Fluid Object Types

Arjun Guha, Joe Gibbs Politz, and Shriram Krishnamurthi

Brown University

Abstract. Objects in popular scripting languages are lightweight and dynamic. Programmers use these objects in ways that confound existing type systems. We present a core calculus, λ_S^{ob}, that distills the essence of objects in a number of scripting languages. Using λ_S^{ob}, we examine characteristic object-oriented scripting idioms that are untypable by conventional type systems. We develop *fluid object types*, a novel generalization of structural object types. Fluid object types can specify the presence or absence of fields, their position in the inheritance chain, and employ string pattern types to describe (possibly infinite) collections of fields. We have implemented fluid object types in our experimental type-checker for JavaScript. We briefly report on our experience with our prototype.

1 Introduction

In most statically-typed object-oriented languages, an object's type or class enumerates its member names and their types. This set of names is finite and the names are first-order, barring cumbersome reflection APIs. In contrast, in many "scripting" languages, reflection is trivial, class hierarchies are fluid, objects' shapes are amorphous, and member names are first-class strings that can be computed dynamically. These features are difficult to type-check, so it is unsurprising that scripting languages are untyped.

This paper presents *fluid object types*, a type language for describing the dynamic, reflective idioms employed by scripting languages. Fluid object types have two novel features: they employ *string patterns* to describe possibly-infinite collections of fields and *presence annotations* to precisely state the position of inherited fields. We develop fluid object types for λ_S^{ob}, a core calculus of lightweight, dynamic objects that can express idioms employed by JavaScript, Lua, Python, and Ruby.

In section 2, we present the runtime semantics of λ_S^{ob}. In section 3, we present a sequence of λ_S^{ob} programs that demonstrate its expressivity and highlight type-checking challenges. These examples distill scripting idioms employed by real-world code (appendix A). Section 4 develops fluid object types and discusses typing our examples. We detail subtyping and typing in sections 5 and 6. Section 7 discusses our prototype type-checker that employs fluid object types.

2 λ_S^{ob}: A Core Calculus of Lightweight Objects

Figure 1 defines the syntax and runtime semantics of λ_S^{ob}, a calculus of lightweight objects. The objects of λ_S^{ob} are extensible records with inheritance and field-

$$
\begin{array}{rll}
P = \cdots & & \text{Patterns} \\
c = num \mid str \mid bool & & \text{Constants} \\
v = c \mid \mathsf{func}(x:T).e \mid \Lambda\alpha <: T.e \mid \texttt{\{}\ str\texttt{:}\ v \cdots\ \texttt{\}} & & \text{Values} \\
e = x \mid v \mid e\texttt{(}e\texttt{)} \mid \texttt{\{}\ str\texttt{:}\ e \cdots\ \texttt{\}} \mid e\texttt{[}e\texttt{]} \mid e\texttt{[}e\ \texttt{=}\ e\texttt{]} \mid \mathsf{delete}\ e\texttt{[}e\texttt{]} \mid e\texttt{(}T\texttt{)} & & \text{Expressions} \\
\quad \mid \mathsf{if}\ \texttt{(}e_1\texttt{)}\ e_2\ \mathsf{else}\ e_3 \mid e_1\ \mathsf{hasfield}\ e_2 \mid e\ \mathsf{matches}\ P & & \\
\quad \mid \mathsf{fix}\ \texttt{(}f\texttt{:}T\texttt{)}.e \mid e_1\ \texttt{+}\ e_2 & & \\
E = \bullet \mid E\texttt{(}e\texttt{)} \mid v\texttt{(}E\texttt{)} \mid \texttt{\{}\ str\texttt{:}\ v \cdots\ str\texttt{:}E\texttt{,}\ str\texttt{:}e \cdots\ \texttt{\}} \mid E\texttt{[}e\texttt{]} \mid v\texttt{[}E\texttt{]} & & \text{Contexts} \\
\quad \mid v\texttt{[}v\ \texttt{=}\ E\texttt{]} \mid E\texttt{[}e\ \texttt{=}\ e\texttt{]} \mid v\texttt{[}E\ \texttt{=}\ e\texttt{]} \mid \mathsf{delete}\ E\texttt{[}e\texttt{]} \mid \mathsf{delete}\ v\texttt{[}E\texttt{]} & & \\
\quad \mid \mathsf{if}\ \texttt{(}E\texttt{)}\ e_2\ \mathsf{else}\ e_3 \mid E\ \mathsf{hasfield}\ e \mid v\ \mathsf{hasfield}\ E \mid E\ \mathsf{matches}\ P & & \\
\quad \mid E\ \texttt{+}\ e \mid v\ \texttt{+}\ E & &
\end{array}
$$

$$
\begin{array}{ll}
\beta_v & \mathsf{(func}(x:T).e\mathsf{)}(v) \hookrightarrow e[x/v] \\
\text{E-Fix} & \mathsf{fix}(f:T).e \hookrightarrow e[f/\mathsf{fix}(f:T).e] \\
\text{E-TApp} & (\Lambda\alpha:S.e)(T) \hookrightarrow e[\alpha/T] \\
\text{E-GetField} & \texttt{\{}\cdots str\texttt{:}\ v \cdots\ \texttt{\}[}str\texttt{]} \hookrightarrow v \\
\text{E-Inherit} & \texttt{\{}\ str\ \texttt{:}\ v \cdots\ \texttt{"parent":}\ v_p\ \texttt{\}[}str_x\texttt{]} \hookrightarrow v_p\texttt{[}str_x\texttt{]}, \text{ if } str_x \notin (str \cdots) \\
\text{E-Update} & \texttt{\{}\cdots str\texttt{:}\ v \cdots\ \texttt{\}[}str\ \texttt{=}\ v_x\texttt{]} \hookrightarrow \texttt{\{}\cdots str\texttt{:}\ v_x \cdots\ \texttt{\}} \\
\text{E-Create} & \texttt{\{}\ str\texttt{:}\ v \cdots\ \texttt{\}}\ \texttt{[}str_x\ \texttt{=}\ v_x\texttt{]} \hookrightarrow \texttt{\{}\ str_x\texttt{:}\ v_x\texttt{,}\ str\texttt{:}\ v \cdots\ \texttt{\}} \text{if } str_x \notin (str \cdots) \\
\text{E-Del} & \mathsf{delete}\ \texttt{\{}\cdots\ str_x\texttt{:}\ v_x\ \cdots\ \texttt{\}}\ \texttt{[}str_x\texttt{]} \hookrightarrow \texttt{\{}\cdots\texttt{\}} \\
\text{E-NoDel} & \mathsf{delete}\ \texttt{\{}\ str\texttt{:}\ v\ \cdots\ \texttt{\}}\ \texttt{[}str_x\texttt{]} \hookrightarrow \texttt{\{}\ str\texttt{:}\ v\ \cdots\ \texttt{\}} \text{if } str_x \notin (str \cdots) \\
\text{E-IfTrue} & \mathsf{if}\ \texttt{(}\mathsf{true}\texttt{)}\ e_2\ \mathsf{else}\ e_3 \hookrightarrow e_2 \\
\text{E-IfFalse} & \mathsf{if}\ \texttt{(}\mathsf{false}\texttt{)}\ e_2\ \mathsf{else}\ e_3 \hookrightarrow e_3 \\
\text{E-Has} & \texttt{\{}\cdots str\texttt{:}v \cdots\ \texttt{\}}\ \mathsf{hasfield}\ str \hookrightarrow \mathsf{true} \\
\text{E-HasNot} & \texttt{\{}\ str\texttt{:}v \cdots\ \texttt{\}}\ \mathsf{hasfield}\ str\ \hookrightarrow \mathsf{false}, \text{ when } str' \notin (str \cdots) \\
\text{E-Match} & str\ \mathsf{matches}\ P \hookrightarrow \mathsf{true},\ str \in P \\
\text{E-}\neg\text{Match} & str\ \mathsf{matches}\ P \hookrightarrow \mathsf{false},\ str \notin P \\
\text{E-String+} & str_1 + str_2 \hookrightarrow str_1\, str_2
\end{array}
$$

$$\text{E-Cxt } E\langle e_1\rangle \to E\langle e_2\rangle, \text{ when } e_1 \hookrightarrow e_2$$

$$
\begin{array}{rcl}
\mathsf{let}\ x\ \texttt{=}\ e_1\ \mathsf{in}\ e_2 & \equiv & \mathsf{(func}(x:T).e_2\mathsf{)}(e_1) \\
\mathsf{let\ rec}\ x\ \texttt{=}\ e_1\ \mathsf{in}\ e_2 & \equiv & \mathsf{(func}(x:T).e_2\mathsf{)(fix}\ x:T.e_1\mathsf{)} \\
\mathsf{func}(x_1:T_1 \cdots x_n:T_n).e & \equiv & \mathsf{func}(x_1:T_1).\cdots.\mathsf{func}(x_n:T_n).e \\
e_f(e_1 \cdots e_n) & \equiv & e_f(e_1)\cdots(e_n)
\end{array}
$$

Fig. 1. Syntax and Runtime Semantics of λ_S^{ob}

deletion. λ_S^{ob} distinguishes `"parent"` as a reference to the parent object for inheritance (E-Inherit). However, `"parent"` is otherwise undistinguished; a program can retrieve, update, or even delete `"parent"`. Field names are first-class strings, not first-order labels. A single program may work with an arbitrary collection of field names, or even dynamically construct names by concatenating strings (E-String+). λ_S^{ob} programs can use reflection (**hasfield**) to determine which fields are present on an object. In addition, λ_S^{ob} programs can use **matches** to determine

if a string matches a particular pattern. We leave the representation of patterns unspecified. (Other languages use a mix of regular expressions and ad hoc testing to achieve the same effect.) Finally, λ_S^{ob} includes explicit type annotations and instantiations; we introduce types in section 4.

3 Idiomatic λ_S^{ob}: Type-Checking Challenges

The examples below demonstrate that λ_S^{ob} can model many characteristic uses of objects in scripting languages. Indeed, we translate them to various real-world scripting languages in appendix A. For now, we leave the static types unspecified. We type-check these examples in section 4.5, after developing fluid object types.

Example 1: Prototype-based Objects λ_S^{ob} can easily encode prototype inheritance:

```
let Rect = { "area": func(self:?) . self["x"] * self["y"] } 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
```

Above, methods are simply function valued-fields that take an explicit self argument. With this encoding, a program can directly access the `"parent"` field to redefine methods or apply methods to other objects.

Example 2: Extracting Methods Supporting method-extraction promotes code reuse across structurally similar values. For example, arrays may have a collection of utility methods in their common parent:

```
let ArrParent = { "slice": func(self:?,begin:?,end:?). ··· , ··· } in
let arr1 = { "0": 3, "1": 20, "2": 59, "length": 3, "parent": ArrParent }
```

The implementation of slice may only require that `self["len"]` be defined; it does not actually need self to be an array. Therefore, this single slice method can be copied and applied to other kinds of collections. This exact scenario occurs frequently in JavaScript programming. Web browsers have other kinds of collections, such as HTMLNodeList, that are not arrays and thus do not have various array methods:

```
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
```

However, in the code above, slice is applied to an HTMLNodeList to convert it to an array and make other utility methods (e.g., map and reduce) available.

Example 3: Classes λ_S^{ob} can also encode classes by having methods close over a particular value of self. Although explicitly passing self to each method is enormously flexible, it is error-prone.[1] We build on the prototypal shapes of Example 1 below:

```
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
```

In this encoding, methods do not require an explicit self argument, but the underlying methods are still accessible via the `"_class_"` field.[2] This encoding allows methods themselves to manipulate objects' structure. However, it also allows other code to freely modify objects. We will use types to hide the _class_ field from other code.

Example 4: Ad Hoc Private Fields All fields are public in λ_S^{ob}'s lightweight objects. In such object systems, it is common to use a convention, such as "field names that begin with an underscore are private".[3] This convention is easily violated by malicious or buggy code. A module may wish to protect some of the fields in objects in its implementation, and only provide client code with access to a safe lookup function.

For example, a simple dynamic check to ensure that untrusted code does not access fields that begin and end with underscores is as follows:

```
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 },
               "_private_", 0) // returns 0
```

These checks are notoriously difficult to implement in full-fledged scripting languages. Our type system for λ_S^{ob} will demonstrate techniques for statically verifying that such code is correct.

Example 5: Dictionaries as Dictionaries There is no need to implement dictionaries in λ_S^{ob}; objects can be used as dictionaries themselves. Some care must be taken in implementation, however, as a naïve approach may accidentally extract a method from the object's parent, or the parent itself:

[1] JavaScript suffers exactly this error; `this` is an implicit argument but is supplied at each method call [11].

[2] Python and Ruby have similar encodings; methods close over their self argument, but the underlying method is still accessible.

[3] Python and Dart employ such conventions.

```
let ObjectParent = { "serialize": func(self:?).···, ··· } in
let dict = { "habitat": "a natural home or environment",
             "park": "a large enclosed piece of ground",
             "parent": ObjectParent } in
dict["habitat"] // returns "a natural home or environment"
dict["serialize"] // returns func(self) ... !
dict["parent"] // returns ObjectParent!
```

To avoid such mishaps, programmers should guard dynamic dictionary accesses:[4]

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

This example presents a safe alternative, which prefixes words with `"w_"` and then uses a dynamic check to ensure only words are accessed. Our type system reasons about field tests to give reasonable static types to dictionaries.

Type System Features These examples elicit a baseline feature set to typecheck:

1. Addition and deletion of fields,
2. Prototype-based inheritance with method extraction,
3. Class-based inheritance with bound and unbound method extraction,
4. Invariants on pattern-based field conventions, and
5. Field-presence guards on field access.

4 Types for Objects in Scripting Languages

The preceding examples guide the development of fluid object types in this section. Our full type language is in figure 2, but we motivate and incrementally develop fluid object types in this section. In subsequent sections, we detail the associated type system.

4.1 Simple Records

Structural record types are a natural starting point, since λ_S^{ob} does not have classes. Recall that record types are finite maps from field names to field types:

$$T = \cdots \mid \{str_1 : T_1 \cdots str_n : T_n\}$$

Record types can type trivial programs, but not those in section 3.

$$
\begin{array}{rll}
L = & P \mid \alpha \mid L_1 \cap L_2 \mid L_1 \cup L_2 \mid L_1 L_2 \mid \overline{L} & \text{String patterns} \\
b = & \mathsf{Num} \mid \mathsf{Bool} & \text{Base types} \\
\alpha = & \cdots & \text{Type variables} \\
T = & b \mid T_1 \to T_2 \mid \mu\alpha.T \mid \top \mid \forall \alpha <: S.T & \\
& \mid \; L \mid \{L_1^{p_1} : T_1 \cdots L_n^{p_n} : T_n, L_A : \mathbf{abs}\} & \text{Types} \\
p = & \circ \mid \downarrow \mid \uparrow & \text{Field presence} \\
\Gamma = & \cdot \mid \Gamma, x : T \mid \Gamma, \alpha <: T & \text{Environments}
\end{array}
$$

$\boxed{\Gamma \vdash T}$

$$\text{WF-Object}\ \frac{\Gamma \vdash T_1 \cdots \Gamma \vdash T_n \qquad \forall i. L_i \cap L_A = \emptyset \text{ and } \forall j \neq i. L_i \cap L_j = \emptyset}{\Gamma \vdash \{L_1^{p_1} : T_1, \cdots, L_n^{p_n} : T_n, L_A : \mathbf{abs}\}}$$

$$\text{WF-TVar}\ \frac{\alpha \in dom(\Gamma)}{\Gamma \vdash \alpha}$$

Fig. 2. Types for λ_S^{ob}

$\boxed{P : \{str\}}$ $\quad P_1 \cap P_2 \quad P_1 \cup P_2 \quad \overline{P} \quad P_1 \subseteq P_2 \quad P_1 P_2 \quad P = \emptyset$

Fig. 3. Functions and Predicates over String Types

4.2 Field Patterns

Record types are too simple because they statically enumerate all their fields' names. Interesting programs use computed field lookups, where the exact field name is not statically known. Consider the type of `c` in the following example:

```
let getCoord = func(pt:{"x" : Num, "y" : Num}, c:?) . pt[c]
```

The intended type for `c` is not an arbitrary string (Str), but a string in the set {`"x"`, `"y"`}. We thus extend our types with precise string types, L-types, that denote sets of strings:

$$
\begin{array}{ll}
L = P & \text{String patterns} \\
T = \cdots \mid L &
\end{array}
$$

The exact representation of L-types is not significant; we only require that the functions and predicates in figure 3 be defined.

Type-checking field lookup with L-typed strings is straightforward. If `c` has type {`"x"`, `"y"`}, then `pt[c]` has type Num. In general, if *fld* has type L and *obj* has

[4] A fact the developers of Google Docs know all too well. At the time of this writing, typing `__proto__`, JavaScript's `parent`, crashes Google Docs: `www.google.com/support/forum/p/Google+Docs/thread?tid=0cd4a00bd4aef9e4`.

type $\{str_1 : T_1 \cdots str_n : T_n\}$ then at runtime $obj\texttt{[}fld\texttt{]}$ may lookup any $str_i \in L$. Thus, the type of $obj\texttt{[}fld\texttt{]}$ is the join of the matching fields' types.

Patterns Replace Field Names L-types allow computed field names, but record types still specify a finite list of fields. They cannot express the type of the dictionary in Example 5, so we reuse L-types to generalize field names to patterns:

$$T = \cdots \mid \{L_1 : T_1 \cdots L_n : T_n\}$$

This is the first of the two key features of fluid object types. We can now describe an object with an arbitrary collection of fields. For example, in the following type, all fields that begin with an underscore have type Num and all other fields have type Bool.[5]

$$\{\texttt{\_.*} : \mathsf{Num}, \overline{\texttt{\_.*}} : \mathsf{Bool}\}$$

Something odd happened here—this type seems to describe an object with an infinite number of fields! Since objects are finite dictionaries, we could interpret this type as a specification of fields' types *if they are present*; if a field is absent then indexing that field gets stuck, but typing is preserved. We do better and address this problem with *presence annotations* in the next section.

In summary, fluid object types use patterns (L-types) to describe collections of fields in objects. There are three common classes of L-types:

- The L-type for the set of all strings is the usual Str-type.
- An L-type that represents a singleton set, $\{str\}$, is the type of the string literal str. In $e_1\texttt{[}e_2\texttt{]}$, if $e_2 : \{str\}$ then the expression is a conventional, known-label field lookup.
- An L-type that represents a possibly-infinite set of strings is useful for typing operations on computed field names. For example, in object update, $e_1\texttt{[}e_2\texttt{ = }e_3\texttt{]}$, if the type of e_2 is the co-finite set $\overline{\{\texttt{"parent"}\}}$, then the operation does not affect `"parent"` and does not affect the inheritance chain.

4.3 Presence Annotations

Conventional structural object types do not expose the position of members on the inheritance chain; types are "flattened" to include inherited members. However, the `"parent"` field of λ_S^{ob} objects allows programs to distinguish inherited fields. If we flatten object types, all such programs would be untypable.

Fluid object types allow us to expose the precise structure of the inheritance chain with ease. In the following type, move is present on the parent:

$$\{\texttt{"parent"} : \{\texttt{"move"} : \mathsf{Num} \to \mathsf{Num}\}\}$$

This precision unfortunately makes conventional uses of class hierarchies untypable. Functions that consume subtypes of a particular class are agnostic to

[5] We often use regular expressions to describe patterns. However, regular expressions are not fundamental; any decision procedure over strings is adequate.

the position of methods on the inheritance chain; all that matters are methods' types. However, the type above requires move to be present on precisely the first parent, and does not admit objects that inherit but do not override move.

To remedy this, we introduce the second feature of fluid object types. We add *presence annotations* to fields and a pattern for fields that are *definitely absent*:

$$\begin{aligned} p &= \downarrow \mid \uparrow \mid \circ \\ T &= \cdots \mid \{L_1^{p_1} : T_1, \cdots, L_n^{p_n} : T_n, L_A : \mathbf{abs}\} \end{aligned}$$

We interpret presence annotations as follows:

- $L^{\downarrow} : T$ indicates that all fields $str \in L$ are *definitely present on the object itself* with type T. These fields are not inherited and are not absent.
- $L^{\uparrow} : T$ indicates that all fields $str \in L$ are either *present on the object itself or along the inheritance chain* with type T.
- $L^{\circ} : T$ indicates that all fields $str \in L$ *may be present on the object itself*. If a field *str* is present it has type T, but it may be absent.
- The $L_A : \mathbf{abs}$ annotation indicates that all fields $str \in L_A$ are *definitely absent* on the object; however, they may be present higher up on the inheritance chain. Looking up a definitely absent field therefore does not fail if the field is inherited.

Notation We use two abbreviations to simplify the syntax of fluid objects.

- We elide writing an empty set of absent fields, $\emptyset : \mathbf{abs}$. For example:

$$\{\texttt{"x"}^{\downarrow} : \mathsf{Num}, \texttt{"y"}^{\downarrow} : \mathsf{Num}\} = \{\texttt{"x"}^{\downarrow} : \mathsf{Num}, \texttt{"y"}^{\downarrow} : \mathsf{Num}, \emptyset : \mathbf{abs}\}$$

- It is often convenient for the set of absent fields to be the complement of all other fields; we abbreviate this to $\star : \mathbf{abs}$. For example, $\star$ represents $\overline{\{\texttt{"x"}, \texttt{"y"}, \texttt{"parent"}\}}$ in the following type:

$$\{\texttt{"x"}^{\downarrow} : \mathsf{Num}, \texttt{"y"}^{\downarrow} : \mathsf{Num}, \texttt{"parent"}^{\downarrow} : P, \star : \mathbf{abs}\}$$

Flexibility and Guarantees of Presence Annotations The $L^{\downarrow} : T$ and $L^{\uparrow} : T$ annotations ensure that indexing an L-typed field produces a T-typed value; programs cannot get stuck indexing fields. They also allow other fields with the same name but different types to exist further up the inheritance chain, but these do not affect the type of object indexing.

The $L^{\circ} : T$ annotation provides weaker guarantees. For example, if obj has the type:

$$\{\texttt{"x"}^{\circ} : \mathsf{Num}, \texttt{"parent"}^{\downarrow} : \{\texttt{"x"}^{\downarrow} : \mathsf{Bool}\}\}$$

then obj["x"] may return either field. For full generality, we present a typing rule that determines that obj["x"] has type $\mathsf{Num} \sqcup \mathsf{Bool}$. What would be more useful is an operator that narrows the $\circ$ to a $\downarrow$.

4.4 Reflection

Consider typing Example 5 from section 3. We might give dict the following type:

$$\{\mathsf{w\_.*}^{\circ} : \mathsf{Str}\}$$

With this type, the expression `dict["w_habitat"]` is not typable. The type indicates that `"w_habitat"` may not exist on the object, so we must also compute the type of `dict["parent"]["w_habitat"]`. However, `"parent"` is not specified.

We must first establish that `dict["w_habitat"]` is definitely present. In Example 5, the safeLookup function guards dictionary indexing with a **hasfield** check:

```
if (dict hasfield lookup) dict[lookup] else default
```

The type system can *if-split* [25] to account for such guards and narrow the types of dict and lookup in the true branch. We present a simple if-splitting rule in section 6; here we describe the types in the true-branch after narrowing.

In the true-branch, we have established that the value of lookup names a member of dict; we can thus narrow a possibly absent member ($\circ$) to definitely present ($\downarrow$). However, we do not know exactly which member to narrow. In particular, the following narrowing is wrong:

$$\mathsf{dict} : \{\mathsf{w\_.*}^{\downarrow} : \mathsf{Str}\}$$

The type above states that all members are definitely present. However, the program only establishes that a single string is definitely present.

We express this by splitting the pattern $\mathsf{w\_.*}$ into two components, a type variable α that represents the string bound to lookup and $\mathsf{w\_.*} \cap \overline{\alpha}$, the remainder of the pattern. Thus in the true-branch, the type-environment, Γ', modifies the enclosing type environment, Γ, as follows:

$$\Gamma' = \Gamma, \alpha <: P, \mathsf{word} : \alpha, \mathsf{dict} : \{\alpha^{\downarrow} : \mathsf{Str}, P \cap \overline{\alpha}^{\circ} : \mathsf{Str}\}$$
$$\text{where } P = \mathsf{w\_.*}$$

The patterns used above include type variables and set operators:

$$L = P \mid \alpha \mid L_1 \cap L_2 \mid L_1 \cup L_2 \mid L_1 L_2 \mid \overline{L}$$

The final specification of fluid object types and string pattern types is in figure 2. With these types, we can type-check the examples from section 3.

4.5 Type-Checking Examples

With fluid object types introduced, we now revisit the examples from section 4.5.

Example 1 The types for Rect and Cuboid are as follows:

$$\mathsf{Rect}: \quad \left\{ \texttt{"area"}^{\downarrow} : \{\texttt{"x"}^{\downarrow} : \mathsf{Num}, \texttt{"y"}^{\downarrow} : \mathsf{Num}\} \to \mathsf{Num} \right\}$$

$$\mathsf{Cuboid}: \left\{ \begin{array}{l} \texttt{"vol"}^{\downarrow} : \left\{ \begin{array}{l} \texttt{"x"}^{\downarrow} : \mathsf{Num}, \texttt{"y"}^{\downarrow} : \mathsf{Num}, \texttt{"z"}^{\downarrow} : \mathsf{Num}, \\ \texttt{"area"}^{\uparrow} : \{\texttt{"x"}^{\downarrow} : \mathsf{Num}, \texttt{"y"}^{\downarrow} : \mathsf{Num}\} \to \mathsf{Num} \end{array} \right\} \to \mathsf{Num}, \\ \texttt{"parent"}^{\downarrow} : \mathsf{Rect} \end{array} \right\}$$

We abuse notation slightly for brevity; in Cuboid we use Rect as an abbreviation for the whole type. The presence annotations on these types are interesting. On Rect's `"area"` method, the argument omits listing absent fields and only specifies fields the method needs. This allows an object like shape, which has a `"z"` field, to be freely used with `"area"`. `Cuboid["vol"]` marks area as inherited (↑), since it is agnostic to its position on the inheritance chain.

Example 2 ArrParent only requires `"slice"`'s argument to have a Num-typed `"len"` field and consistently-typed numeric fields:

$$\forall\alpha <: \top.\mu\beta. \left\{ \begin{array}{ll} \texttt{"slice"}^{\downarrow} & : \{\mathsf{Dec}^{\circ} : \alpha, \texttt{"len"}^{\downarrow} : \mathsf{Num}\} \\ & \to \{\mathsf{Dec}^{\circ} : \alpha, \texttt{"len"}^{\downarrow} : \mathsf{Num}, \texttt{"parent"}^{\downarrow} : \beta\} \\ \cdots & \cdots \end{array} \right\}$$

$$\mathsf{Dec} = \texttt{0}\,|\,\texttt{[1-9]}\,\texttt{[0-9]*}$$

Fields that match Dec may be present; if they are, they must have type Num as indicated by the regular expression pattern Dec. If ArrParent is instantiated with the type HTMLElement, then `"slice"` can be freely used on the object in Example 2.

Example 3 The types of Rect and Cuboid are the same as in Example 1 in this class encoding, but note that `shape2["vol"]` is closed over its self-argument. We can give shape2 the following type, hiding _class_, to prevent external code from observing its internals:

$$\{\texttt{"x"}^{\downarrow} : \mathsf{Num}, \texttt{"y"}^{\downarrow} : \mathsf{Num}, \texttt{"z"}^{\downarrow} : \mathsf{Num}, \texttt{"vol"}^{\uparrow} :\to \mathsf{Num}\}$$

Example 4 The type of safeGetField is:

$$\forall\alpha <: \top.\{\overline{\texttt{\_.*\_}}^{\circ} : \alpha\} \to \mathsf{Str} \to \alpha \to \alpha$$

The pattern $\overline{\texttt{\_.*\_}}^{\circ} : \alpha$ indicates that all non-underscored fields have type α, if they are present.

Example 5 The types in this example are:

$$\begin{array}{ll} \mathsf{ObjectParent} & : \{\texttt{"serialize"} : \top \to \mathsf{Str}\} \\ \mathsf{dict} & : \forall\alpha <: \top.\{\mathsf{w\_.*} : \alpha, \texttt{"parent"}^{\downarrow} : \mathsf{ObjectParent}, \texttt{"serialize"} : \mathbf{abs}\} \\ \mathsf{safeAssign} & : \forall\alpha <: \top.\forall\beta <: \{\mathsf{w\_.*}^{\circ} : \alpha\}.\beta \to \mathsf{Str} \to \alpha \to \beta \\ \mathsf{safeLookup} & : \forall\alpha <: \top.\{\mathsf{w\_.*}^{\circ} : \alpha\} \to \mathsf{Str} \to \alpha \to \alpha \end{array}$$

$\boxed{\Gamma \vdash S <: T}$

$$\frac{\Gamma \vdash T_a <: S_a \qquad \Gamma \vdash S_r <: T_r}{\Gamma \vdash S_a \to S_r <: T_a \to T_r} \qquad \frac{}{\Gamma \vdash b <: b} \qquad \frac{}{\Gamma \vdash T <: \top}$$

$$\frac{\Gamma \vdash S <: T[\alpha/\mu\alpha.T]}{\Gamma \vdash S <: \mu\alpha.T} \qquad \frac{\Gamma \vdash S[\alpha/\mu\alpha.S] <: T}{\Gamma \vdash \mu\alpha.S <: T} \qquad \frac{}{\Gamma \vdash \alpha <: \alpha}$$

$$\frac{\alpha <: S \in \Gamma \qquad \Gamma \vdash S <: T}{\Gamma \vdash \alpha <: T} \qquad \frac{\Gamma, \alpha <: U \vdash S <: T}{\Gamma \vdash (\forall \alpha <: U.S) <: (\forall \alpha <: U.T)}$$

$$\text{S-Str}\frac{L_1 \subseteq L_2}{\Gamma \vdash L_1 <: L_2}$$

$$\text{S-Ob}\frac{\begin{array}{c}(1)\forall i,j.\text{if } L_i \cap M_j \neq \emptyset \text{ then } p_i <: q_j \text{ and } \Gamma \vdash S_i <: T_j \\ (2)\bigcup_i^{1\cdots m} M_i \subseteq \bigcup_j^{1\cdots n} L_j \cup L_A \qquad (2')M_A \subseteq L_A \\ (3)\forall j.\text{if } M_j \cap L_A \neq \emptyset \text{ then } q_j = \circ \text{ or } q_j = \uparrow \\ (4)\forall j.\text{if } q_j = \uparrow \text{ then } \Gamma \vdash \mathit{inherit}_\Gamma(\{L_1^{p_1} : S_1, \cdots, L_n^{p_n} : S_n, L_A : \mathbf{abs}\}, M_j) <: T_j\end{array}}{\Gamma \vdash \{L_1^{p_1} : S_1, \cdots, L_n^{p_n} : S_n, L_A : \mathbf{abs}\} <: \{M_1^{q_1} : T_1, \cdots, M_m^{q_m} : T_m, M_A : \mathbf{abs}\}}$$

$\boxed{p <: q}$

$$p\text{-Refl}\frac{}{p <: p} \qquad p\text{-PMaybe}\frac{}{\downarrow <: \circ} \qquad p\text{-Inherit}\frac{}{\downarrow <: \uparrow} \qquad p\text{-IMaybe}\frac{}{\downarrow <: \circ}$$

$\boxed{\mathit{inherit}_\Gamma : S \times L \to T}$

$$\mathit{inherit}_\Gamma(\{L_1^{p_1} : T_1 \cdots L_n^{p_n} : T_n, L_A : \mathbf{abs}\}, L_Q) = \bigsqcup \{T_i \mid L_Q \cap L_i \neq \emptyset\} \cup \mathcal{T}_P$$

$$\mathcal{T}_P = \begin{cases} \emptyset & \text{if } L_Q \subseteq \bigcup\{L_i \mid p_i \neq \circ\} \text{ and} \\ & \neg\exists L_k.\texttt{"parent"} \in L_k \\ \{\mathit{inherit}_\Gamma\,(T_k, L_Q \cap (L_A \cup \bigcup\{L_i | p_i = \circ\}))\} & \text{if } L_Q \subseteq \bigcup_i^{1\cdots n} L_i \cup L_A \text{ and} \\ & \exists L_k.\texttt{"parent"} \in L_k \end{cases}$$

Fig. 4. Algorithmic Subtyping

We assume that serialize can serialize arbitrary values. Notably, dict's type allows serialize to be called, but the types of safeAssign and safeLookup ensure that they cannot access serialize and only manipulate words in the dictionary.

5 Subtyping

This section presents algorithmic subtyping for λ_S^{ob}. Figure 4 is the entire algorithmic subtyping relation. We present fluid object types along with equirecursive μ-types and bounded quantification (kernel rule). Some form of recursive type is necessary to type-check objects—we choose equirecursive μ-types.

Bounded quantification is commonly used to encode data structures; we also employ bounded quantification in our account of reflection. The majority of the algorithmic subtyping relation is conventional. The two interesting rules are S-Str and S-Ob.

Subtyping string types with S-Str uses pattern inclusion. For example, if patterns are defined as regular languages, inclusion is decidable. When patterns include variables, these inclusion constraints can be discharged by existing string solvers [15]. Discharging these patterns may require constraints on L-bounded type variables in Γ. The appropriate Γ is always unambiguous from context, therefore we write $L_1 \subseteq L_2$ instead of $\Gamma \vdash L_1 \subseteq L_2$.

Algorithmic subtyping for objects, S-Ob, is a generalization of algorithmic subtyping of records. Recall that algorithmic subtyping for records combines the declarative width, depth, and permutation subtyping rules. S-Ob combines generalizations of depth, width, and permutation subtyping, in addition to a *flattening* rule for inherited fields. For object types, $\Gamma \vdash S <: T$ if and only if they satisfy the four antecedents of S-Ob:

1. (Permutation and Depth) The types of fields with overlapping names must be subtypes. In addition, $p <: q$ is a partial order on presence-annotations. Intuitively, $p <: q$ means that T can "forget" that a field is definitely present.
2. (Width) The field patterns of S must include the field patterns of T. Therefore, T can "hide" fields of S.
3. (Depth) Any field that is absent (**abs**) in S may be possibly absent ($\circ$) or inherited ($\uparrow$) in T. This is depth subtyping for absent fields; T can "forget" that a field is absent and introduce it with $\circ$ or $\uparrow$ annotations.
4. (Flattening) For an inherited field ($\uparrow$) to appear on T, subtyping must ensure that the field is defined, with the appropriate type, somewhere on the inheritance chain of S. The metafunction *inherit* calculates this type on S and the pattern of the $\uparrow$-annotated field.

Flattening When $\Gamma \vdash S <: T$, the subtype S can describe the exact position of fields in the inheritance chain, e.g.

$$S = \{\texttt{"x"}^{\downarrow} : \mathsf{Num}, \texttt{"parent"}^{\downarrow} : \{\texttt{"y"}^{\downarrow} : \mathsf{Str}\}, \texttt{"y"} : \mathbf{abs}\}$$

However, T may lose this information and flatten the type to

$$T = \{\texttt{"x"}^{\downarrow} : \mathsf{Num}, \texttt{"y"}^{\downarrow} : \mathsf{Str}\}$$

The purpose of *inherit* is to ensure that $\uparrow$-annotated fields are appropriately flattened in the supertype, T. For an object type T, $\mathit{inherit}_{\Gamma}(T, L_Q)$ calculates the join of all field on S with patterns that may intersect L_Q, given constraints on L-bounded type variables in Γ. This includes all of the fields on the inheritance chain of S. Recall that fluid object types allow fields with the same pattern to have different types at different points in the inheritance chain. Consider the following type:

$$T = \{\texttt{"x"}^{\downarrow} : \mathsf{Num}, \texttt{"z"}^{\downarrow} : \mathsf{Num}, \texttt{"parent"}^{\downarrow} : \{\texttt{"z"}^{\downarrow} : \mathsf{Bool}\}\}$$

With this definition of T, $\mathit{inherit}(T, \texttt{"z"}) = \mathsf{Num}$,[6] since indexing a T-typed object with `"z"` cannot produce the Bool-valued field in the parent. However, consider the slightly different type:

$$T' = \{\texttt{"x"}^{\downarrow} : \mathsf{Num}, \texttt{"z"}^{\circ} : \mathsf{Num}, \texttt{"parent"}^{\downarrow} : \{\texttt{"z"}^{\downarrow} : \mathsf{Bool}\}\}$$

In T', since `"z"` may be absent ($\circ$) indexing may produce either `"z"` field. Therefore, $\mathit{inherit}(T', \texttt{"z"}) = \mathsf{Num} \sqcup \mathsf{Bool}$. Finally, consider the following type:

$$U = \{\texttt{"z"}^{\circ} : \mathsf{Num}\}$$

In U, `"z"` may be absent, but U does not specify the type of `"parent"`. It is thus unknown if `"parent"` exists, much less if it has a `"z"` field and what its type might be. Therefore, $\mathit{inherit}(U, \texttt{"z"})$ is undefined. We can see this by looking at the definition of *inherit*, specifically in defining $\mathcal{T}_P$ for U and `"z"`. The first case of $\mathcal{T}_P$ does not apply, because the annotation on `"z"` is $\circ$, contradicting the side condition. The second case does not apply either, because there is no `"parent"` field on U.

inherit requires a join operator, $\sqcup$, over types. The join operator must satisfy:

$$(S \sqcup T = U) \Rightarrow (S <: U \wedge T <: U)$$

We elide the full definition; the interesting case is for object types:

$$\{L_i^{p_i} : S_i, \cdots, L_A : \mathbf{abs}\} \sqcup \{M_j^{q_j} : T_j, \cdots, M_A : \mathbf{abs}\}$$

Which is a pairwise intersections of patterns:

$$\left\{ \begin{array}{l} (L_i \cap M_j)^{p_i \sqcup q_j} : S_i \sqcup T_j \cdots, \\ (L_i \cap M_A)^{\circ} : S_i \cdots, (L_A \cap M_j)^{\circ} : T_j \cdots, \\ L_A \cap M_A : \mathbf{abs} \end{array} \right\}$$

For computing the join of the presence annotations $p_i \sqcup q_j$, we use the lattice induced by the $p <: q$ relation in figure 4.[7]

Lemma 1 (Decidability of Subtyping) *If the functions and predicates on string patterns (figure 3) are decidable, then the subtype relation is finite-state.*

Proof: By coinduction on the subtyping judgments. The full proof is included in the supplemental materials.

6 Typing

Figure 5 presents the typing relation, which includes a conventional account of functions and bounded quantification. The interesting typing rules are marked and discussed below.

[6] An omitted Γ argument to *inherit* denotes an empty type environment.

[7] The join operator, as presented, may introduce a number of $\emptyset$ field patterns. Inspection of the typing rules shows that empty patterns are inconsequential.

$$ty(num) = \mathsf{Num} \qquad ty(bool) = \mathsf{Bool}$$

$\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma \vdash e : S \qquad \Gamma \vdash S <: T}{\Gamma \vdash e : T} \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{}{\Gamma \vdash c : ty(c)}$$

$$\frac{\Gamma, x : S \vdash e : T}{\Gamma \vdash \textbf{func}\ (x : S).e : S \to T} \qquad \frac{\Gamma \vdash e_f : S \to T \qquad \Gamma \vdash e_a : S}{\Gamma \vdash e_f(e_a) : T}$$

$$\frac{\Gamma, \alpha <: S \vdash e : T}{\Gamma \vdash \Lambda\alpha <: S.e : \forall\alpha <: S.T} \qquad \frac{\Gamma \vdash e : \forall\alpha <: U.T \qquad \Gamma \vdash S <: U}{\Gamma \vdash e(S) : T[\alpha/S]}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2, e_3 : T}{\Gamma \vdash \textbf{if}\ (e_1)\ e_2\ \textbf{else}\ e_3 : T}$$

$$\text{T-Str}\ \frac{}{\Gamma \vdash str : \{str\}} \qquad \text{T-Str+}\ \frac{\Gamma \vdash e_1 : L_1 \qquad \Gamma \vdash e_2 : L_2}{\Gamma \vdash e_1 + e_2 : L_1 L_2}$$

$$\text{T-Object}\ \frac{\Gamma \vdash e_1 : S_1 \cdots \Gamma \vdash e_n : S_n}{\Gamma \vdash \{\ str_1 : e_1, \cdots, str_n : e_n\ \} : \{str_1^{\downarrow} : S_1, \cdots, str_n^{\downarrow} : S_n, \star : \textbf{abs}\}}$$

$$\text{T-Update}\ \frac{\begin{array}{c}\Gamma \vdash e_o : T \qquad \Gamma \vdash e_f : L \qquad \Gamma \vdash e_v : S \qquad L \subseteq \bigcup\{L_1 \cdots L_n\} \\ T = \{L_1^{p_1} : T_1, \cdots, L_n^{p_n} : T_n, L_A : \textbf{abs}\} \qquad \forall L_i \cap L \neq \emptyset.\Gamma \vdash S <: T_i\end{array}}{\Gamma \vdash e_o[e_f\ =\ e_v] : T}$$

$$\text{T-Delete}\ \frac{\begin{array}{c}\Gamma \vdash e_o : T \qquad \Gamma \vdash e_f : L \qquad L \subseteq \bigcup\{L_1 \cdots L_n\} \qquad \forall L \cap L_i \neq \emptyset.p_i = \circ \\ T = \{L_1^{p_1} : T_1, \cdots, L_n^{p_n} : T_n, L_A : \textbf{abs}\}\end{array}}{\Gamma \vdash \textbf{delete}\ e_o[e_f] : T}$$

$$\text{T-GetField}\ \frac{\begin{array}{c}U = \{L_1^{p_1} : S_1, \cdots, L_n^{p_n} : S_n, L_A : \textbf{abs}\} \qquad \Gamma \vdash e_o : U \qquad \Gamma \vdash e_f : L_Q \\ T = inherit_{\Gamma}(U, L_Q)\end{array}}{\Gamma \vdash e_o[e_f] : T}$$

$$\text{T-HasField-Split}\ \frac{\begin{array}{c}\Gamma(o) = \{\cdots L^{\circ} : S \cdots\} \qquad \Gamma(f) = L \qquad \Gamma \vdash e_3 : T \\ \Gamma, \alpha <: L, f : \alpha, o : \{\cdots \alpha^{\downarrow} : S, (L \cap \overline{\alpha})^{\circ} : S \cdots\} \vdash e_2 : T\end{array}}{\Gamma \vdash \textbf{if}\ (o\ \textbf{hasfield}\ f)\ e_2\ \textbf{else}\ e_3 : T}$$

Fig. 5. Typing

T-Str is the typing rule for string literals, which ascribes a string *str* the singleton L-type $\{str\}$. T-Str+ concatenates two string patterns.

T-Object is the typing rule for object literals. In an object literal, all fields are definitely present ($str^{\downarrow}$) and all field names are statically known. Thus, each field name is a singleton L-type. In addition, since the object has no more fields, the complement of its fields is definitely absent ($\star : \textbf{abs}$).

Typing Object Operations Field update and field deletion are not affected by inheritance, so typing these operations does not treat `"parent"` specially. Both T-Update and T-Delete allow the field name to be an arbitrary L-typed expression, and not just a string constant. Therefore, at runtime, the index expression may reduce to any string, $str \in L$. The necessary restrictions are:

- For T-Update, if a field $(L_i^{p_i} : T_i)$ overlaps at all with the type of the field name to update (L), then the type of the value (S), must be a subtype of the type of the field (T_i).
- For T-Delete, if a field overlaps with the type of the field name to delete, then it must be possibly absent ($\circ$-annotated).

Object types are thus invariant under updates and deletion; we do not account for strong updates in this paper, but they are supported by other type systems for scripting languages (section 9).

T-GetField types field lookup and must therefore account for inheritance. To do so, it uses the *inherit* metafunction from object subtyping (section 5).

Typing Reflection T-HasField-Split is a simple *if-splitting* rule that accounts for the use of reflection in the conditional, and refines the types of o and f in the true-branch when the check succeeds. There are various sophisticated if-splitting techniques for typing complex conditionals and control [26,12,7]. T-HasField-Split could be adapted to the aforementioned systems. [8]

T-HasField-Split introduces a new type variable α in the true branch, bounded by $\alpha <: L$, as the type of the string f. Using α, it splits the type of pattern L, marking α as definitely present ($\downarrow$) whereas $L \cap \overline{\alpha}$ remains possibly absent ($\circ$). Therefore, an `o[f]` expression in the true branch has exactly the type S.

Soundness and Mutable State

We prove standard progress and preservation theorems for λ_S^{ob}. The proofs are over an extended semantics with mutable references, reference types, and store typings. This paper omits references for clarity, but we account for state in our proofs and prototype implementation.

Theorem 1 (Preservation) *If $\Sigma \vdash \sigma$, $\cdot \vdash e : T$, and $\sigma e \to \sigma' e'$, then there exists a Σ', such that $\Sigma \subseteq \Sigma'$, $\Sigma' \vdash \sigma'$, and $\Sigma'; \cdot \vdash e' : T$.*

Theorem 2 (Progress) *$\Sigma \vdash \sigma$ and $\cdot \vdash e : T$ then either $e \in v$ or there exist σ' and e' such that $\sigma e \to \sigma' e'$.*

[8] This single typing rule is adequate for type-checking, but the proof of preservation requires auxiliary rules in the style of Typed Scheme [25].

7 Implementation

We implement fluid object types for our existing prototype type-checker for JavaScript. This type-checker accounts for other facets of JavaScript using techniques described in our earlier work [11,12]. The implementation directly encodes the type system presented in this paper. We achieve this close correspondence because the type-checker does not work with JavaScript directly. A front-end desugars JavaScript to a core calculus [11] that is closely related to λ_S^{ob} (section 9). Earlier, the type-checker tackled trivial object-oriented programs using recursive record types. We fully replace these with fluid object types.

Pattern Representations We use two representations for string patterns: finite sets of strings and finite automata. For finite automata, we use the representation and decision procedure of Hooimeijer and Weimer [16]. Their implementation is fast and based on mechanically proven principles. A thin wrapper transparently converts patterns represented as sets to equivalent finite automata when necessary. The representation of patterns is thus fully abstract to the type-checker.

Type-Checking Experiments We type-check the λ_S^{ob} examples in this paper, various other λ_S^{ob} benchmarks, and $7,000$ lines of JavaScript code. The JavaScript code we type-check consist of two suites of programs:

- 11 Google Chrome Experiments [9] and Google Gadgets [10] [21], and
- The ADsafe runtime [11] and various sample ADsafe widgets [20].

Type-checking JavaScript involves addressing various issues that are orthogonal to typing objects, which is the subject of this paper. In addition, typing existing untyped code requires some refactorings. We discuss these in detail in our earlier work [20,21]. We make two observations about typing objects in JavaScript.

- JavaScript programs heavily manipulate the DOM, which is a massive, object-oriented API with functions and objects for manipulating HTML, CSS, local storage, the canvas, etc. These APIs are specified in various IDL files, [12] which are a type-like interface definitions. We process $4,000$ lines of IDL to build a type environment of object types for Web applications.
- In an earlier paper [20], we present a type-based approach for language-based Web sandboxes and apply our technique to partially verify the ADsafe sandbox. Web sandboxes employ a combination of static and dynamic checks that are notoriously difficult to write. The type-checker in that paper, which used a rudimentary form of fluid object types, found 10 bugs in the ADsafe runtime. Three of these were security bugs that are discussed in that paper. However, there were still portions of ADsafe we could not type-check. The fluid object types presented in this paper can type-check those.

[9] `www.chromexperiments.com`
[10] `desktop.google.com`
[11] `www.adsafe.org`
[12] `www.w3.org/TR/WebIDL`

– The aforementioned paper also notes that other JavaScript security frameworks employ more sophisticated reasoning with string patterns. For example, Caja uses the field named `"foo_w__"` to store a flag that determines if the `"foo"` field is writable. We can easily express this and the 8 other patterns used by Caja using fluid object types:

$$\{\texttt{.*\_w\_\_}^{\circ} : \mathsf{Bool}, \overline{\texttt{.*\_w\_\_}}^{\circ} : T\}$$

Performance Subtyping patterns (figure 4) and type well-formedness (figure 2) suggest two possible performance bottlenecks: both require pairwise pattern intersection checks and subtyping requires pattern inclusion checks, which are reducible to pattern intersection. Finite automata intersection takes exponential time and it is easy to construct synthetic examples that demonstrate the worst-case time complexity of subtyping. Fortunately, we have reason to believe that realistic programs work with a small collection of "interesting" patterns; most patterns are constant strings. The only interesting pattern in the JavaScript standard library is for array indices. ADsafe and Google Caja, discussed above, have two and eight patterns respectively. We test our implementation with patterns from the aforementioned systems. On these patterns, our type-checker is fast; it runs various benchmarks in approximately one second on an Intel Core i5 processor.

8 Conclusion

We present *fluid object types* to type-check lightweight, dynamic objects. Fluid object types feature two novelties. First, they generalize field names to *field patterns* to describe collections of fields. Second, they use *presence annotations* to describe the position of fields on the inheritance chain. We present a subtyping algorithm and type soundness proofs for our type system, which includes equirecursive types, bounded quantification, and mutable references; these are essential features of many realistic object type systems.

We demonstrate that our type system can type-check both standard uses of objects (i.e., classes and prototypes) and also more esoteric uses of objects in scripting languages, such as dictionaries and naming conventions for private fields. Finally, we implement fluid object types in an experimental type-checker for JavaScript.

Future Directions In our current type system, object update and deletion are invariant. This restriction is easy to address for functional objects, but much harder in the presence of state. Our presentation in this paper omits state for clarity, although the proofs in the appendices include mutable references. A meaningful account of strong updates must be presented with state.

Useful extensions to λ_S^{ob} include getters, setters, and object proxies. Proxies "virtualize" various object operations [4]. For example, a proxy can define an function that is called to handle field lookups and thus simulate an object with

an unbounded collection of fields. The field patterns of fluid object types are well-suited to type-checking proxies and proxy-like constructs.[13]

9 Related Work

Our work builds on the long history of semantics and types for objects and recent work on semantics and types for scripting languages.

Semantics of Scripting Languages There are various semantics for scripting languages [10,11,18,22] that model each language in detail. This paper focuses on type-checking a core calculus of objects and elides many features and details of individual scripting languages. Our semantics also abstracts the plethora of string-matching operations available in real scripting languages into a single pattern matching construct, and makes object-reflection manifest for type-checking.

Extensible Records The representation of objects in λ_S^{ob} is derived from extensible records, surveyed by Fisher and Mitchell [8] and Bruce, et al. [6]. Unlike the surveyed languages, field names in λ_S^{ob} are first-class strings; λ_S^{ob} includes operators to enumerate over fields and test for the presence of fields, since these are typical of scripting languages. Our fluid object types account for these features using presence-annotations and field-name patterns that are related to types for scripting languages, discussed below.

Types and Contracts for Untyped Languages There are various type systems retrofitted onto untyped languages. We discuss those that support objects.

Strongtalk [5] is a typed dialect of Smalltalk that uses *protocols* to describe objects. Field patterns can describe more ad hoc objects than the protocols of Strongtalk, which are a finite enumeration of fixed names. Strongtalk protocols may include a brand; they are thus a mix of nominal and structural types. In contrast, fluid object types are purely structural, though we do not anticipate any difficulty incorporating brands.

Our work shares features with various JavaScript type systems. In the type system of Anderson, et al. [3], objects' fields may be potentially present; it employs strong updates to turn these into definitely present fields. Recency types [14] support field type-changes during initialization. Zhao's type system [27] also allows unrestricted object extension, but omits prototypes. In contrast to these works, our fluid object types do not support strong updates. We instead allow possible-absent fields to turn into definitely-present fields via reflection, which they do not support. Strong updates would be fruitful to type initialization patterns. In these type systems, field names are first-order labels. Thiemann's [24] type system for JavaScript allows first-class strings as field names, which we generalize to field patterns. In addition, we allow inheritance chains to be precisely typed by distinguishing possibly-inherited fields from fields that are

[13] Python, Ruby, Lua, and ECMAScript 6 (the upcoming JavaScript standard) all have proxies or proxy-like constructs.

immediately present. These are useful for typing features of scripting languages (section 3).

RPython [2] compiles Python programs to efficient byte-code for the CLI and the JVM. Dynamically updating Python objects cannot be compiled. Thus, RPython stages evaluation into an interpreted initialization phase, where dynamic features are permitted, and a compiled running phase, where dynamic features are disallowed. Our types give guarantees without staging restrictions.

DRuby [9] does not account for reflection in general. However, as a special case, An, et al. [1] build a type-checker for Rails-based Web applications that partially-evaluates dynamic operations, producing a program that DRuby can verify. In contrast, our types tackle reflection directly.

System D [7] uses dependent refinements to type dynamic dictionaries. Fluid object types can type-check dictionaries as a special case of objects; we also account for inheritance, recursive objects and imperative state. System D accounts for richer control-dependent type reasoning than the single if-splitting rule presented in this paper. However, our earlier work [12] accounts for control and and state based type reasoning using other techniques. The authors of System D suggest integrating a string decision procedure to reason about dictionary keys. We use DPRLE [16] to support exactly this style of reasoning.

Heidegger, et al. [13] present *dynamically*-checked contracts for JavaScript that use regular expressions to describe objects. Our implementation uses regular expressions for *static* checking.

Regular Expression Types Regular tree types and regular expressions can describe the structure of XML documents (e.g., XDuce [17]) and strings (e.g., XPerl [23]). These languages verify XML-manipulating and string-processing programs. Our type system uses patterns not to describe trees of objects like XDuce, but to describe objects' field names. Our string patterns thus allow individual objects to have semi-determinate shapes. Like XPerl, field names are simply strings, but our strings are used to index objects, which are not modeled by XPerl.

References

1. An, J.D., Chaudhuri, A., Foster, J.S.: Static typing for Ruby on Rails. In: IEEE International Symposium on Automated Software Engineering (2009)
2. Ancona, D., Ancona, M., Cuni, A., Matsakis, N.D.: RPython: a step towards reconciling dynamically and statically typed OO languages. In: ACM SIGPLAN Dynamic Languages Symposium (2007)
3. Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for JavaScript. In: European Conference on Object-Oriented Programming (2005)
4. Austin, T.H., Disney, T., Flanagan, C.: Virtual values for language extension. In: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (2011)
5. Bracha, G., Griswold, D.: Strongtalk: Typechecking Smalltalk in a production environment. In: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (1993)

6. Bruce, K.B., Cardelli, L., Pierce, B.C.: Comparing object encodings. Information and Computation 155(1–2) (1999)
7. Chugh, R., Rondon, P.M., Jhala, R.: Nested refinements for dynamic languages. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2012)
8. Fisher, K., Mitchell, J.C.: The development of type systems for object-oriented languages. Theory and Practice of Object Systems 1 (1995)
9. Furr, M., An, J.D., Foster, J.S., Hicks, M.: Static type inference for Ruby. In: ACM Symposium on Applied Computing (2009)
10. Furr, M., An, J.D.A., Foster, J.S., Hicks, M.: The Ruby Intermediate Language. In: ACM SIGPLAN Dynamic Languages Symposium (2009)
11. Guha, A., Saftoiu, C., Krishnamurthi, S.: The essence of JavaScript. In: European Conference on Object-Oriented Programming (2010)
12. Guha, A., Saftoiu, C., Krishnamurthi, S.: Typing local control and state using flow analysis. In: European Symposium on Programming (2011)
13. Heidegger, P., Bieniusa, A., Thiemann, P.: Access permission contracts. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2012)
14. Heidegger, P., Thiemann, P.: Recency types for dynamically-typed, object-based languages: Strong updates for JavaScript. In: ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages (2009)
15. Hooimeijer, P., Veanes, M.: An evaluation of automata algorithms for string analysis. In: International Conference on Verification, Model Checking, and Abstract Interpretation (2011)
16. Hooimeijer, P., Weimer, W.: A decision procedure for subset constraints over regular languages. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (2009)
17. Hosoya, H., Vouillon, J., Pierce, B.C.: Regular expression types for XML. ACM Transactions on Programming Languages and Systems 27 (2005)
18. Maffeis, S., Mitchell, J.C., Taly, A.: An operational semantics for JavaScript. In: Asian Symposium on Programming Languages and Systems (2008)
19. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
20. Politz, J.G., Eliopoulos, S.A., Guha, A., Krishnamurthi, S.: Type-based verification of JavaScript sandboxing. In: USENIX Security Symposium (2011)
21. Saftoiu, C.: Runtime type-discovery for javascript. Tech. Rep. CS-10-05, Brown University (2010)
22. Smeding, G.J.: An executable operational semantics for Python. Master's thesis, Utrecht University (2009)
23. Tabuchi, N., Sumii, E., Yonezawa, A.: Regular expression types for strings in a text processing language. Electronic Notes in Theoretical Computer Science 75 (2003)
24. Thiemann, P.: Towards a type system for analyzing JavaScript programs. In: European Symposium on Programming (2005)
25. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of Typed Scheme. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2008)
26. Tobin-Hochstadt, S., Felleisen, M.: Logical types for untyped languages. In: ACM SIGPLAN International Conference on Functional Programming (2010)
27. Zhao, T.: Type inference for scripting languages with implicit extension. In: ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages (2010)

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 **func**(*x*:*T*) { *e* } instead of **func**(*x*:*T*) . *e*.

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

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

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

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

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:*

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

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:*

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

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

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

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:*

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

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 (Γ, 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 $(\Gamma, S, T) \in TR(\mathcal{ST}(R))$. By definition of TR, there exists a U such that $(\Gamma, S, U), (\Gamma, U, T) \in \mathcal{ST}(R)$. We will show that $(\Gamma, S, T) \in \mathcal{ST}(TR(R))$, 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 $(\Gamma, S, U), (\Gamma, U, T) \in \mathcal{ST}(R)$, 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, $(\Gamma, S, U) \in \mathcal{ST}(R)$, 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 $(\Gamma, \mathit{inherit}(S, L_i), U_i) \in R$.
  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** $U = \mathbf{Ref}\ U'$**.** Only case S-Ref applies, thus $S = \mathsf{Ref}\ S'$ and $T = \mathsf{Ref}\ T'$, with $(S', U'), (U', S'), (U', T'), (T', U') \in R$. By definition of TR, $(S', T'), (T', S') \in TR(R)$. Thus, $(\mathsf{Ref}\ S', \mathsf{Ref}\ T') \in \mathcal{ST}(TR(R))$.

– **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 $reachables_{\mathcal{ST}}(S, T)$ is finite [19, Proposition 21.9.11]. Thus, the algorithm $gfp_{\mathcal{ST}}$ [19, Definition 21.5.5] terminates [19, Theorem 21.5.12]. ■

Lemma 10 *For all Γ, T, L, M, $inherit_\Gamma(T, L) \sqcup inherit_\Gamma(T, M) = inherit_\Gamma(T, L \sqcup M)$.*

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 $\Gamma \vdash S'_i <: T'_j$.

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

Lemma 12 *For all Γ, S, T, L_Q, M_Q, if:*

H1. $\Gamma \vdash S <: T$,
H2. $\Gamma \vdash L_Q \subseteq M_Q$,
H3. $inherit_\Gamma(S, L_Q) = S'$, *and*
H4. $inherit_\Gamma(T, M_Q) = T'$,

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

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

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

We thus have available the hypotheses of S-Object:

I1. $\forall i, j.L_i \cap M_j \neq \emptyset \Rightarrow p_i <: q_j \wedge \Gamma \vdash S'_i <: T'_j$,
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 $q_j =\uparrow \wedge \Gamma \vdash inherit(S, M_j) <: T'_j$, 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:

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

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. $\forall L'_Q, M'_Q, S_P, T_P.|S_P| < |S| \wedge |T_P| < |T| \wedge \Gamma \vdash L'_Q \subseteq M'_Q \wedge \Gamma \vdash S_P <: T_P \Rightarrow \big(inherit_\Gamma(S_P, L'_Q) = S'_P \wedge inherit_\Gamma(T_P, M'_Q) = T'_P \Rightarrow \Gamma \vdash S'_P <: T'_P\big)$.

The goal thus reduces to:

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

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

Using lemma 10, we rewrite the goal to:

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

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

a. $\Gamma \vdash \bigsqcup\{S'_i \mid \Gamma \vdash L_Q \cap L_i \neq \emptyset\} <: \bigsqcup\{T'_j \mid \Gamma \vdash M_Q \cap M_j \neq \emptyset\}$
   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 $\Gamma \vdash \bigsqcup\{S'_i \mid \Gamma \vdash L_Q \backslash L_A \cap L_i \neq \emptyset\} <: \bigsqcup\{T'_j \mid \Gamma \vdash M_Q \backslash M_A \cap M_j \neq \emptyset\}$. Lemma 11 now applies.

b. $\Gamma \vdash \mathit{inherit}(S_p, \mathcal{L} \cap \overline{\bigcup \mathcal{N}}) <: \mathit{inherit}(T_p, \mathcal{M})$
   By induction (HInd). The following cases allow us to apply HInd.
   - $|S_P| < |S|$ and $|T_P| < |T|$ are trivial.
   - We show that $\Gamma \vdash S_P <: T_P$. By H5 and H6, `"parent"` $\in L_P, M_P$ thus $L_P \cap M_P \neq \emptyset$. Therefore, $\Gamma \vdash \mathsf{Ref}\ S_P <: \mathsf{Ref}\ T_P$ by I1. By (S-Ref), it follows that $\Gamma \vdash S_P <: T_P$.
   - We show that $\Gamma \vdash \mathcal{L} \cap \overline{\bigcup \mathcal{N}} \subseteq \mathcal{M}$. It is sufficient to show that $\forall x. \Gamma \vdash x \in \mathcal{L} \cap \overline{\bigcup \mathcal{N}} \Rightarrow \Gamma \vdash x \in \mathcal{M}$.
     By definition of $\mathcal{L}$, $x \in L_Q$, thus by (H2), $x \in M_Q$.
     By (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. $\Gamma \vdash \mathit{inherit}(S_p, \mathcal{L} \cap \bigcup \mathcal{N}) <: \bigsqcup\{T'_j \mid \Gamma \vdash M_Q \cap M_j \neq \emptyset\}$.
   Rewrite the left-hand side by expanding the definition of $\mathcal{N}$, distributing the intersection over the union, and applying lemma 10:

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

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

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

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

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

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:

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

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 $\exists i.$`"parent"` $\in M_i$ then $q_i =^{\downarrow}$. But by I2, `"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. $\exists L_P.$`"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 $\neg\exists M_P.$`"parent"` $\in M_P$, and
H8. $M_Q \subseteq \bigcup_i^{1\cdots m} \{M_i \mid q_i \neq^\circ\}$.

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

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

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

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

to:

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

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 $F_i = \textbf{abs}$ 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

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

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

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

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

Fig. 6. Auxiliary Typing Rules for If-Splitting

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

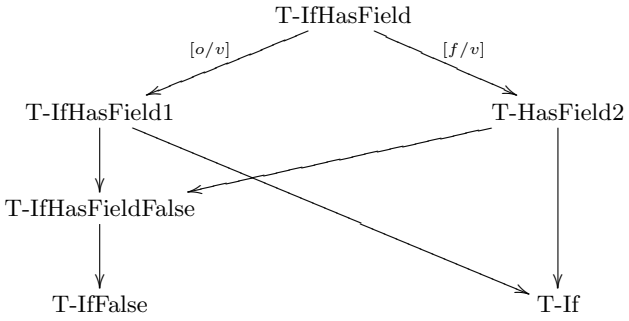

Fig. 7. Usage of Auxiliary Typing Rules by Substitution

- $\{L_1^{p_1} : S_1 \cdots L_m^{p_m} : S_m\}$ *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* $\{str_1^{\downarrow} : U_1 \cdots str_n^{\downarrow} : U_n, \overline{\{str_1 \cdots str_n\}} : \mathbf{abs}\} <: \{L_1^{p_m} : S_1 \cdots L_m^{p_m} : S_m\}$,
- $\mathsf{Ref}\ S$, *then* $v = loc$ *and* $\Sigma(loc) <: S$,
- $S \to T$, *then* $v = \mathbf{func}(x)\ \{\ e\ \}$,
- 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* $\Sigma; \Gamma \vdash e_1, e_2 <: \mathsf{Str}$ *and* $\Gamma \vdash T <: \mathsf{Str}$.
- $\Sigma; \Gamma \vdash \mathbf{fix}\ (f : S).e : T$ *then* $\Sigma; \Gamma, f : S \vdash e <: S$ *and* $\Gamma \vdash S <: T$.
- $\Sigma; \Gamma \vdash \mathbf{ref}\ e : T$ *then* $\Sigma; \Gamma \vdash e : S$ *and* $\mathsf{Ref}\ S = T$
- $\Sigma; \Gamma \vdash l : T$ *then* $\Sigma(l) = T$
- $\Sigma; \Gamma \vdash \mathbf{deref}\ e : T$, *then* $\Sigma; \Gamma \vdash e : \mathsf{Ref}\ S$ *with* $S <: T$,
- $\Sigma; \Gamma \vdash e_1 = e_2 : T$, *then* $\Sigma; \Gamma \vdash e_1 : \mathsf{Ref}\ S$, $\Sigma; \Gamma \vdash e_2 : U$, $U <: S$, *and* $\mathsf{Ref}\ S <: T$,
- $\Sigma; \Gamma \vdash e_f(e \cdots) : T$, *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* $\Gamma \vdash \{L^p : S \cdots, L_A : \mathbf{abs}\} <: T$.
- $\Sigma; \Gamma \vdash \mathbf{delete}\ e_o[e_f] : T$, *then* $\Sigma; \Gamma \vdash e_o : \{L_1^{p_1} : S_1 \cdots\}$, $\Sigma; \Gamma \vdash e_f : L$, $\forall L \cap L_i \neq \emptyset.F_i \neq T^{\downarrow}$, $\Gamma \vdash \{L_1^{p_1} : S_1 \cdots\} <: T$
- $\Sigma; \Gamma \vdash e_1\ \mathbf{hasfield}\ e_2 : \mathsf{Bool}$, *then* $\Sigma; \Gamma \vdash e_1 : \{L_1 : F_1 \cdots L_n : F_n\}$, $\Sigma; \Gamma \vdash e_2 : L$.
- $\Sigma; \Gamma \vdash e\ \mathbf{matches}\ P : \mathsf{Bool}$, *then* $\Sigma; \Gamma \vdash e : L$.
- $\Sigma; \Gamma \vdash \mathbf{if}\ (v_1)\ \{\ e_2\ \}\ \mathbf{else}\ \{\ e_3\ \} : T$, *then* $\Sigma; \Gamma \vdash v_1 : \mathsf{Bool}$, $\Sigma; \Gamma \vdash e_2 : T$, *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:

- $\sigma_1 E\langle(\textsf{func}\ (x : S')\ \{\ e\ \})(v)\rangle \to \sigma_1 E\langle e[x/v]\rangle$.
  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), $\Sigma_1; \cdot \vdash e[x/v] : T$.
- $\sigma_1 E\langle\textsf{fix}\ (x : S.\ e\rangle \to \sigma_1 E\langle e[x/\textsf{fix}\ (x : S).e]\rangle$.
  By inversion, $\Sigma_1; \cdot \vdash e : S$, $\Sigma_1; x : S \vdash e : S$, and $\cdot \vdash S <: T$. By substitution (lemma 17), $\Sigma_1; \cdot \vdash e[x/\textsf{fix}\ (x:S).\ \mathsf{e}] : T$.
- $\sigma E\langle(\lambda\alpha <: S.e)(U)\rangle \to \sigma E\langle e[\alpha/U]\rangle$
  By type substitution (lemma 16).
- $\sigma E\langle\textsf{ref}\ v\rangle \to \sigma, (l, v)E\langle l\rangle$ 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, $\Sigma_2; \cdot \vdash l : \mathsf{Ref}\ S$.
- $\sigma E\langle\textsf{deref}\ \mathsf{l}\rangle \to \sigma E\langle\sigma(l)\rangle$
  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, $\Sigma; \Gamma \vdash l : T$.
- $\sigma E\langle\{\ \cdots str\mathtt{:}\ v \cdots\ \}[str]\rangle \to \sigma E\langle v\rangle$. 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], $inherit.(\{\cdots str^{\downarrow} : T' \cdots\}, str) = T'$.
- $\sigma E\langle\{\ \cdots\ \texttt{"parent"}\mathtt{:}\ l\ \}[str]\rangle \to \sigma E\langle(\textsf{deref}\ l)[str]\rangle$, 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 $inherit$, $inherit.(\{\cdots \texttt{"parent"}^{\downarrow} : \mathsf{Ref}\ S_P\}, str) = inherit...(S_P, str)$, which is a subtype of T.
  Type right-hand side with T-Sub and T-GetField, using $\Sigma_1; \cdot \vdash str : str$, $inherit...(S_P, str) <: T$, and $\Sigma_1; \cdot \vdash (\textsf{deref}\ l) : S_P$. This holds since $\Sigma_1 : \cdot \vdash l : \mathsf{Ref}\ S_P$ above.
- $\sigma E\langle\{\ \cdots str : v \cdots\ \}[str\ =\ v']\rangle \to \sigma E\langle\{\ \cdots str : v' \cdots\ \}\rangle$
  By inversion (T-Update), $\Sigma; \Gamma \vdash \{\cdots\} : \{L : S \cdots\}$, $\Gamma \vdash \{L : S \cdots\} <: T$, and $\Sigma; \Gamma \vdash v' : U'$. By inversion (T-Object), $\Sigma; \Gamma \vdash \{\cdots str\mathtt{:}v \cdots\} : \{\cdots str : U \cdots\}$ 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, $\Gamma \vdash \{\cdots str : U' \cdots\} <: \{\cdots str : U \cdots\} <: T$.
- $\sigma E\langle\{\ \cdots\ \}[str\ =\ v']\rangle \to \sigma E\langle\{\ \cdots\ \}\rangle$ where $str \notin \cdots$
  By inversion of T-Update, $\Sigma; \Gamma \vdash \{\cdots str\mathtt{:}v \cdots\} : S$ and $\Gamma \vdash S <: T$.
- $\sigma E\langle\textsf{delete}\ \{\ \cdots str : v \cdots\ \}[str]\rangle \to \sigma E\langle\{\ \cdots\cdots\ \}\rangle$
  Similar to to E-UpdateField case above.
- $\sigma E\langle\textsf{delete}\ \{\ \cdots\ \}[str]\rangle \to \sigma E\langle\textsf{delete}\ \{\ \cdots\ \}\rangle$ where $str \notin \cdots$
  Similar to to E-UpdateField case above.
- $\sigma E\langle\textsf{fieldin}\ \{\ s\mathtt{:}v\mathtt{,}rest \cdots\ \}\ \textsf{init}\ v_{acc}\ \textsf{do}\ v_f\rangle \to \sigma E\langle\textsf{fieldin}\ \{\ str_2 : v_2\ \cdots\ \}\ \textsf{init}\ v_f(str_1)(v_{acc})\ \textsf{do}\ v_f\rangle$
  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.

- $\sigma E\langle$**fieldin** `{` s`:`v`}` **init** v_{acc} **do** $v_f\rangle \to \sigma E\langle v_f$`(`$s$`)(`$v_{acc}$`)`$\rangle$
  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.)
- $\sigma E\langle$**if**`(`**true**`) {` e_1 `}` **else** `{` e_2 `}`$\rangle \to \sigma E\langle e_1\rangle$
- $\sigma E\langle$**if**`(`**false**`) {` e_1 `}` **else** `{` e_2 `}`$\rangle \to \sigma E\langle e_2\rangle$
- $\sigma E\langle$`{` $\cdots str$`:`$v \cdots$ `}` **hasfield** $str\rangle \to \sigma E\langle$**true**$\rangle$
- $\sigma E\langle$`{` $\cdots$ `}` **hasfield** $str\rangle \to \sigma E\langle$**false**$\rangle$ where $str \notin \cdots$
- $\sigma E\langle str$ **matches** $P\rangle \to \sigma E\langle$**true**$\rangle$
- $\sigma E\langle str$ **matches** $P\rangle \to \sigma E\langle$**false**$\rangle$
- $\sigma E\langle str_1$ `+` $str_2\rangle \to \sigma E\langle str_1 str_2\rangle$

■

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 $e = E\langle$**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`(`v_2`)`, **ref** v, **deref** v, v_1 `=` v_2, and **if** `(`v_1`) {` e_2 `}` **else** `{` e_3 `}` are routine.
- Consider $ae = v_1$`[`v_2`]`. 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 $ae = v_1\texttt{[}v_2\ \texttt{=}\ v_3\texttt{]}$. 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 $ae = \textsf{delete}\ v_1\texttt{[}v_2\texttt{]}$. 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 $ae = v_1\ \textsf{hasfield}\ v_2$. 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 $ae = v\ \textsf{matches}\ P$. By inversion and canonical forms, $v = str$. Thus either E-Matches or E-NoMatch apply.
– Consider $ae = \textsf{fieldin}\ \texttt{\{}\ \ s_1 : v_1\texttt{,}\ \ s_2 : v_2\ \ \cdots\texttt{\}}\ \textsf{init}\ v_{acc}\ \textsf{do}\ \ v_f$. 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 $ae = v_1\texttt{+}\ v_2$. By inversion and canonical forms, $v_1 = str_1$ and $v_2 = str_2$. Thus E-Str+ applies.
– Consider $ae = \textsf{fix}\ \texttt{(}f\texttt{:}S\texttt{).}\ \texttt{e}$. E-Fix applies trivially.