \mid (S, T) \in R\}
\]

25
Definition 5 (Bottom-Up Subexpressions) $S$ is a bottom-up subexpression of $T$, written $S \preceq T$, if $(S,T)$ is in $\mu BU$, defined as follows:

$$\text{BU}(R) = \{ (T,T) \mid T \text{ is a finite type } \}$$

$$\cup \{ (S, \{ \text{lp} : T, \text{rest} \cdots \}) \mid (S,T) \in R \}$$

$$\cup \{ (S, \{ \text{lp} : T, \text{rest} \cdots \}) \mid (S,\{\text{rest}\}) \in R \}$$

$$\cup \{ (S, \{ \text{lp} : T \}) \mid (S,T) \in R \}$$

$$\cup \{ (S, \text{Ref } T) \mid (S,T) \in R \}$$

$$\cup \{ (S, T_1 \rightarrow T_2) \mid (S,T_1) \in R \}$$

$$\cup \{ (S, T_1 \rightarrow T_2) \mid (S,T_2) \in R \}$$

$$\cup \{ (S, \forall \alpha <: U.T) \mid (S,U) \in R \}$$

$$\cup \{ (S, \forall \alpha <: U.T) \mid (S,T) \in R \}$$

$$\cup \{ (S[x/\mu x.T], \mu x.T) \mid (S,T) \in R \}$$

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

$$ae = v_1(v_2)$$

$$\text{fix (f:T), e}$$

$$v_1 + v_2$$

$$\text{ref } v$$

$$\text{deref } v$$

$$v_1 = v_2$$

$$v_1 [v_2]$$

$$v_1[v_2] = v_3$$

$$\text{delete } v_1[v_2]$$

$$\text{if } (v_1) \{ e_2 \} \text{ else } \{ e_3 \}$$

$$v_1 \text{ hasfield } v_2$$

$$v_1 \text{ matches } v_2$$

$$\text{fieldin } \{ s_1 : v_1, s_2 : v_2 \cdots \} \text{ init } v_{acc} \text{ do } v_f$$

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, $\mathcal{T}_T^\downarrow$ 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 $T$. 26
The second position in the each right-hand clause in $BU$ is smaller than the left.

**Lemma 5** If $(S, T[x/U]) \in T_{[x/U]t}$, then either $(S, U) \in T^+_U$ or $S = S'[x/U]$ for some $(S', T) \in T^+_T$.

**Proof** By case analysis on $T$.

### D Subtyping

**Lemma 6** If:

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

is a monotone function, and for all $R$, $TR(S(R)) \subseteq S(TR(R))$, then $\nu S$ 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)** $TR(\nu ST) \subseteq \nu ST$

**Proof:**

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

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

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

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

- $U = \{L^p_i : U_1, \ldots, L^p_n : U_n, L_A : \text{abs}\}$.

Since $(\Gamma, S, U), (\Gamma, U, T) \in ST(R)$, by cases of $ST$,

(1) $S = \{K^{p_1}_i : S_1, \ldots, K^{p_n}_i : S_n, K_A : \text{abs}\}$, and

(2) $T = \{M^q_i : T_1, \ldots, M^q_m : T_m, M_A : \text{abs}\}$.

By hypothesis, $(\Gamma, S, U) \in ST(R)$, 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) $\bigcup_{i}^{1..n} L_i \subseteq \bigcup_{h=1}^{h-1} K_h \cup K_A$.

(4') $L_A \subseteq K_A$,

(5) $\forall i.$ if $L_i \cap K_A \neq \emptyset$ then $(p_j = o$ or $p_j = \uparrow)$,

(6) $\forall i.$ if $p_i = \uparrow$ then $(\Gamma, \text{inherit}(S, L_i), U_i) \in R$.

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) \( \bigcup_{j}^{1:m} M_j \subseteq \bigcup_{i}^{1:m} L_i \cup L_A \).
(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 \((\Gamma, \text{inherit}(U, M_j), T_j) \in R \).

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

### Proof:

a. \( \bigcup_{j}^{1:m} M_j \subseteq \bigcup_{h}^{1:m} K_h \cup K_A \).

**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, \((o_h, p_i) \in R \) and \((p_i, q_j) \in R \Rightarrow (o_h, q_j) \in TR(R) \), and \((S_h, U_i) \in R \). Thus \((U_i, T_j) \in R \Rightarrow (S_h, T_j) \in TR(R) \).

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, \( M_j \cap L_A \neq \emptyset \), so by (5) \( p_i = \circ \) or \( p_i = \uparrow \). Only \( p \)-Ref 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 \((\Gamma, \text{inherit}(S, M_j), T_j) \in TR(R)\).

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

- **Case U = L_u:** Only case S-Str applies for \( S <: U \) and \( U <: T \). Thus, \( S = L_u \) and \( T = L_T \), with \( L_u \subseteq L_u \subseteq L_T \). Thus \( S <: T \) follows by transitivity of the subset relation.

- **Case U = Ref U’:** Only case S-Ref applies, thus \( S = \text{Ref} S’ \) and \( T = \text{Ref} T’ \), with \((S’, U’), (U’, S’), (U’, T’), (T’, U’) \in R \). By definition of TR, \((S’, T’), (T’, S’)) \in TR(R) \). Thus, \((\text{Ref} S’, \text{Ref} T’) \in ST(TR(R)) \).
- 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:
  - $S = \forall \alpha <: U_1, S_2$,
  - $(\Gamma, S_2, U_2) \in ST(R)$,
  - $(\Gamma, \alpha <: U_1, U_2) \in ST(R)$.
  - $U = \mu \alpha.T$ This case is addressed in [19, chapter 21].
- $U = U_1 \rightarrow U_2$ See [19, page 288]

---

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

**Proof:** By case analysis on the subtyping rules.

**Lemma 9** $ST$ 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 $\mu$-types, the set $reachables_{ST}(S, T)$ is finite [19, Proposition 21.9.11]. Thus, the algorithm $gfp_{ST}$ [19, Definition 21.5.5] terminates [19, Theorem 21.5.12].

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

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

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

**Lemma 11** If $L_Q \subseteq M_Q \subseteq \bigcup_{j=1}^{m} M_j$ and $\forall i, j. L_i \cap M_j \neq \emptyset \Rightarrow \Gamma \vdash S_i' <: T_j'$ then

\[
\Gamma \vdash \bigcup_{i} \{S_i' | \Gamma \vdash L_Q \cap L_i \neq \emptyset\} <: \bigcup_{j} \{T_j' | \Gamma \vdash M_Q \cap M_j \neq \emptyset\}
\]
Proof: It is sufficient to show that for all $S'_i$ on the left-hand side, there exists a $T'_j$ such that $\Gamma \vdash S'_i <: T'_j$.

For any $S'_i$, since $L_i \cap L_Q \neq \emptyset$, $\exists \text{str} \in L_i \cap L_Q$. 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 $\text{str} \in \bigcup \{M_j | q_j \neq \circ\}$. Therefore, $\exists M_j, \text{str} \in M_j$, hence $L_i \cap M_j \neq \emptyset$ and so $\Gamma \vdash S'_i <: T'_j$. 

Lemma 12 For all $\Gamma, S, T, L_Q, M_Q$, if:

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

then $\Gamma \vdash S' <: T'$.

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

$$S = \{L^1_p : S'_1, \ldots, L^n_p : S'_n, L_A : \text{abs}\}$$
$$T = \{M^1_p : T'_1, \ldots, M^m_p : T'_m, M_A : \text{abs}\}$$

We thus have available the hypotheses of S-Object:

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

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

By definition of $\text{inherit}$, the goal is:

$$\Gamma \vdash \bigcup_{i}^{1..n} \{S'_i | \Gamma \vdash L_Q \cap L_i \neq \emptyset\} <: \bigcup_{j}^{1..m} \{T'_j | \Gamma \vdash M_Q \cap M_j \neq \emptyset\}$$

The condition on both applications of $\text{inherit}$ are $L_Q \subseteq \bigcup \{L_i | p_i \neq \circ\}$ and $M_Q \subseteq \bigcup \{M_j | q_j \neq \circ\}$. 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. $\exists L_i, "\text{parent}\" \in L_i \land S'_i = \text{Ref } S_P$,
H5'. $L_Q \subseteq \bigcup_{i}^{1..n} L_i \cup L_A$,
H6. $\exists M_i, "\text{parent}\" \in M_i \land T'_i = \text{Ref } T_P$, and
H6'. $M_Q \subseteq \bigcup_{j}^{1..m} M_j \cup M_A$.

Hind. $\forall L_Q, M_Q, S_P, T_P, |S_P| < |T| \land \Gamma \vdash L'_Q \subseteq M_Q \land \Gamma \vdash S_P <: T_P \Rightarrow (\text{inherit}_P(S_P, L'_Q) = S'_P \land \text{inherit}_P(T_P, M'_Q) = T'_P \Rightarrow \Gamma \vdash S'_P <: T'_P)$.
The goal thus reduces to:

\[ \Gamma \vdash \bigcup \{ S'_i \mid \Gamma \vdash L_Q \cap L_i \neq \emptyset \} \cup \text{inherit}(S_p, \mathcal{L}) <:\bigcup \{ T'_i \mid \Gamma \vdash M_Q \cap M_i \neq \emptyset \} \cup \text{inherit}(T_p, \mathcal{M}) \]

where \( \mathcal{L} = L_Q \cap (L_A \cup \bigcup \{ L_i | p_i = \circ \}) \) 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 \( S \):

\[ \mathcal{N} = \{ M_j \mid M_Q \cap M_j \cap L_A \neq \emptyset \land q_j = \uparrow \} \]

\[ \mathcal{L}^+ = \mathcal{L} \cap \bigcup \mathcal{N} \]

\[ \mathcal{L}^- = \mathcal{L} \cap \bigcup \overline{\mathcal{N}} \]

Using lemma 10, we rewrite the goal to:

\[ \Gamma \vdash \bigcup \{ S'_i \mid \Gamma \vdash L_Q \cap L_i \neq \emptyset \} \cup \text{inherit}(S_p, \mathcal{L} \cap \bigcup \mathcal{N}) \cup \text{inherit}(S_p, \mathcal{L} \cap \bigcup \overline{\mathcal{N}}) <:\bigcup \{ T'_i \mid \Gamma \vdash M_Q \cap M_i \neq \emptyset \} \cup \text{inherit}(T_p, \mathcal{M}) \]

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

a. \( \Gamma \vdash \bigcup \{ S'_i \mid \Gamma \vdash L_Q \cap L_i \neq \emptyset \} <:\bigcup \{ T'_i \mid \Gamma \vdash M_Q \cap M_i \neq \emptyset \} \)

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 \( L_Q \cap L_A \cap L_i \). Similarly, \( M_Q \cap M_j \neq \emptyset \) iff \( M_Q \cap M_A \cap M_j \). We can therefore rewrite the subgoal to \( \Gamma \vdash \bigcup \{ S'_i \mid \Gamma \vdash L_Q \cap L_i \neq \emptyset \} <:\bigcup \{ T'_i \mid \Gamma \vdash M_Q \cap M_j \neq \emptyset \} \).

Lemma 11 now applies.

b. \( \Gamma \vdash \text{inherit}(S_p, \mathcal{L} \cap \bigcup \mathcal{N}) <: \text{inherit}(T_p, \mathcal{M}) \)

By induction (HInd). The following cases allow us to apply HInd.

- \( |S_p| < |S| \) and \( |T_p| < |T| \) are trivial.
- We show that \( \Gamma \vdash S_p <: T_p \). By H5 and H6, "\text{parent}" \( \in L_P \cup M_P \) thus \( L_P \cap M_P \neq \emptyset \). Therefore, \( \Gamma \vdash \text{Ref} S_p <: \text{Ref} T_P \) by 11. By (S-Ref), it follows that \( \Gamma \vdash S_P <: T_P \).
- We show that \( \Gamma \vdash \mathcal{L} \cap \bigcup \mathcal{N} \subseteq \mathcal{M} \). It is sufficient to show that \( \forall x. \Gamma \vdash x \in \mathcal{L} \cap \bigcup \mathcal{N} \Rightarrow \Gamma \vdash x \in \mathcal{M} \).

By definition of \( \mathcal{L} \), \( x \in L_Q \), thus by (H2), \( x \in M_Q \).

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 \( x \in L_i \cup M_j \), \( p_i <: q_j \), and by p-Ref, \( q_j = \circ \).

Therefore, \( x \in \mathcal{M} \) since it is in both sets.

c. \( \Gamma \vdash \text{inherit}(S_p, \mathcal{L} \cap \bigcup \overline{\mathcal{N}}) <:\bigcup \{ T'_i \mid \Gamma \vdash M_Q \cap M_j \neq \emptyset \} \)

Rewrite the left-hand side by expanding the definition of \( \mathcal{N} \), distributing the intersection over the union, and applying lemma 10:

\[ \Gamma \vdash \bigcup \{ \text{inherit}(S_p, \mathcal{L} \cap M_j) \mid M_Q \cap M_j \cap L_A \neq \emptyset \land q_j = \uparrow \} <:\bigcup \{ T'_i \mid \Gamma \vdash M_Q \cap M_j \neq \emptyset \} \]
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 \( \text{inherit}(S_p, L \cap M_j) \) on the left-hand side, the associated \( T'_j \) is on the right-hand side by definition of \( N \). We now show that

\[ \Gamma \vdash \text{inherit}(S_p, L \cap M_j) <: T'_j \]

Since \( M_j \cap L_A \neq \emptyset \) and \( q_j = \top \), 14 applies and \( \Gamma \vdash \text{inherit}(S, M_j) <: T'_j \).

By lemma 10 \( \text{inherit}(S, M_j) = \text{inherit}(S, L \cap M_j) \sqcup \text{inherit}(S, L \cap M_j) \), so \( \Gamma \vdash \text{inherit}(S, L \cap M_j) <: T'_j \) by the definition of joins. Further, by the definition of \( \text{inherit} \),

\[ \Gamma \vdash \text{inherit}(S, L \cap M_j) = \text{inherit}(S_p, L \cap M_j \cap L_A \bigcup \{L_i | p_i = \circ\}) \sqcup \ldots \]

and by the definition of \( L \), \( L \cap M_j \cap L_A \bigcup \{L_i | p_i = \circ\} \) is the same as \( L \cap M_j \).

By the definition of joins:

\[ \Gamma \vdash \text{inherit}(S_p, L \cap M_j) <: T'_j \]

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.

By well-formedness of types, if \( \exists i. \text{"parent"} \in M_i \) then \( q_i = \top \). But by I2, "parent" \( \in \bigcup_{i=1}^{m} L_i \cup L_A \), 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. \( \exists L_P. \text{"parent"} \in L_P \land S'_i = \text{Ref } S_P \),

H6. \( L_Q \subseteq \bigcup_{i=1}^{m} L_i \cup L_A \),

HRP. If \( \neg \exists M_P. \text{"parent"} \in M_P \), and

H8. \( M_Q \subseteq \bigcup_{i=1}^{m} M_i \{q_i \neq \circ\} \).

By HRP, since \( \neg \exists M_P. \text{"parent"} \in M_P \). The goal is therefore:

\[ \Gamma \vdash \bigcup_{i=1}^{m} \{S'_i | \Gamma \vdash L_Q \cap L_i \neq \emptyset \} \sqcup \text{inherit}(S_P, L_Q \cap (L_A \bigcup \{L_i | p_i = \circ\})) <: \bigcup_{j=1}^{m} \{T'_j | \Gamma \vdash M_Q \cap M_j \neq \emptyset\} \]

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

\[ \Gamma \vdash \text{inherit}(S_P, L_Q \cap (L_A \bigcup \{L_i | p_i = \circ\})) <: \bigcup_{j=1}^{m} \{T'_j | \Gamma \vdash M_Q \cap M_j \neq \emptyset\} \]

to:

\[ \Gamma \vdash \text{inherit}(S_P, L_Q \cap L_A) \sqcup \text{inherit}(S_P, L_Q \cap \{L_i | p_i = \circ\}) <: \bigcup_{j=1}^{m} \{T'_j | \Gamma \vdash M_Q \cap M_j \neq \emptyset\} \]

There are two cases.
Lemma 13 If \( \{ L_1 : F_1 \cdots L_n : F_n \} \prec : \{ M_1 : G_1 \cdots M_m : G_m \} \), then \( \bigcup_{j=1}^{m} \{ M_j \mid G_j = T^j \} \subseteq \bigcup_{i=1}^{n} \{ L_i \mid F_i = T^i \} \).

**Proof:** By contradiction.

Assume there exists some string \( x \) with \( x \in \bigcup_{j=1}^{m} \{ M_j \mid G_j = T^j \} \) and \( x \notin \bigcup_{i=1}^{n} \{ L_i \mid F_i = T^i \} \). By assumption of S-Object, \( \bigcup_{j=1}^{m} M_j \subseteq \bigcup_{i=1}^{n} L_i \), so it must be that there is some \( L_i \) that contains \( x \), but either has type \( T^o \) or \( \text{abs} \). That is, there must be an \( M_j \) with \( x \in M_j \) and an \( L_i \) with \( x \in L_i \) with \( G_j = T^j \) and either \( F_i = \text{abs} \) or \( F_i = T^\alpha \). This violates S-Object, which asserts that since \( L_i \cap M_j \neq \emptyset \), it must be that \( F_i \prec G_j \), which cannot happen since possibly absent and definitely absent fields cannot subtype definitely present fields.

---

**E Typing**

\[
\begin{align*}
\text{IfHasField1} & \quad \Gamma \vdash v : S \cdots \Gamma(f) = L \quad \Sigma; \Gamma, \alpha <: L, f : \alpha \vdash e_2 : T \quad L' = L \cap \pi \quad \Gamma \vdash e_3 : T \\
\quad & \quad \Sigma; \Gamma \vdash \text{if } (\{ \text{str} : v \cdots \} \text{ hasfield } f) e_2 \text{ else } e_3 : T \\
\text{IfHasField2} & \quad \Gamma(o) = \{ \cdots L^o : S \cdots \} \quad \Sigma; \Gamma, o : \{ \cdots \text{str}^k : S, L^o : S \cdots \} \vdash e_2 : T \\
\quad & \quad L' = L \cap \{ \text{str} \} \quad \Gamma \vdash e_3 : T \\
\quad & \quad \Sigma; \Gamma \vdash \text{if } (o \text{ hasfield } \text{str}) e_2 \text{ else } e_3 : T \\
\text{IfHasFieldFalse} & \quad \Gamma \vdash v : S \cdots \Sigma; \Gamma \vdash e_3 : T \quad \text{str}_2 \notin \text{str} \cdots \\
\quad & \quad \Sigma; \Gamma \vdash \text{if } (\{ \text{str} : v \cdots \} \text{ hasfield } \text{str}_2) e_2 \text{ else } e_3 : T \\
\text{IfFalse} & \quad \Sigma; \Gamma \vdash e_3 : T \\
\quad & \quad \Sigma; \Gamma \vdash \text{if } (\text{false}) e_2 \text{ else } e_3 : T
\end{align*}
\]

Fig. 6. Auxiliary Typing Rules for If-Splitting

---

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

---

33
Lemma 15 (Inversion) If:

- \{L^1_1 : S_1 \cdots L^m_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 : U_1 \cdots str_n : U_n, \{str_1 \cdots str_n\} : \text{abs}\} and \{str_1 : U_1 \cdots str_n : U_n, \{str_1 \cdots str_n\} : \text{abs}\} \ll \{L^1_1 : S_1 \cdots L^m_m : S_m\},
- \text{Ref } S, \text{ then } v = \text{loc and } \Sigma(\text{loc}) \ll S,
- S \rightarrow T, \text{ then } v = \text{func}(x) \ {f} \ {e} \ {j},
- L \text{ then } v = \text{str and } \Gamma \vdash \text{str} < L

**Proof:** By induction on the typing derivation.

**Lemma 15 (Inversion) If:**

- \Sigma; \Gamma \vdash \{\text{str} : v_1 \cdots \} : T then \Sigma; \Gamma \vdash v : S \cdots \text{ and } \Gamma \vdash \{\text{str} : S \cdots\} \ll T
- \Sigma; \Gamma \vdash e_1 + e_2 : T then \Sigma; \Gamma \vdash e_1, e_2 < \text{Str and } \Gamma \vdash T \ll \text{Str}
- \Sigma; \Gamma \vdash \text{fix } (f : S). e : T then \Sigma; \Gamma, f : S \vdash e < S \text{ and } \Gamma \vdash S \ll T
- \Sigma; \Gamma \vdash \text{ref } e : T then \Sigma; \Gamma \vdash e : S \text{ and } \text{Ref } S = T
- \Sigma; \Gamma \vdash ! : T \text{ then } \Sigma(!) = T
- \Sigma; \Gamma \vdash \text{deref } e : T, \text{ then } \Sigma; \Gamma \vdash e : \text{Ref } S \text{ with } S \ll T,
- \Sigma; \Gamma \vdash e_1 = e_2 : T, \text{ then } \Sigma; \Gamma \vdash e_1 : \text{Ref } S, \Sigma; \Gamma \vdash e_2 : U, U < S, \text{ and } \text{Ref } S < T,
- \Sigma; \Gamma \vdash e f (e \cdots) : T, \text{ then } \Sigma; \Gamma \vdash e f : S \cdots \rightarrow T', \Sigma; \Gamma \vdash e : S \cdots, \text{ and } T' < T.
- \Sigma; \Gamma \vdash e_o e f_j : T, \text{ then } \Sigma; \Gamma \vdash e_o : \{L^1_1 : S_1 \cdots\}, \Sigma; \Gamma \vdash e f : L, \text{ inherit}(\{L^1_1 : S_1 \cdots\}, L) = T', \text{ and } T' < T.
- \Sigma; \Gamma \vdash e_o e f_j = e o : T, \text{ then } \Sigma; \Gamma \vdash e_o : \{L^1 : S \cdots, L_A : \text{abs}\}, \Sigma; \Gamma \vdash e f : L', \Sigma; \Gamma \vdash e o : U, \forall L.\lfloor L \cap L' \neq \emptyset \rfloor \text{ then } U < S, \text{ and } \Gamma \vdash \{L^1 : S \cdots, L_A : \text{abs}\} \ll T.
- \Sigma; \Gamma \vdash \text{delete } e_o e f_j : T, \text{ then } \Sigma; \Gamma \vdash e_o : \{L^1_1 : S_1 \cdots\}, \Sigma; \Gamma \vdash e f : L, \forall L \cap L_1 \neq \emptyset. F_1 \neq T', \Gamma \vdash \{L^1_1 : S_1 \cdots\} < T
- \Sigma; \Gamma \vdash e_1 \text{ hasfield } e_2 : \text{Bool}, \text{ 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 \text{ matches } P : \text{Bool}, \text{ then } \Sigma; \Gamma \vdash e : L.
- \Sigma; \Gamma \vdash \text{if } (\forall x) \ {e_2} \ {\text{else}} \ {e_3} : T, \text{ then } \Sigma; \Gamma \vdash v_1 : \text{Bool}, \Sigma; \Gamma \vdash e_2 : T, \text{ and } \Sigma; \Gamma \vdash e_3 : 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 (e hasfield f) e2 else e3 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, $v = \{\text{str}; v \ldots\}$ 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 antecedents 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, $v = \text{str}$, $\Sigma; \Gamma \vdash v : \text{str}$, and $\Gamma \vdash \text{str} <: L$. By type substitution followed by induction, $\Sigma; \Gamma, o : \{\text{str}^1 : S, L^o : S \ldots\} \vdash e_2 : T$. 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:

\[
\text{if } (\text{str}; v' \ldots \text{hasfield } str') e_2 \text{ else } e_3
\]

There are two cases.

1. If $\text{str}' \in \text{str} \ldots$ then by type substitution and induction, $\Sigma; \Gamma, \alpha <: L, f : \alpha e_2 : T[\alpha/\text{str}][f/v] = \Sigma; \Gamma \vdash e_2[f/v] : T$. By induction, $\Sigma; \Gamma \vdash e_3[f/v] : T$. Finally, the conditional has type $\text{Bool}$. Thus the expression is typable by T-If.

2. If $\text{str}' \notin \text{str} \ldots$ then the term is trivially typable by T-IfHasFieldFalse.

There are two cases.

1. If $\text{str}' \in \text{str} \ldots$ then the result is trivially typable by T-If.

2. If $\text{str}' \notin \text{str} \ldots$ then the term is trivially typable by T-IfHasFieldFalse. □

Lemma 18 (Main Preservation) If $\Sigma_1 \vdash ae : T$, $\Sigma_1 \vdash \sigma_1$, and $\sigma_1 E(\langle ae\rangle) \rightarrow \sigma_2 E(\langle e_2\rangle)$ then there exists a $\Sigma_2$, such that:

1. $\Sigma_2 \supseteq \Sigma_1$,
2. $\Sigma_2 \vdash \sigma_2$, and
3. $\Sigma_2 \vdash e_2 : T$.

35
Proof: By case-analysis on \( ae \), using inversion (lemma 15) where specified:

\[-\sigma_1 E((\text{func } (x:S') \ {e} \ y)(v)) \rightarrow \sigma_1 E(e[x/v]).\]

By inversion, \( \Sigma_1; \vdash v : S, \vdash S < : S', \Sigma_1; x : S \vdash e : T', \) and \( \vdash T' < : T.\)

By substitution (lemma 17), \( \Sigma_1; \vdash e[x/v] : T.\)

\[-\sigma_1 E(\text{fix } (x : S, e)) \rightarrow \sigma_1 E(e[fix (x:S). e]).\]

By inversion, \( \Sigma_1; \vdash e : S, \Sigma_1; x : S \vdash e : S, \) and \( \vdash S < : T.\) By substitution (lemma 17), \( \Sigma_1; \vdash e[x/fixed (x:S). e] : T.\)

\[-\sigma_1 E((\lambda a < : S.e)(U)) \rightarrow \sigma_1 E(e[a/U]).\]

By type substitution (lemma 16).

\[-\sigma_1 E(\text{ref } v) \rightarrow \sigma, (l,v) E(l)\] where \( l \notin \text{dom}(\sigma).\) By inversion, \( \Sigma_1; \vdash v : S \) and \( \text{Ref } S < : T.\) Let \( \Sigma_2 = l : S, \Sigma_1.\) By T-Loc, \( \Sigma_2; \vdash l : \text{Ref } S.\)

\[-\sigma_1 E(\text{deref } l) \rightarrow \sigma E(\text{e}(l))\]

By inversion (T-SetRef), \( \Sigma, \Gamma \vdash l : \text{Ref } S, \Sigma, \Gamma \vdash v : S, \) and \( \text{Ref } S = T.\) By inversion (T-Loc), \( \Sigma(l) = T \) thus \( \Sigma \vdash \sigma[l := v].\) By T-Loc, \( \Sigma, \Gamma \vdash l : T.\)

\[-\sigma_1 E(\{ \cdot \cdot \cdot :: v \cdot \cdot \cdot \} \rightarrow \sigma E(v)).\] By inversion, \( \Sigma_1; \vdash str : L, \Sigma_1; \vdash \{ \cdot \cdot \cdot :: v \cdot \cdot \cdot \} : S, \Sigma_1; \vdash v : T', T' < : \text{inherit.}(S, L), \) and \( \text{inherit.}(S, L) < : T.\) By inversion, \( \{ \cdot \cdot \cdot :: T' \cdot \cdot \cdot \} < : S.\) Thus \( \vdash T' < : T.\)

By lemma 12, \( \text{inherit.}(\{ \cdot \cdot \cdot :: T' \cdot \cdot \cdot \}, str) < : \text{inherit.}(S, L).\) By [REF], \( \text{inherit.}(\{ \cdot \cdot \cdot :: T' \cdot \cdot \cdot \}, str) = T'.\)

\[-\sigma_1 E(\{ \cdot \cdot \cdot :: \text{"parent" : } l \cdot \cdot \cdot \} \rightarrow \sigma E(\text{deref } l [\text{str}]),\] where \( \text{str} \notin \cdot \cdot \cdot \).

By inversion of the left-hand side, \( \Sigma_1; \vdash str : L, \Sigma_1; \vdash \{ \cdot \cdot \cdot :: \text{"parent" : } T_p \} : S, \) and \( \text{inherit.}(S, L) < : T.\) By lemma 12, \( \text{inherit.}(\{ \cdot \cdot \cdot :: \text{"parent" : } T_p \}, str) < : \text{inherit.}(S, L).\) By inversion, \( \Sigma_1; \vdash l : \text{Ref } S_p = T_p.\) Since \( \text{str} \notin \cdot \cdot \cdot \) and by definition of \( \text{inherit.},\) \( \text{inherit.}(\{ \cdot \cdot \cdot :: \text{"parent" : } \text{Ref } S_p \}, str) = \text{inherit.}(S_p, str),\) which is a subtype of \( T.\)

Type right-hand side with T-Sub and T-GetField, using \( \Sigma_1; \vdash str : str, \text{inherit.}(S_p, str) < : T,\) and \( \Sigma_1; \vdash \text{ deref } l : S_p.\) This holds since \( \Sigma_1; \vdash l : \text{Ref } S_p\) above.

\[-\sigma_1 E(\{ \cdot \cdot \cdot :: v \cdot \cdot \cdot \} \rightarrow \sigma E(\{ \cdot \cdot \cdot :: \text{v' \cdot \cdot \cdot} \})\]

By inversion (T-Update), \( \Sigma, \Gamma \vdash \{ \cdot \cdot \cdot :: v \cdot \cdot \cdot \} : \{ L : S \cdot \cdot \cdot \}, \Gamma \vdash \{ L : S \cdot \cdot \cdot \} < : T, \) and \( \Sigma; \Gamma \vdash \text{str } : U'.\) By inversion (T-Object), \( \Sigma; \Gamma \vdash \{ \cdot \cdot \cdot :: v \cdot \cdot \cdot \} : \{ S \cdot \cdot \cdot \} \land \Gamma \vdash \{ \cdot \cdot \cdot :: str : U \cdot \cdot \cdot \} < : \{ L : S \cdot \cdot \cdot \}.\) Thus by inversion (T-Update), \( \Gamma \vdash \text{str } : U < : U.\) The resulting expression is typable by S-Object, thus by S-Sub, \( \Gamma \vdash \{ \cdot \cdot \cdot :: str : U \cdot \cdot \cdot \} < : \{ \cdot \cdot \cdot :: str : U \cdot \cdot \cdot \} < : T.\)

\[-\sigma_1 E(\{ \cdot \cdot \cdot :: \text{str } = \text{v'} \}) \rightarrow \sigma E(\{ \cdot \cdot \cdot \})\] where \( \text{str} \notin \cdot \cdot \cdot \).

By inversion of T-Update, \( \Sigma; \Gamma \vdash \{ \cdot \cdot \cdot :: str : v \cdot \cdot \cdot \} : S \land \Gamma \vdash S < : T.\)

\[-\sigma_1 E(\text{delete } \{ \cdot \cdot \cdot :: \text{str } \cdot \cdot \cdot \}) \rightarrow \sigma E(\{ \cdot \cdot \cdot \})\] Similar to to E-UpdateField case above.

\[-\sigma_1 E(\text{delete } \{ \cdot \cdot \cdot :: \text{str } \cdot \cdot \cdot \}) \rightarrow \sigma E(\text{delete } \{ \cdot \cdot \cdot \})\] Similar to to E-UpdateField case above.

\[-\sigma_1 E(\text{fieldin } \{ s : v, \text{rest} \cdot \cdot \cdot \} \rightarrow \text{init } v_{acc} \text{ do } v_{j}) \rightarrow \sigma E(\text{fieldin } \{ s : v, \text{rest} \cdot \cdot \cdot \} \rightarrow \text{init } v_{f}(\text{str}[\text{i}]) (v_{acc}) \text{ do } v_{j}\}

By inversion, \( \Sigma; \Gamma \vdash v_{acc} : T \land \Sigma; \Gamma \vdash v_{f} : (\text{Str } \rightarrow T) \rightarrow T.\) The double application can be typed by T-App, and the resulting expression will by typable by T-FieldIn.
Proof: there exists a ε

Theorem 4 (Typed Progress)

Lemma 19 (Preservation) If Σ₁ ⊨ e₁ : T, Σ₁ ⊨ σ₁, and σ₁ e₁ → σ₁ e₂, then there exists a Σ₂, such that:

i. Σ₂; ⊨ e₂ : T,
ii. Σ₂ ⊨ σ₂, and
iii. Σ₁ ⊆ Σ₂.

Proof: By case-analysis of the reduction rules, there exists an evaluation context, E, an active expression, ae, and an expression, e', such that e₁ = E(ae) and e₂ = E(e'). There thus exists a subderivation Σ₁; ⊨ ae : S of the original typing derivation. Lemma 18 now applies, so we have Σ₂ ⊆ Σ₁, Σ₂ ⊨ σ₂, and Σ₂; ⊨ e' : S. Replacing the original subderivation, we have Σ₂; ⊨ E(e') : T.

Theorem 4 (Typed Progress) If Σ ⊨ σ and Σ; ⊨ e : T then either e ∈ v or there exist σ' and e' such that σ e → σ' e'.

Proof: By case-analysis of the reduction rules, either e ∈ v, e = E(ae), or e = E(err). By inspection of the typing relation, err is untypable. We therefore consider the case where e = E(ae) by case-analysis on the definition of active expressions.

The cases where ae is of the form v₁(ε₂), ref v, deref v, v₁ = v₂, and if (v₁) { ε₂ } else { ε₃ } are routine.

Consider ae = v₁[ε₂]. By inversion, Σ; ⊨ v₁ : {L₁^p₁ : T₁, ..., Lⁿ^p : Tⁿ} and Σ; ⊨ v₂ : L. By canonical forms, v₁ = {str⁻¹ : w₁, ..., str⁻m : w⁻m} and {str₁ : S₁, ..., str⁻m : S⁻m}. Also by canonical forms, v₂ = str₁ and str⁻m < L.

If str⁻m ∈ {strₙ}, then E-GetField applies.

If str⁻m ∉ {strₙ}, then we show that "parent" ∈ {str₁, ..., str⁻m} so that E-Inherit applies. This holds by the 2nd case of inherit, which requires that "parent" exist if str⁻m ∉ {str₁, ..., str⁻m}.
Consider $ae = v_1[v_2 = v_3]$. By inversion, $\Sigma;\vdash v_1 : \{L^p : S \cdots\}$ and $\Sigma;\vdash v_2 : LQ$. By canonical forms, $v_1 = \{str : w \cdots\}$ and $v_2 = strQ$. Thus either E-Create or E-Update apply.

Consider $ae = delete v_1[v_2]$. By inversion, $\Sigma;\vdash v_1 : \{L^p : S \cdots\}$ and $\Sigma;\vdash v_2 : LQ$. By canonical forms, $v_1 = \{str : w \cdots\}$ and $v_2 = strQ$. Thus either E-Delete or E-Delete-None apply.

Consider $ae = v_1 hasfield v_2$. By inversion, $\Sigma;\vdash v_1 : \{L^p : S \cdots\}$ and $\Sigma;\vdash v_2 : LQ$. By canonical forms, $v_1 = \{str : w \cdots\}$ and $v_2 = strQ$. 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, $\Sigma;\vdash v_1 : \{L^p : S \cdots\}$. 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-Fix applies trivially.