TeJaS: Retrofitting Type Systems for JavaScript

Benjamin S. Lerner
Brown University
blerner@cs.brown.edu

Joe Gibbs Politz
Brown University
joe@cs.brown.edu

Arjun Guha
UMass Amherst
arjun@cs.umass.edu

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Abstract

JavaScript programs vary widely in functionality, complexity, and use, and analyses of these programs must accommodate such variations. Type-based analyses are typically the simplest such analyses, but due to the language's subtle idioms and many application-specific needs—such as ensuring general-purpose type correctness, security properties, or proper library usage—we have found that a single type system does not suffice for all purposes. However, these varied uses still share many reusable common elements.

In this paper we present TeJaS, a framework for building type systems for JavaScript. TeJaS has been engineered modularly to encourage experimentation. Its initial type environment is reified, to admit easy modeling of the various execution contexts of JavaScript programs, and its type language and typing rules are extensible, to enable variations of the type system to be constructed easily.

The paper presents the base TeJaS type system, which performs traditional type-checking for JavaScript. Because JavaScript demands complex types, we explain several design decisions to improve user ergonomics. We then describe TeJaS's modular structure, and illustrate it by reconstructing the essence of a very different type system for JavaScript. Systems built from TeJaS have been applied to several real-world, third-party JavaScript programs.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

Keywords JavaScript, type systems

DLS '13, October 28, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2508168.2508170

1. Retrofitting Type Systems

There is a venerable line of research on *retrofitting* type systems onto previously "untyped" languages, with Morris's seminal dissertation [11] an early example. In practice, these languages are not actually free of types; rather, all type discrimination is performed by primitive operations at run-time. A common practice for retrofitting type systems has been to identify the resulting set of run-time errors and try to eliminate them statically. For instance, Smalltalk type systems [1, 22] focused on eliminating message-not-found errors, and Soft Scheme [25] addressed the wide range of Scheme dynamic errors. Identifying run-time errors and catching them statically can be viewed as the design principle for retrofitted type systems.

Unfortunately, this principle is rather difficult to apply to JavaScript (JS), because there are so few actual run-time errors.[1] In this, JS inherits the forgiving mentality of the browser environment where it was born; instead of flagging errors and halting execution, it simply continues running. As a result, operations that in other languages might reasonably be expected to generate errors do not in JS: subtracting one string from another, accessing an array outside its bounds, reading and writing non-existent fields, calling functions with the wrong number of arguments, etc.

Obviously, every one of these operations has a well-defined semantics in JS (usually, ironically, involving the value `undefined`). Therefore, it becomes a *purpose-specific judgment call* whether or not these should be considered static type errors. If the purpose is to enforce a strict, Java-like programming style, they should probably *all* be considered errors. If, in contrast, the goal is to check existing code for violations that are oblivious to the precise behavior of these operations, then—to minimize the amount of refactoring needed to pass type-checking—we might want to block *none* of them. In practice, we have found our needs (Section 2) to fall somewhere in between. For these reasons, it does not make sense to have just one canonical type system

[1] Expert JavaScript readers are encouraged to try to enumerate them all.

for JS; instead, we need a framework that makes it easy to quickly construct a new one for a given need.

Type Systems for JavaScript In recent years we have built several type systems to statically analyze JS programs for different purposes (summarized in Section 2). Our papers on these systems have focused on the novel features distinguishing the variants from the (implicit) base system, without explaining the base system in any detail. As a result of building these systems we have also created a parameterized framework, called TeJaS, to simplify the construction of new ***t***ype systems for ***J***ava***S***cript. Finally, we have also had to address several ergonomic details necessary to make our system usable. This is the first paper to present the base system as a whole, and the latter two aspects for the first time. Specifically:

- we present the base type system;
- we explain several forms of syntactic sugar to make types easier to write;
- we explain the module-level architecture of our implementation and how it aids developing the aforementioned variant systems;
- we explain how external type environments allow for varied guarantees even from a single type system; and,
- we use some of these features to create a new type system that mimics an essential feature of TypeScript.

Our infrastructure is publicly available at https://github.com/brownplt/TeJaS.

The rest of this paper is organized as follows. Section 2 describes the various systems we have analyzed using variations on the type system presented here. Section 3 presents the type language itself. Section 4 explores several forms of type-level syntactic sugar, and how they help capture central idioms of JS. Section 5 introduces the architecture of our system, illustrating it via the kernels of three of the type-system variants described below. Section 6 describes our use of reified environment files. Section 7 presents related work, and Section 8 touches on several avenues for future improvements to our system.

2. Theme and Variations: Uses of our Type Systems

This paper focuses on the engineering details of the TeJaS type system, but to place our design in context we briefly describe several projects that have already used it. These have resulted in papers of their own, so we give only an overview of each problem and the impact each project had on the design of the TeJaS system:

ADsafety Politz et al. [15] analyze ADsafe (http://www.adsafe.org), a sandbox for containing untrusted third-party scripts in web pages. The sandbox consists of two parts: an offline tool that searches for banned identifiers and expressions in the target script, and an online library implementing a reference monitor for the protected resources of the page. The intended behavior is that together, scripts passing the offline checks could not access protected resources either directly or indirectly by subverting the monitor.

To evaluate their system, the authors summarized the offline tool's behavior as a concise *type*; extensive testing confirmed that scripts passing the tool did indeed have the type, and conversely that scripts that failed to typecheck could exploit flaws (that have since been fixed) in the tool. Moreover, they typechecked the online monitor and thereby showed that when given well-typed arguments, it cannot leak protected resources. This both demonstrated the type language's expressiveness, in accurately matching the offline tool's behavior, and the type checker's efficacy, as it could successfully check a large program with subtle type invariants.

Violations of Private Browsing in Extensions Lerner et al. [9] examine Mozilla Firefox extensions for potential violations of private browsing mode. In this mode, browsers should not leave on disk any data that relates to the user's private browsing session. Browsers have been extensively audited to implement this properly. However, third-party extensions may not be so circumspect, and no tool support currently exists to guide developers (or users) when extensions behave poorly; instead, Mozilla developers manually audit extensions before offering them for widespread availability.

The authors evaluated their system by examining twelve Mozilla-audited extensions, lightly annotated them (but did not refactor their code), and found four extensions that violated the private-browsing invariant. This analysis utilized essentially the same type system as ADsafety, and carefully managed the type environment to capture the *modal* nature of private browsing: writing to disk is acceptable extension behavior while in non-private mode. The appropriate type environment here ascribed a particular flag the type True, rather than Bool, and allowed the type system to perform a form of dead-code elimination during typechecking.

JQuery Lerner et al. [8] examine web programs written using the jQuery library. JQuery provides a higher-level API for selecting a set of document elements, navigating from a current set of elements to a new set, and then uniformly manipulating those selected elements in various ways. Several of jQuery's APIs behave subtly differently depending on how many nodes they are given; additionally not all APIs are applicable to all types of elements. Yet jQuery APIs are designed to stifle these errors, making it difficult to detect why a program behaves unexpectedly.

To analyze such queries, the authors extended the TeJaS type system with constructions for CSS selectors and for "local structures" within a document. Moreover, they added new *kinds* to the type system to represent the sizes of jQuery collections, and thereby account for the varying behaviors of the jQuery APIs. They evaluated their system by typechecking several examples culled from online and printed tutorials, and showed that the system correctly detects when queries match the declared local document structures.

$$
\begin{array}{rcl}
\alpha \in \textit{Type identifiers} & & \\
S, T \in & \textit{Typ} ::= & \mathsf{Num} \mid r \mid \mathsf{True} \mid \mathsf{False} \mid \mathsf{Undef} \mid \mathsf{Null} \\
& & \mid \mathsf{Ref}\, T \mid \mathsf{Src}\, T \mid \mathsf{Sink}\, T \mid \mu\alpha :: K \,.\, T \mid \alpha \\
& & \mid \forall \alpha <: T \,.\, T \mid T[T, \cdots, T] \\
& & \mid \Lambda\alpha :: K \,.\, T \mid T\langle T, \cdots, T\rangle \\
& & \mid [T_{self}]\, T_1 \times \cdots \times T_n \to T_{ret} \\
& & \mid [T_{self}]\, T_1 \times \cdots \times T_n \times T_{rest} \ldots \to T_{ret} \\
& & \mid \top \mid \bot \mid T \cup T \mid T \cap T \mid \{\!\{ f \ldots \}\!\} \\
r \in \textit{Regex} & \ni & \texttt{"\_\_proto\_\_"} \mid \texttt{"\_\_code\_\_"} \mid \epsilon \mid \cdots \\
K \in \textit{Kinds} & ::= & \star \mid K \to K \\
f \in \textit{Fields} & ::= & r : \hat{}\, T \mid r : ?\, T \mid r : !\, T \\
& & \mid r : \mathsf{Absent} \mid r : \mathsf{Hidden} \\
\Gamma \in & \textit{Env} ::= & \cdot \mid \Gamma, x : T \mid \Gamma, \alpha = T \mid \Gamma, \alpha <: T \\
& & \mid \Gamma, \alpha :: K \mid \Gamma, l =_L T
\end{array}
$$

Figure 1: Abstract syntax of base type language for TeJaS

Since each of these projects largely uses the same base type system, practical software engineering considerations led us to re-architect our system: now, each variation can in fact reuse the implementation of the base type system, without having to directly modify it. We first present the base type system, and then the architecture that makes supporting these and other variations possible. Readers interested strictly in modularity may find it possible to only skim the presentation of base types.

3. The Base Type Language

Our base type language, which to us represents the essence of typing JavaScript, is presented in Fig. 1. Many features are largely straightforward. It contains primitive types for numeric, `null` and `undefined` values, as well as top and bottom types. Function types may be fixed-arity or variadic, with distinguished receiver parameters (the type of `this` within the function). (For brevity, when the receiver type is irrelevant, it may be elided.) To describe mutability precisely, our language supports read/write, read-only and write-only reference cells. Additionally, our language supports type-level functions and (equi-)recursive types.

Typographic conventions The abstract syntax of types (as presented in Fig. 1) are written in sans serif. JS program fragments, and the concrete syntax of types (and their syntactic sugar, presented below) are written in `typewriter font` with keyword highlighting. As one final clarification, the abstract syntax of object types uses {{doubled braces}}.

Union, Intersection, and Boolean Types On top of this, we add support for union and intersection types. For instance, `True` and `False` are primitive types in our system, and `Bool` is the union type `(True + False)`. As a more complicated example, the addition operator might have type

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

$$
\textsc{WF-TObject}\ \frac{\Gamma \vdash T_1, \ldots, \Gamma \vdash T_n \qquad \underset{1 \le i \ne j \le n}{\forall} r_i \cap r_j = \emptyset \qquad \left(\bigcup_{1 \le i \le n} r_i\right) = \texttt{/.*/}}{\Gamma \vdash \{\!\{ r_1 : T_1, \cdots, r_n : T_n \}\!\}} \qquad \textsc{WF-TVar}\ \frac{\alpha \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \alpha}
$$

Figure 2: Well-formedness of types. Other rules for other types are straightforward.

$$(\mathsf{Num} \times \mathsf{Num} \to \mathsf{Num}) \cap (\mathsf{Str} \times \top \to \mathsf{Str})$$

which will yield a number when given two numeric arguments, a string when given a string as its first argument, and otherwise will not apply; this is known as *finitary overloading* [21]. (Note that this example type is a *conservative approximation* and does not capture all of JS's behavior; for example, it does not express JS's implicit object-to-string conversions. Perhaps surprisingly, we have found it sufficient to typecheck most well-behaved example programs. However, the whole point of this paper is, you don't have to agree with our decision: you're free to change it!)

Our intersection types are *ordered* and left-biased. For example, we can express the exact type of the logical negation operator as:

$$(\mathsf{True} \to \mathsf{False}) \cap (\mathsf{False} \to \mathsf{True}) \cap (\top \to \mathsf{Bool})$$

When the argument is known to be a specific boolean, we can determine specifically what the result will be; when the argument is any other value, the result is merely boolean.

String and Object Types Because JS makes such extensive use of first-class strings as field names in objects, we need a much richer type than a mere "String" to describe them. Otherwise, the field-accessing operations would be unable to distinguish *which* field is being accessed. However, we still need a decidable type system. Therefore, we replace the inexact "String" type with the set $Regex$ of regular expressions. Using them, string literals have precise singleton types, while finite and co-finite sets of strings (i.e., "one of the strings *A*, *B*, or *C*", or "any string except *D* or *E*") can be expressed exactly. Additionally, *families* of strings can be expressed: for instance, we can express the set "only strings of the form *getX* or *setX*". Finally, we define Str as an alias for the regular expression `/.*/`, i.e., the set of all strings, and ϵ for the empty set of strings.

We make extensive use of these regular expression string types in defining the syntax for object types. For orthogonality, object types describe *immutable records* of fields; we defer the mutability decisions to the reference types described above.[2] Because objects may change over the lifetime of the program, or because multiple different objects may be de-

[2] Also see Section 4.2 for further explanation of this design decision.

scribed by a single type, we need a means to describe the *presence* of fields on an object. We distinguish five cases:

- *aField* `:!` T ⇒ definitely present
- *aField* `:?` T ⇒ possibly present
- *aField* `:^` T ⇒ definitely present either on this object or via the prototype chain
- *aField* : Absent ⇒ definitely not present on this object, may or may not be present via the prototype chain
- *aField* : Hidden ⇒ cannot be accessed

To ensure that typechecking for arbitrary fields is both possible and deterministic, object types must be well-formed: every possible field name must be accounted for exactly once. We formalize the well-formedness of types in Fig. 2.

Presence annotations interact strongly with subtyping relationships (defined in Fig. 7); for example, two object types cannot be subtypes if they assert different fields as necessarily present or hidden.

We distinguish two special field names: all objects can have a `__proto__` field that stores its prototype, while JS functions are modeled as objects with a `__code__` field that stores the function's closure.

We also provide some lightweight, local syntactic sugar to simplify writing these object types. First, we define the concrete syntax `r : T` to mean definite (`:!`) or possible (`:?`) presence, depending on whether `r` defines a finite or infinite set of strings, respectively. Second, since field "names" can be arbitrary regular expressions, it is both tedious and unnecessary to explicitly describe "all remaining possible field names". Instead we support the "catchall" syntax ⋆, which computes that expression automatically. Third, by default we assume all fields are Hidden: any fields not explicitly mentioned by the programmer (or covered by the catchall) are presumed Hidden.[3] For example, the empty object type ⦃⦄ has no accessible fields.

A complete example All of the features described above are necessary to define a (simplified) type for homogeneous arrays. We present the type using the sugar defined above:

```
1 Array = μ array :: ⋆→⋆ .
2     Λ α :: ⋆ . Ref ⦃
3       /[0-9]+|\+Infinity|-Infinity|NaN/ : α,
4       __proto__ : Ref ⦃
5         concat : ∀ β . [array⟨β⟩]array⟨β⟩→array⟨β⟩
6       ⦄,
7       ⋆ : Absent
8     ⦄
```

In words, `Array` is a recursive type-constructor (of kind ⋆→⋆) that produces a reference to an object type whose numeric-like fields (i.e., the infinite set of field names matched by the regular expression)—if present—are bound to values of type α, whose `__proto__` definitely is present and is a reference to an object that contains a method `concat` that operates on two arrays of the same type to produce a new one, and for which all other fields are not accessible (as they are absent on the array object and hidden on its prototype).

[3] In a prior exposition on object types [17], we did not assert the total-coverage requirement, and did not reify Hidden as a possible field type.

$$
\begin{array}{rrl}
l \in & \textit{Locations} & \\
op_p \in & \textit{Prefix ops} & \ni \texttt{!} \mid \texttt{-} \mid \textbf{typeof} \mid \cdots \\
op_i \in & \textit{Infix ops} & \ni \texttt{+} \mid \texttt{\&\&} \mid \texttt{<=} \mid \texttt{==} \mid \textbf{in} \mid \cdots \\
c \in & \textit{Constants} & ::= \textit{num} \mid \textit{str} \mid \textit{bool} \mid \textbf{null} \\
v \in & \textit{Values} & ::= c \mid \textbf{func}(\vec{x})\ \{\ e\ \} \mid \{\ \overrightarrow{str : v}\ \} \\
e \in & \textit{Expressions} & ::= c \mid x \mid op_p\ e \mid e\ op_i\ e \mid \textbf{func}(\vec{x})\ \{\ e\ \} \\
 & & \mid e(e) \mid e\ ;\ e \mid \textbf{if}\ (e)\ e\ \textbf{else}\ e \\
 & & \mid \textbf{let}\ x\ \texttt{=}\ e\ \textbf{in}\ e \mid \textbf{letrec}\ \overrightarrow{x = e}\ \textbf{in}\ e \\
 & & \mid l\texttt{:}\ e \mid \textbf{break}\ l\ e \mid \textbf{try}\ e\ \textbf{catch}(x)\ e \\
 & & \mid \textbf{try}\ e\ \textbf{finally}\ e \mid \textbf{throw}\ e \mid e\texttt{:=}e \\
 & & \mid \textbf{ref}\ e \mid \textbf{deref}\ e \mid \{\ \overrightarrow{str : e}\ \} \mid [\ \vec{e}\ ] \\
 & & \mid e[e] \mid e[e\texttt{=}e] \mid \textbf{delete}\ e[e] \mid e\langle e \rangle \\
 & & \mid \Lambda x <: T\,.\,e \mid \textbf{check}\ T\ e \mid \textbf{cheat}\ T\ e
\end{array}
$$

Figure 3: Syntax of λ_{JS}

3.1 Typing Rules

This type system is defined relative to the λ_{JS} core language [6], which offers a concise and tractable version of (all of) JavaScript. Fig. 3 presents the syntax of λ_{JS} that we type-check, and the typing rules of our system are presented Figs. 5 to 7. Most of the rules are straightforward; the complexity comes from dealing with JS's rules for non-fixed-arity functions, and defining a sufficiently precise system for handling objects.

Rules T-App-Fixed and T-App-Var handle function application, and ensure that if fewer arguments are present than are expected, they may legitimately have type Undef; likewise if more arguments are present than expected, and the function is variadic, that they all match the function type's variadic argument type. To support finitary overloading of functions, we include two ordered, specialized rules for typechecking applications whose first expression has an intersection type (T-App-Inter1 and T-App-Inter2).

Typechecking objects is largely delegated to two auxiliary judgments, $\Gamma \vdash T \Leftarrow T_{obj} @ r$ and $\Gamma \vdash f_1 <:_f f_2$. The former computes the inherited type of a field named r in the object-type T_{obj}, with rules to walk the prototype chain when it is available, a rule to handle Hidden prototypes, and a rule for all other cases. The latter judgment determines when one field descriptor is a "subfield" of another. S-Obj uses both judgments to decide whether two object types are subtypes: ensuring that every visible field on the upper bound is inherited by the lower bound at a subtype (i.e., width subtyping), and by comparing all pairings of fields for subfield compatibility (essentially depth subtyping).

3.2 Type Soundness

The type system presented so far is essentially a combination of the flow-insensitive part of the system used in [7] and the object-typing system of [17], which have both been proven sound. However, because TeJaS provides a modular mechanism for combining type system fragments, we do not have a single type system, but rather a family of type systems. Therefore, to prove soundness of each of them, we would need the ability to demonstrate that the composition of sound type-system modules into a larger type system is also sound. However, combining soundness and modularity is an open problem [10].

More broadly, we believe TeJaS should not be limited to constructing only sound type systems (provided, of course, the unsoundness is explicitly advertised). For instance, in our work on jQuery (Section 2), users are given the choice of making unsound assumptions—a choice that exploits TeJaS's modularity—to reduce the number of type errors they get. In addition, languages like TypeScript are expressly unsound, and we want TeJaS to be suitable for creating such experimental systems; indeed, we reconstruct its essence in Section 5.2. In addition, given the dynamic nature of JavaScript, a system that enabled the construction of only sound type systems would be too restrictive.

4. Ergonomics: Simplifying Writing Types Without Complicating the Core

Our type system needs to be quite sophisticated to support most features of JS: because it is essentially an extension of $F^{\omega}_{<:}$, type inference is undecidable [13]. Accordingly, we allow programmers to supply type definitions and type annotations in comments, ensuring that the annotated program both can run in unmodified JS engines and can be analyzed by our type system:

```
1 /*:: type myType = ...; */
2 function aFunction(x) /*: myType -> Num */ {...}
3 var x = /*: Bool */true;
```

However, the precise details of the types that arise in checking real JS programs are both tiresome and tricky. We therefore need to pay careful attention to the ergonomics of the type system, or else it will be effectively unusable. We focus here on two efforts: applying local type inference wherever possible, and expressing common patterns of type constructions as syntactic sugar. We also highlight the similarities and differences between class-based and prototype-based OO types in our language.

Both efforts above aim to lower the annotation burden on programmers, either by eliding annotations entirely or by defining customized shorthands applicable to the particular task. However, the first is general-purpose: it depends only on types defined by the `Base` system, and can benefit all subsequent users of the system. Further, it is non-trivial to infer types even heuristically in a system as expressive as ours, and so we choose to implement this inference once and for all, and incorporate it into the `Base` type checker. Extensions are free to add additional inference if they so choose. By contrast, because extensions are free to reinterpret concrete written types (via the elaborator, see Section 5), the meaning of syntactic type sugar necessarily varies between extensions. We therefore do not implement these idioms in the `Base` layer, and leave them to extensions.

4.1 Local Type Inference

We formulate our approach as a bidirectional type system [14]: where possible, we propagate type information inward to subexpressions, and thereby avoid the need for explicitly annotating them. This effort is motivated by, and primarily benefits, typechecking function, object and literals.[4] Additionally, we attempt to infer types for variables, but this is surprisingly subtle.

Functions and objects Inferring types for JS functions is challenging because they do not have an explicit arity: it is not an error to call a function with more arguments than it has formal parameters, and when calling functions with fewer arguments than formal parameters, JS will pad the remainder with `undefined` values. Moreover, JS always makes all arguments available via the `arguments` array. As a result, inferring one "best" type for a function is impossible. However, web programming is full of anonymous functions defined solely to be used as event listeners: in these cases, we can easily infer the appropriate type of the handler from the calling context that installs it.

Likewise, it is impossible to define a "best" type for an object literal, as later code may add, modify, or remove fields from that value, and it is impossible from the literal alone to infer what those fields may be. However, when object literals are defined and immediately passed as arguments to functions or assigned into locations of known type, again we can use the context to infer their appropriate types.

Types for variables It is certainly convenient to type variables by their initial value, as it avoids many annotations that are trivially guessable. But doing so presents both an implementation and a semantic challenge. From an implementation standpoint, as part of the desugaring of JS to λ_{JS}, all variables are hoisted to the top of their scope and initialized to **`undefined`**; as this is almost never the intended type for the variable, we detect this pattern and avoid such premature type ascription.

Semantically, choosing a type based on the initializer usually suffices, but when the initializer is a boolean, string or object literal, this choice infers overly-precise types for the variable: namely, `True` or `False`, a singleton string pattern, or the precise type of that object literal. We might reasonably try automatically generalizing to `Bool` or `Str` (though an ap-

[4] Empty array literals, however, provide no information about their contents, and must be annotated with types.

propriate, similarly general object type is less obvious), but doing so uniformly would cripple the precision of the Private Browsing and ADsafety analyses. In practice, the lack of this generalization has not been too onerous an annotation burden, but we leave for future work implementing heuristics to retry typechecking automatically at more general types.

4.2 Syntactic Sugar for Types

Our experiences building several JS type systems have suggested several type idioms, or patterns of types, that occur frequently and are of fairly general utility. As noted earlier, such idioms are not provided by the Base layer; however, we suggest that most extensions would do well to adopt them. We present our basic representation of object types, then show three idioms here: "with" types for simple type extension, "thin" arrows for modeling function objects, and "type trios" for representing JS object-construction patterns.

Objects and references Because JS permits developers to treat objects as dictionaries, and dynamically add and delete their fields, λ_{JS} models objects as *references to immutable records of values*. For example, the code

```
anObj.aNewField = 5;
delete anObj.someField;
```

desugars in essence to[5]

```
anObj := (deref anObj)[aNewField=5];
anObj := delete (deref anObj)[someField];
```

(An alternate design, directly using mutable dictionaries of (possibly) mutable fields, was considered but abandoned. It has considerable difficulty handling these cases compositionally: what is the meaning of `anObj.aNewField`, prior to its creation? Moreover, because JS objects are passed by reference through function calls, at least one level of referencing is inherent in the semantics.)

The *type* of `anObj`, therefore, must be a reference to an object type: Ref⟦aNewField :? Num, someField :? T, ⋆: Absent⟧ (for some type T). But since practically *every* object that JS programmers write will be wrapped in a Ref, we define syntactic sugar to make writing these types easier. Specifically, we provide concrete syntax that automatically supplies the Ref wrapper, as well as syntax for read-only object types and the uncommon case (but still needed; see Section 4.3) of a bare object type:

`{ fields... }` ⇒ Ref⟦*fields...*⟧
`#{ fields... }` ⇒ Src⟦*fields...*⟧
`{{ fields... }}` ⇒ ⟦*fields...*⟧

"With" Types Due to JS's prototype-based nature, objects behave identically to their prototype, except for any fields which they add or override. Accordingly, we provide a convenient syntax for describing the types of these extensions:

```
type Base = { a: Num, b: Str }
type Derived = { Base with b: Num, c: Bool }
// equivalent to
type Derived' = { a: Num, b: Num, c: Bool }
```

`Derived'` contains all the fields of `Base`, plus additional fields that add to or override same-named fields of `Base`.

As a technical note, `with`-types are implemented to commute past recursive types. This allows for easily extending an object type with new methods:

```
type Foo = mu foo :: * . { f: [foo] Num -> Num }
type Bar = { Foo with g : [foo] Str -> Str }
// equivalent to
type Bar' = mu foo :: * . {
  f: [foo] Num -> Num,
  g: [foo] Str -> Str }
```

"Thin", "Fat" and Terse Arrow Type Syntax In JS, functions are objects whose prototype chain reaches `Function`, which implies functions can (and often do) have extra properties attached to them. We therefore model functions as objects with a `__code__` field, whose type is a traditional arrow type. To notate this, we distinguish between "thin arrows" and "fat arrows", where the former desugar into the latter:

```
type aToBfun = ([T]A -> B)
// equivalent to
type aToBfun' = {__code__ : [T]A => B, * : Absent}
```

Both definitions above specify the type (in abstract syntax):

Ref⟦`__code__ : [T]A` → `B, * : Absent`⟧

(Recall, the `[T]` denotes the type of **`this`** within the body of the function.) This notation composes neatly with the previous `with`-type sugar:

```
type PointConstr = {
  ([Point] Num * Num -> Point) with
  prototype : {
    getX : [Point] -> Num,
    getY : [Point] -> Num
} }
// equivalent to
type PointConstr' = {
  __code__  : [Point] Num * Num => Point,
  prototype : {
    getX : [Point] -> Num,
    getY : [Point] -> Num },
  * : Absent }
```

Finally, any function not explicitly called with an invocant is supplied with the global object bound to **`this`**. To prevent such functions from actually using **`this`**, we allow function types to be written without an explicit invocant type:

```
type noThisFun = (A -> B)
// equivalent to
type noThisFun' = ([#{* : Hidden}] A -> B)
```

This read-only object, whose fields are all Hidden, is the supertype of all readable objects; it therefore accepts the global object, but prevents any inadvertent access to its fields.

[5] λ_{JS} actually uses an additional layer of (de-)references, to support aliasing between variables, that is elided here; these references are inserted automatically and do not impact the types described here.

Type trios With-type syntax is convenient for concisely expressing prototype-style extensions of objects. However, JS programmers often use the available object system to emulate a class-like hierarchy. When building such class-like systems in JS, objects fall into one of three roles:

- *Constructor* functions, used to create new instances; fields on these functions are akin to static class members in class-based OO programming;
- *Prototype* objects, whose members are the methods common to all instances of a constructor; and
- *Instances*, created by constructor functions.

These three object types are mutually recursive:

- A constructor's `prototype` field points to the prototype object
- A prototype object's `constructor` field points to the constructor
- An instance's `__proto__` field points to the prototype object

However, the fact that the `__proto__` field is a tangible, visible field is an unfortunate leaky abstraction: class-like code should not be examining the `__proto__` field directly, and in fact, the presence of that field substantially complicates subtyping, and in turn typechecking. Instead, the appropriate model for the instance type must hide the `__proto__` field, and then include all the fields of its prototype, but at inherited (`:^`) presence. This duplication is entirely mechanical, and tedious to construct manually. We therefore provide sugar for simultaneously defining this trio of related types:

```
type constructor FooConstr = {
  [Foo] arguments -> Foo with
  aSharedConstant : Num, ... }
and prototype FooProto = {
  aSharedMethod : [Foo] Num -> Str, ... }
and instance Foo = { field1 : Num, ... }
// equivalent to
type FooConstr = {
  __code__ : [Foo] arguments -> Foo,
  aSharedConstant : Num, ... ,
  prototype : FooProto }
and FooProto = {
  aSharedMethod : [Foo] Num -> Str, ... ,
  constructor : FooConstr }
and Foo = { field1 : Num, ... ,
  __proto__ : Absent,
  aSharedMethod :^ [Foo] Num -> Str }
```

4.3 Matching Class-like Subtyping with Prototype Inheritance

It is an appealing but flawed intuition that objects further down a prototype chain are necessarily subtypes of objects closer to the chain's root. For example, nothing prevents a programmer from writing

```
var base = {x : 5, y : 6};
var derived = {__proto__ : base, x : false};
```

While `derived` and `base` contain the same field names, their types differ; clearly they cannot be used interchangeably, so their types must not be related by subtyping. But consider a more class-like object hierarchy, deliberately written with suggestive syntactic sugar:

```
type constructor BaseConstr = [Base] -> Base
and prototype BaseProto = {toString : [Base] -> Str}
and instance Base = {x : Num, y : Num}

type constructor DerivedConstr = [Derived] -> Derived
and prototype DerivedProto = {__proto__: BaseProto}
and instance Derived = {Base with z : Num}
```

Every field present in `Base` is also present in `Derived`: `x` and `y` are explicitly present, and `toString` (inherited from `BaseProto`) is implicitly present (through `DerivedProto`'s `__proto__`). But `Derived` is *not* a subtype of `Base`: once desugared, both types are revealed as *references* to object types, and hence can only subtype *invariantly*—and `Base` is clearly not a subtype of `Derived`, as it has fewer fields. (In fact, even `DerivedProto` is not a subtype of `BaseProto`, for a similar reason.)

Accordingly, even though the following code will run without error, it will not typecheck as written:

```
/*: Base * Num -> Base */
function setX(obj, val) {
  obj.x = val;
  return obj;
}
aDerivedObj = setX(aDerivedObj, 5);
```

The class-like intuition behind these types, however, is *almost* right: rather than specifically requiring an argument of type Ref {{x:Num,y:Num}} (which is the definition of `Base`), the function `setX` need only require a reference to *any object* with a numeric `x` field. With judicious and clever use of bounded polymorphism, we can write this type precisely:

```
/*: forall a <: {{x:Num}}.ref a * Num->ref a */
function setX(obj, val) ...
```

Here we see the need for bare object types, so that we can constrain the fields of the object covariantly, without the interference of the outer Ref type constructor.

While correct, the type above has two shortcomings in our current system. First, it rapidly becomes unwieldy to define the types for functions that take multiple instance-typed arguments. We have not yet implemented syntactic sugar to simplify the situation, though it should not prove difficult. The second problem, however, is that type inference for such bounded-polymorphic functions does not currently work in the higher-order case (i.e., passing `setX` and `aDerivedObj` to another function). This is not a failure unique to TeJaS; as mentioned earlier, our type system is essentially that of System $F^{\omega}_{<:}$, and so type inference is in general undecidable. We plan to exploit the syntactic sugar for these function types to guide our inference heuristics (see Section 8).

5. Modularizing the System

The base system is naturally quite large, so building variations on it is not trivial. Drawing on the experience gained from our several variations (Section 2), we have therefore modularized the type system. In particular, we distinguish the following seven features:

1. the type and kind language, as presented in Fig. 1, and the binding forms needed for environments;
2. environments over this type language;
3. kind checking over this type language;
4. subtyping over these environments and type language;
5. typechecking over these environments and type language;
6. a decorator for weaving these type annotations into λ_{JS} expressions; and
7. an elaborator for desugaring written type annotations into this type language.

ML's functor system allows us to separate these features into modules and explicitly record the dependencies among them. Such functorization allows us to, say, independently vary the concrete syntax recognized by the elaborator (7) without having to rewrite the type definitions (1), or change the subtyping rules (4) without affecting the decorator (6).

By themselves, these module boundaries are simply clean software engineering. But ML also permits mutually recursive functors, and we can exploit this ability to define extension points for our type system. In particular, we split each of these seven modules into two layers. The ***Base*** layer will define the base system (as presented in Section 3), leaving holes in its definitions for an *Extension* layer to fill. Concretely, the `Base` types module defines

```
type typ = TRef of typ | TRegex of pat | ...
         | TEmbed of Ext.typ
type kind = KStar | KFunc of kind * kind
          | KEmbed of Ext.kind
```

The `TEmbed` and `KEmbed` constructors take arguments whose types are defined by extension-layer modules. Those types in turn define constructors `TBase` and `KBase` that recursively include base types and kinds, respectively. Each pair of corresponding modules is instantiated mutually recursively. The resulting composite system includes all the behaviors of the `Base` layer, and can extend or override them without having to reimplement everything from scratch.

(Note that most extensions may not care about modifying all seven of these layers; for example, the definition of environments is often unchanged. Nevertheless, our functorization and ML's type system together require that skeleton versions of all seven modules must be written. This ensures consistency: it is impossible to mistakenly mix modules from multiple type systems.)

The next two sections illustrate the effectiveness of this functorized approach. To demonstrate the range of expressiveness, the rest of this section presents the central type definitions from three experiments: an empty extension that simply instantiates the base system, an extension that overrides function types to implement TypeScript-like semantics, and an extension that adds new kinds to produce a type system tailored to analyzing code using the jQuery library. These extensions mainly use features 1, 4 and 5.

In Section 6, we highlight the flexibility of reified type environments (made possible by features 2 and 7). In particular, we show that a single type checker implementation can yield results of varying precision depending on the environment, and also show how we can import type descriptions written without TeJaS in mind into our environments.

Discussion We chose ML's functor system because our initial, purpose-built type systems were already implemented in ML, and it was relatively easy to refactor them into functors. Our design is not, however, intrinsically tied to ML.

While we have encoded open data types within mutually-recursive ML functors, the syntactic overhead is non-trivial. Additionally, module (and function) boundaries make it tricky to surgically replace, say, just one typing rule. Consequently, a complete re-engineering of our system might try other languages or take advantage of recent progress in extensible type system design (see Section 7).

5.1 Example: Base Type System Only

The `Bare` extension adds nothing to the base type system. Accordingly, the types-definition module is, in its entirety:

```
module Make : BARE_TYP =
  functor (BASE : TYPS) -> struct
    type baseKind = BASE.kind
    type kind = KBase of baseKind
    type baseTyp = BASE.typ
    type typ = TBase of baseTyp
    type baseBinding = BASE.binding
    type binding = BBase of baseBinding
    type env = binding list IdMap.t
  end
```

In total, the `Bare` extension comprises 1,600LOC, though much of this is formulaic: 400 lines are entirely ML boilerplate; 100 lines parse written type annotations; and another 250 lines implement the syntactic sugar defined earlier.

This `Bare` system alone can typecheck a non-trivial amount of code. In prior work [7, 18], we examined a corpus of Google Gadgets, 3,800 lines of interactive, third-party code. Preparing these programs for analysis required minor refactoring (e.g., explicitly declaring variables with `var`) and adding explicit type annotations; the precise counts are shown in Fig. 4. Once annotated, all but two lines of these programs passed the typechecker.[6] (The two exceptions, in `watchimer.js`, use **`typeof`** tests to distinguish between **`undefined`** and initialized values, and proving these lines type-safe requires flow typing [7].)

Annotating third-party code with rich types can be laborious. Thus, as an orthogonal enhancement to this type

[6] Though these experiments were conducted on an earlier implementation of our system, the type rules are the same.

system—i.e., neither reliant upon nor overly specialized for the particular type system variant being used—Saftoiu [18] wrote a JS-to-JS compiler that instrumented each gadget to record type information. After manually interacting with each gadget to obtain reasonable code coverage, the recorded types can then be used as a first approximation of the intended types of the code. (Because this approach necessarily under-approximates program behavior, it generally is not sufficient. In our case, it is followed by actually running the type-checker, and the dynamic procedure only provides candidate types.) This technique succeeds with over 91% of the necessary annotations; Fig. 4 gives further details.

Filename	LOC	Changes (% of LOC)	Annotations (% of changes)	Refactorings (% of changes)	Inferred (% of annotations)
animation.js	70	5 (7.14%)	5 (100.00%)	0 (0.00%)	4 (80.00%)
metronome.js	106	16 (15.09%)	12 (75.00%)	4 (25.00%)	10 (83.33%)
countdown.js	129	8 (6.20%)	4 (50.00%)	4 (50.00%)	4 (100.00%)
catchit.js	165	25 (15.15%)	9 (36.00%)	16 (64.00%)	6 (66.67%)
hashapass.js	257	30 (11.67%)	20 (66.67%)	10 (33.33%)	14 (70.00%)
morse.js	275	25 (9.09%)	12 (48.00%)	13 (52.00%)	12 (100.00%)
rsi.js	328	49 (14.94%)	22 (44.90%)	27 (55.10%)	22 (100.00%)
topten.js	443	85 (19.19%)	18 (21.18%)	67 (78.82%)	18 (100.00%)
text2wav.js	488	50 (10.25%)	41 (82.00%)	9 (18.00%)	38 (92.68%)
resistor.js	591	52 (8.80%)	32 (61.54%)	20 (38.46%)	32 (100.00%)
watchimer.js*	947	34 (3.59%)	17 (50.00%)	17 (50.00%)	15 (88.24%)
TOTAL	3,799	379 (9.98%)	192 (50.66%)	187 (49.34%)	175 (91.15%)

Figure 4: Typechecking Google Gadgets, and counts of necessary refactorings and annotations. *Note: watchimer.js contains two `typeof` tests that need flow-sensitive typing to be checked.

5.2 Example: TypeScript's Covariant Function Calls

As a proof of concept, we have implemented an extension to provide TypeScript's semantics for functions. This extension overrides the `TArrow` type of our base system, and replaces it with one that has the new semantics. The types-definition module is gratifyingly similar to the `Bare` one: the only change necessary is adding a single type constructor

```
    type typ =
      | TBase of BASE.typ
      | TArrow of typ list * typ option * typ
```

In fact, the entire extension is only 1,860LOC: other than minor naming-convention differences, the 260-line difference between the two is precisely that which defines how TypeScript's arrow types behave. Furthermore, the developer of this experimental system inherits the entire, complex base type system (Section 3), as well as the additional features like inference described above (Section 4). Indeed, the base language of TeJaS is richer and more powerful than that of TypeScript (as we discuss in Section 7), demonstrating the value of our modular approach.

Using this extended type system, the following example typechecks, while it fails when using the `Bare` system:

```
/*:: type subt = #{ x : Num, y : Str }
     type supert = #{ x : Num }  */
/*: #(#(supert * Num -> Str) * supert -> Str) */
function f2(aFun, obj) { return g2(obj, 5); }
/*: #(subt -> Str) */
function g2(obj) { return obj.y + "foo"; }
f2(g2, { x : 5, y : true });
```

In this example, the actual type of `g2` has a different arity than that expected for `aFun`; moreover, its expected argument is a *subtype* of the one for `aFun`. Implementing such drastically different behavior for arrow types was pleasingly straightforward, and required only minimal interaction with the existing typing rules.

5.3 Example: Adding New Kinds

In Section 2 we discuss the application of TeJaS to jQuery programs. To analyze jQuery client code, we built a type system that could describe jQuery collections, and in particular that could capture the essential notion of size information.[7] For example, rather than typing a non-empty query result as `jQuery⟨Element⟩`, we can more precisely give it the type `jQuery⟨1+⟨Element⟩⟩`. We implemented this type system as an extension of TeJaS: the key features of this extension required defining a new kind, *"multiplicities"*, to describe size information (i.e., the `1+⟨·⟩`). Additionally, we overrode two existing type constructors to enable them to use multiplicities. The types-definition module for the `JQuery` type system is (see [8] for the meaning of these constructions)

```
module Make : JQUERY_TYP =
    functor (Css : Css.CSS) ->
      functor (BASE : TYPS) -> struct
  type baseKind = BASE.kind
  type kind =
    | KBase of baseKind
    | KMult of kind
  type baseTyp = BASE.typ
  type typ =
    | TBase of BASE.typ
    (* can quantify or abstract over multiplicities *)
    | TForall of id * sigma * typ
    | TLambda of (id * kind) list * typ
    | TApp of typ * sigma list
    (* description of partial page structure *)
    | TDom of ...
  and multiplicity = ...
  and sigma = STyp of typ | SMult of multiplicity
  type baseBinding = BASE.binding
  type binding = BBase of BASE.binding
    | BMultBound of multiplicity * kind
  type env = binding list IdMap.t
end
```

The jQuery type system is approximately 5,770LOC (including the 1,600LOC of `Bare` boilerplate), and includes functionality (e.g., a decision procedure for `CSS` selectors) that integrates smoothly into the `Base` type system. Again, most `Base` functionality was directly reused, without any changes.

[7] Sized and indexed types have been described before [20, 26]; such approaches are infeasible for the JS setting [8].

6. Parameterizing the Type Environment

Research on type systems tends to focus on the rules of inference that characterize the type system. In practice, however, the type environment—which is rarely discussed in papers—is just as important. Furthermore, we have found that the environment represents a powerful point of modularity that is often exploited at best implicitly in other systems. Therefore, TeJaS makes this an explicit point of variation.

6.1 Explicit, External Environment Definitions

From an implementation perspective, once given the ability to parse a written representation of types and desugar them into the internal type representation (needed for programmer-supplied annotations, as in Section 4), it is straightforward to reify the initial type environment as an external file, and to parse that file accordingly. Exposing the initial environment has several engineering advantages:

- A single environment file can be used by multiple type-system variants. This avoids a substantial duplication of trusted code, with all its attendant benefits. Moreover, by sharing a concrete syntax, the same environment file can be reinterpreted by each variant type system: for instance, the type of a built-in function could be given TeJaS's bare semantics or TypeScript's semantics, just by changing which type desugarer is applied.
- A single type system can use multiple environments that define types of varying precision. For instance, consider two environments that define two possible types for the plus operator:

```
"JS-like": (Num × Num → Num)∩(⊤ × ⊤ → Str)
"Strict" : (Num × Num → Num)∩(Str × Str → Str)
```

  The former type more closely models JS's behavior, but the latter type ensures that, for instance, objects are never added inadvertently. Such precision may or may not be needed, depending on the analysis, and the choice need not be hard-coded into the type system itself. We have used this flexibility as a form of progressive typing [16], to provide a set of guarantees of varying precision when analyzing jQuery programs [8].
- Assigning a type to the global object becomes markedly easier. Specifically, we can define the type of JS built-ins:

```
type JSGlobal = { Math : ..., JSON : ..., ... }
```

  and then reuse all our type sugar to separately define a basic type for the global scope of web pages:

```
type Window = mu w :: * . { JSGlobal with
  window : w,
  setInterval : [w]([w]->Undef) * Num -> Num,
  ...
}
```

  We could refine it further with more precise information if needed:

```
type HTMLWindow = { Window with
  Node : ...,
  Element : ...,
  DivElement : ...,
  ...
}
```

  To use any of these types, we simply assign it as the type of the `%global` identifier.

6.2 Importing DOM Bindings from WebIDL

Defining every HTML element type manually is tedious, incomplete and—fortunately—unnecessary. The interfaces for all these objects are defined in WebIDL [24], a language similar to C++ header files that defines the types of the methods and properties of these objects. Modern browsers use these IDL files to automatically generate the glue code connecting their JS runtime to their internal implementation. Additionally, Firefox uses IDL further to define its thousands of extension APIs.

Beneath the concrete syntax, WebIDL files are essentially type environments. To exploit this, we have built a parser for them that translates WebIDL definitions into TeJaS types, and (using the same `with` sugar as above) consolidates these types into the global-object type. As with the variously-precise environments above, we can vary the IDL-to-TeJaS translator to provide different encodings of the built-in objects. We used this facility extensively as we developed our Private Browsing analysis. For instance, we defined an analogue of `with`-types for IDL-defined types, so that we could precisely and easily override the definitions of the specific types relevant to private browsing.

7. Related Work

Work related to TeJaS broadly falls into two groups: other modular type system architectures and other type systems for JS. We briefly survey both groups here.

7.1 Extensible or Modular Type Systems

Creating extensible type systems architectures is a research field in its own right [10, 12]. Some recent progress [5, 19] shows how to decompose monolithic languages and meta-theory into composable modules. These approaches do not yet support the flexibility of use cases we consider here.

Declarative, extensible type systems TinkerType [10] supports a feature-based approach to composing type systems. Individual rules can be written separately, tagged by features, and then all rules in a given feature set are concatenated to form the entire type system. Our setting is slightly different: in addition to composing (nearly-)orthogonal type system features, TeJaS allows for redefining extant type features (as in TypeScript), for which TinkerType provides no particular extra support.

Modular Meta-theory Delaware et al. [5] focus on the modularity and reusability of meta-theoretical results for a language, and refactor both the representation of expressions and types as algebras so that their semantics can be composed. Their approach is elegant, but relies crucially on parametricity to "hide" recursive calls in their algebras. Concretely, this forbids "deep inspection" of expression forms (beyond their root symbol), which eliminates some of the expressiveness that TeJaS provides.

Progressive Types The ADsafety and private browsing projects [9, 15] used type systems to ensure safety properties on target programs, but perhaps surprisingly, allowed programs to typecheck that contained obvious "traditional" type errors, such as possibly having a non-function in function-call position. Such errors could not compromise the intended safety guarantees (as erroneous code would just halt), and so did not need to be prevented.

This insight led to recognizing another dimension in type checkers: the ability to parameterize the typed progress guarantee with a set of allowable runtime errors [16]. Operationally, TeJaS implements this idea via typechecker metafunctions that optimistically select from union types the components that do in fact typecheck. Such metafunctions are not yet TeJaS extension points; that remains future work.

7.2 Other JS Type Systems

Language-modifying approaches AltJS (`http://altjs.org`) collects a suite of tools that "enhance JS with static typing". More accurately, they define new languages with new semantics and type systems, and compile the results to JS. Most of these type systems deliberately eschew many of the features of JS in favor of more typical object-oriented schemes. Some of these systems include structural types; none of them include the support for first-class field names that is necessary to support punning of objects as dictionaries. Others stay closer to idiomatic JS semantics, but in doing so deliberately discard type-system soundness.

TypeScript Microsoft recently released TypeScript (`http://www.typescriptlang.org`), a variant of JS that is easily compilable to standard JS, and that incorporates syntax and types to make writing class-like object patterns more pleasant (e.g., interface declarations that are implicitly recursive; defining functions with optional arguments); we could adopt some of these as syntactic sugar in TeJaS. However, TypeScript's type system is substantially simpler than ours: it does not contain union or intersection types, or refined string types and their consequent support for first-class field names. It does support overloading of functions based on string constants. Concretely, this means it cannot generally give a precise type to a dictionary-lookup function (as done in the ADsafe work), but can precisely type it in limited settings. TypeScript supports defining overloaded types for methods (essentially a limited form of intersection types for functions), but oddly does not then check these types.

In fact, by traditional type system standards, and by their own admission,[8] TypeScript's type system is unsound: code that passes the typechecker can still produce runtime errors or unintended (and arguably unintuitive) behavior.

As noted (Section 3.2), TeJaS's base type system is sound, and so cannot directly mimic TypeScript's behavior—but as Section 5.2 showed, it supports extensions that change the typing rules for function types, to admit such cases.

7.3 DJS

Dependent JavaScript (DJS) [2] is another type-checker for JavaScript that uses very different techniques from ours. DJS supports nested refinements [3], flow-sensitive heap types, and other novelties, which it uses to type-check idiomatic JavaScript code. The particular set of idioms it supports are drawn from *JavaScript: The Good Parts* [4]. In contrast, our approach is to support a parameterizable type system, which doesn't give precedence to any particular set of idioms (which, our experiments have shown, numerous real-world code-bases do not follow anyway). The flexibility afforded by our design shows in our evaluation: our benchmarks are significantly larger and more diverse than those of DJS. Furthermore, the parameterization in TeJaS should make it relatively easy to build a "Good Parts"-style checker.

8. Future Improvements

There are many directions to further improve TeJaS, including the ergonomics of the type language, the expressiveness of the core type system, and the architecture of TeJaS itself. We briefly describe possible work in each of these areas.

Additional type sugar Section 4.2 described several forms of syntactic sugar for types that have been very convenient in our experiments, but as Section 4.3 highlighted, several other idioms are still cumbersome to use. Defining class-like types whose subtyping relation coincides with an OO-intuition of subclassing is particularly tricky, requiring subtle use of bounded quantification. Hiding that subtlety with new type sugar would make it far easier to use, while still retaining the flexibility of the TeJaS type system to handle non-class-like types.

Type inference Currently, TeJaS does not attempt any type inference procedure beyond the bidirectional propagation of type constraints, largely because the type inference problem is known to be undecidable. Saftoiu [18] began an effort at inferring types from execution, but this is capable of constructing only a first approximation of a program's true types. In our various type system experiments (Section 2), however, we have found that *problem-specific* type inference has been surprisingly effective: for instance, type inference for the ADsafe and jQuery sublanguages of JS is decidable. We look to additional problem domains for inspiration in finding additional practical, useful type inference approaches.

[8] `https://typescript.codeplex.com/discussions/428572`

Improving Type System Expressiveness Guha et al. [7] developed flow typing, which supports type narrowing as a consequence of control flow. For instance, the following code typechecks soundly with such flow-sensitive reasoning:

```
function length(s) /*: (Str + Undef) -> Num*/ {
  if (s === undefined) return 0;
  return s.length; // Note: No longer undefined
}
```

However, if that `undefined`-check were more complicated, and abstracted into some external function, flow typing cannot verify this code as safe. Such reasoning about the predicates implied by the return values of functions is known as *occurrence typing* [23]. Occurrence typing can generalize flow typing's specialized support for the `typeof` operator, and can also support reasoning about JS's various reflective operators, such as `Object.hasOwnProperty`.

Extensible Architecture TeJaS grew out of our several projects to analyze specific JS problems via type systems. However, as it has grown, each of our experiments has suggested architectural changes to improve its flexibility, and we expect future projects to encourage still further extension.

Acknowledgments

This work is partially supported by the US NSF. We thank our co-authors who worked on the individual type systems described here.

References

[1] A. H. Borning and D. H. H. Ingalls. A type declaration and inference system for Smalltalk. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1982.

[2] R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2012.

[3] R. Chugh, P. M. Rondon, and R. Jhala. Nested refinements: a logic for duck typing. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012.

[4] D. Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[5] B. Delaware, B. C. de Oliveira, and T. Schrijvers. Meta-theory à la carte. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2013.

[6] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2010.

[7] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *European Symposium on Programming Languages and Systems (ESOP)*, 2011.

[8] B. S. Lerner, L. Elberty, J. Li, and S. Krishnamurthi. Combining form and function: Static types for JQuery programs. In *European Conference on Object-Oriented Programming (ECOOP)*, 2013.

[9] B. S. Lerner, L. Elberty, N. Poole, and S. Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. In *European Symposium on Research in Computer Security (ESORICS)*, Sept. 2013.

[10] M. Y. Levin and B. C. Pierce. TinkerType: a language for playing with formal systems. *Journal of Functional Programming (JFP)*, 13(2):295–316, 2003.

[11] J. H. Morris. *Lambda-calculus Models of Programming Languages*. PhD thesis, Massacheusetts Institute of Technology, 1968.

[12] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: an extensible compiler framework for Java. In *International Conference on Compiler Construction (CC)*, 2003.

[13] B. C. Pierce. Bounded quantification is undecidable. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1992.

[14] B. C. Pierce and D. N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

[15] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: type-based verification of JavaScript sandboxing. In *USENIX Security Symposium*, Aug. 2011.

[16] J. G. Politz, H. Q. de la Vallee, and S. Krishnamurthi. Progressive types. In *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2012.

[17] J. G. Politz, A. Guha, and S. Krishnamurthi. Semantics and types for objects with first-class member names. In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2012.

[18] C. Saftoiu. JSTrace: Run-time type discovery for JavaScript. Technical Report CS-10-05, Brown University, 2010.

[19] C. Schwaab and J. G. Siek. Modular type-safety proofs in Agda. In *Programming Languages meets Program Verification (PLPV)*, 2013.

[20] V. Simonet and F. Pottier. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(1):1–56, 2007.

[21] V. St-Amour, S. Tobin-Hochstadt, M. Flatt, and M. Felleisen. Typing the numeric tower. In *Practical Aspects of Declarative Languages (PADL)*, 2012.

[22] N. Suzuki. Inferring types in smalltalk. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1982.

[23] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2008.

[24] W3C. Web IDL. Written Apr. 2012. http://www.w3.org/TR/WebIDL/.

[25] A. K. Wright and R. Cartwright. A practical soft type system for Scheme. In *ACM conference on LISP and functional programming (LFP)*, 1994.

[26] C. Zenger. Indexed types. *Theoretical Computer Science*, 187 (1–2):147–165, 1997.

$ty(num) = \mathsf{Num} \qquad ty(\texttt{"}s\texttt{"}) = s \qquad ty(/r/) = \mathsf{RegExp} \qquad ty(\textbf{true}) = \mathsf{True} \qquad ty(\textbf{false}) = \mathsf{False} \qquad ty(\textbf{null}) = \mathsf{Null}$

$$ty(\textbf{undefined}) = \mathsf{Undef}$$

TYPECHECKING: $\Gamma \vdash_{\Downarrow} e : T$

$$\text{C-SUB}\ \frac{\Gamma \vdash_{\Uparrow} e \to S \qquad \Gamma \vdash S <: T}{\Gamma \vdash_{\Downarrow} e : T} \qquad \text{C-EXPOSE}\ \frac{\Gamma \vdash T \rightsquigarrow T' \qquad \Gamma \vdash_{\Downarrow} e : T'}{\Gamma \vdash_{\Downarrow} e : T} \qquad \text{C-LABEL}\ \frac{\Gamma \vdash_{\Uparrow} e \to S \qquad \Gamma \vdash S <: T}{\Gamma \vdash_{\Downarrow} l\texttt{:}\ e : T}$$

$$\text{C-LET}\ \frac{\Gamma \vdash_{\Uparrow} e_1 \to S \qquad \Gamma, x : S \vdash_{\Downarrow} e_2 : T}{\Gamma \vdash_{\Downarrow} \textbf{let}\ \texttt{x = }\ e_1\ \textbf{in}\ e_2 : T} \qquad \text{C-INTER}\ \frac{\Gamma \vdash_{\Downarrow} e : T_1 \qquad \Gamma \vdash_{\Downarrow} e : T_2}{\Gamma \vdash_{\Downarrow} e : T_1 \cap T_2} \qquad \text{C-UNION-L}\ \frac{\Gamma \vdash_{\Downarrow} e : T_1}{\Gamma \vdash_{\Downarrow} e : T_1 \cup T_2} \qquad \text{C-UNION-R}\ \frac{\Gamma \vdash_{\Downarrow} e : T_2}{\Gamma \vdash_{\Downarrow} e : T_1 \cup T_2}$$

$$\text{C-FUNC-FIXED}\ \frac{\left|\overrightarrow{T_a}\right| = n = \left|\overrightarrow{a}\right| \quad \Gamma' = \Gamma \text{ with labels (i.e., } l =_L T_l) \text{ removed} \quad \Gamma', \textbf{this} : T_{this}, a_1 : T_{a_1}, \cdots, a_n : T_{a_n} \vdash_{\Downarrow} b : T_{ret}}{\Gamma \vdash_{\Downarrow} \textbf{func(}\overrightarrow{a}\textbf{)}\ \texttt{\{}\ b\ \texttt{\}} : [T_{this}]\,\overrightarrow{T_a} \to T_{ret}}$$

$$\text{C-FUNC-VAR}\ \frac{\left|\overrightarrow{T_a}\right| = n \le m = \left|\overrightarrow{a}\right| \qquad \Gamma \vdash \mathsf{Undef} <: T_{var} \quad \Gamma' = \Gamma \text{ with labels (i.e., } l =_L T_l) \text{ removed} \quad \Gamma', \textbf{this} : T_{this}, a_1 : T_{t_1}, \cdots, a_n : T_{a_n}, a_{n+1} : T_{var}, \cdots, a_m : T_{var} \vdash_{\Downarrow} b : T_{ret}}{\Gamma \vdash_{\Downarrow} \textbf{func(}\overrightarrow{a}\textbf{)}\ \texttt{\{}\ b\ \texttt{\}} : [T_{this}]\,\overrightarrow{T_a} \times T_{var} \ldots \to T_{ret}}$$

$$\text{C-ARRAY}\ \frac{\Gamma \vdash T <: \mathsf{Array}\langle T_e \rangle \qquad \forall i.\, \Gamma \vdash_{\Downarrow} e_i : T_e}{\Gamma \vdash_{\Downarrow} \texttt{[}\ \vec{e}\ \texttt{]} : T} \qquad \text{C-OBJ}\ \frac{abs = /.*/ \setminus \bigcup_i n_i \qquad \Gamma \vdash T_i \Leftarrow \{\!\{\vec{f}\}\!\}@n_i \qquad \Gamma \vdash_{\Downarrow} e_i : T_i \quad \Gamma \vdash \{\!\{abs : \mathsf{Absent}, \overrightarrow{n_i : T_i}\}\!\} <: \{\!\{\vec{f}\}\!\}}{\Gamma \vdash_{\Downarrow} \texttt{\{}\ \overrightarrow{n : e}\ \texttt{\}} : \{\!\{\vec{f}\}\!\}}$$

EXPOSING THE TYPE OF A FIELD: $\Gamma \vdash T \Leftarrow \{\!\{\vec{f}\}\!\}@r$

$$\text{I-PROTO}\ \frac{T_o = \left\{\!\!\left\{ \begin{array}{l} hid : \mathsf{Hidden}, abs : \mathsf{Absent}, \texttt{\_\_proto\_\_} : \mathsf{Ref}\, T_{prot}, \\ \overrightarrow{f_p :!\, T_p}, \overrightarrow{f_i :\hat{}\, T_i}, \overrightarrow{f_m :?\, T_m} \end{array} \right\}\!\!\right\} \quad r \cap hid = \emptyset \quad \Gamma \vdash T'_r \Leftarrow T_{prot}@\Big(r \cap \Big(abs \cup \bigcup f_m\Big)\Big) \quad T_r = T'_r \cup \Big(\bigcup_{i : r \cap f_{p_i} \neq \emptyset} T_{p_i}\Big) \cup \Big(\bigcup_{i : r \cap f_{m_i} \neq \emptyset} T_{m_i}\Big) \cup \Big(\bigcup_{i : r \cap f_{i_i} \neq \emptyset} T_{i_i}\Big)}{\Gamma \vdash T_r \Leftarrow T_o@r}$$

$$\text{I-NO-PROTO}\ \frac{T_o = \left\{\!\!\left\{ \begin{array}{l} hid : \mathsf{Hidden}, abs : \mathsf{Absent}, \texttt{\_\_proto\_\_} : \mathsf{Null}, \\ \overrightarrow{f_p :!\, T_p}, \overrightarrow{f_i :\hat{}\, T_i}, \overrightarrow{f_m :?\, T_m} \end{array} \right\}\!\!\right\} \quad r \cap hid = \emptyset \quad T_u = \begin{cases} \bot & r \cap \Big(abs \cup \bigcup_i f_{m_i}\Big) = \emptyset \\ \mathsf{Undef} & \text{otherwise} \end{cases} \quad T_r = T_u \cup \Big(\bigcup_{i : r \cap f_{p_i} \neq \emptyset} T_{p_i}\Big) \cup \Big(\bigcup_{i : r \cap f_{m_i} \neq \emptyset} T_{m_i}\Big) \cup \Big(\bigcup_{i : r \cap f_{i_i} \neq \emptyset} T_{i_i}\Big)}{\Gamma \vdash T_r \Leftarrow T_o@r}$$

$$\text{I-HIDDEN-PROTO}\ \frac{T_o = \{\!\{hid : \mathsf{Hidden}, abs : \mathsf{Absent}, \overrightarrow{f_p :!\, T_p}, \overrightarrow{f_i :\hat{}\, T_i}, \overrightarrow{f_m :?\, T_m}\}\!\} \quad \texttt{\_\_proto\_\_} \subseteq hid \quad r \cap \Big(hid \cup abs \cup \bigcup_i f_{m_i}\Big) = \emptyset \quad T_r = \Big(\bigcup_{i : r \cap f_{p_i} \neq \emptyset} T_{p_i}\Big) \cup \Big(\bigcup_{i : r \cap f_{i_i} \neq \emptyset} T_{i_i}\Big)}{\Gamma \vdash T_r \Leftarrow T_o@r}$$

$$\text{I-EMPTY}\ \frac{}{\Gamma \vdash \bot \Leftarrow \{\!\{\vec{f}\}\!\}@\epsilon}$$

EXPOSING TYPES BELOW BOUNDS: $\Gamma \vdash T \rightsquigarrow T'$

$$\frac{x <: T_1 \in \Gamma \qquad \Gamma \vdash T_1 \searrow T_2 \qquad \Gamma \vdash T_2 \rightsquigarrow T_3}{\Gamma \vdash x \rightsquigarrow T_3} \qquad \frac{x = T_1 \in \Gamma \qquad \Gamma \vdash T_1 \searrow T_2 \qquad \Gamma \vdash T_2 \rightsquigarrow T_3}{\Gamma \vdash x \rightsquigarrow T_3} \qquad \frac{\Gamma \vdash T \searrow T'}{\Gamma \vdash T \rightsquigarrow T'}$$

SIMPLIFYING TYPES: $\Gamma \vdash T \searrow T'$

$$\frac{\Gamma \vdash T[\mu x\,.\,T/x] \searrow T'}{\Gamma \vdash \mu x\,.\,T \searrow T'} \qquad \frac{T \neq x}{\Gamma \vdash T \searrow T}$$

Figure 5: Type checking rules, and simplifying types

TYPE SYNTHESIS: $\Gamma \vdash_{\Uparrow} e \to T$

$$\textsc{T-Id}\ \frac{x : T \in \Gamma}{\Gamma \vdash_{\Uparrow} x \to T} \qquad \textsc{T-Prim}\ \frac{}{\Gamma \vdash_{\Uparrow} c \to ty(c)} \qquad \textsc{T-Expose}\ \frac{\Gamma \vdash_{\Uparrow} e \to T \qquad \Gamma \vdash T \rightsquigarrow T'}{\Gamma \vdash_{\Uparrow} e \to T'}$$

$$\textsc{T-Let}\ \frac{\Gamma \vdash_{\Uparrow} e_1 \to S \qquad \Gamma, x : S \vdash_{\Uparrow} e_2 \to T}{\Gamma \vdash_{\Uparrow} \texttt{let x = } e_1 \texttt{ in } e_2 \to T}$$

$$\textsc{T-Seq-}\bot\ \frac{\Gamma \vdash_{\Uparrow} e_1 \to \bot}{\Gamma \vdash_{\Uparrow} e_1 \ ; \ e_2 \to \bot} \qquad \textsc{T-Seq}\ \frac{\Gamma \vdash_{\Uparrow} e_1 \to T_1 \qquad \Gamma \vdash_{\Uparrow} e_2 \to T_2}{\Gamma \vdash_{\Uparrow} e_1 \ ; \ e_2 \to T_2}$$

$$\textsc{T-Check}\ \frac{\Gamma \vdash_{\Downarrow} e : T}{\Gamma \vdash_{\Uparrow} \textbf{check}\ e\, T \to T} \qquad \textsc{T-Cheat}\ \frac{}{\Gamma \vdash_{\Uparrow} \textbf{cheat}\ e\, T \to T}$$

$$\textsc{T-Array}\ \frac{\left|\vec{e}\right| > 0 \qquad \Gamma \vdash_{\Uparrow} e_i \to T_i}{\Gamma \vdash_{\Uparrow} [\ \vec{e}\ ] \to \mathsf{Array}\langle \bigcup T_i \rangle} \qquad \textsc{T-Ref}\ \frac{\Gamma \vdash_{\Uparrow} e \to T}{\Gamma \vdash_{\Uparrow} \textbf{ref}\ e \to \mathsf{Ref}\, T}$$

$$\textsc{T-Deref-1}\ \frac{\Gamma \vdash_{\Uparrow} e \to \mathsf{Ref}\, T}{\Gamma \vdash_{\Uparrow} \textbf{deref}\ e \to T} \qquad \textsc{T-Deref-2}\ \frac{\Gamma \vdash_{\Uparrow} e \to \mathsf{Src}\, T}{\Gamma \vdash_{\Uparrow} \textbf{deref}\ e \to T}$$

$$\textsc{T-SetRef-1}\ \frac{\Gamma \vdash_{\Uparrow} e_1 \to \mathsf{Ref}\, T \qquad \Gamma \vdash_{\Downarrow} e_2 : T}{\Gamma \vdash_{\Uparrow} e_1 \texttt{:=} e_2 \to T}$$

$$\textsc{T-SetRef-2}\ \frac{\Gamma \vdash_{\Uparrow} e_1 \to \mathsf{Sink}\, T \qquad \Gamma \vdash_{\Downarrow} e_2 : T}{\Gamma \vdash_{\Uparrow} e_1 \texttt{:=} e_2 \to T} \qquad \textsc{T-Label}\ \frac{\Gamma, l =_L \mathsf{Undef} \vdash_{\Downarrow} e : \mathsf{Undef}}{\Gamma \vdash_{\Uparrow} l\texttt{:}\ e \to \mathsf{Undef}}$$

$$\textsc{T-Break}\ \frac{l =_L T \in \Gamma \qquad \Gamma \vdash_{\Downarrow} e : T}{\Gamma \vdash_{\Uparrow} \textbf{break}\ l\, e \to \bot}$$

$$\textsc{T-Object}\ \frac{abs = \texttt{/.*/} \setminus \bigcup_i n_i \qquad \forall i.\, \Gamma \vdash_{\Uparrow} e_i \to T_i \qquad T = \{\!\{ abs : \mathsf{Absent}, \overrightarrow{n_i : T_i} \}\!\}}{\Gamma \vdash_{\Uparrow} \{\ \overrightarrow{n : e}\ \} \to T}$$

$$\textsc{T-Bracket}\ \frac{\Gamma \vdash_{\Uparrow} fld \to r \qquad \Gamma \vdash_{\Uparrow} obj \to \{\!\{ \vec{f} \}\!\} \qquad \Gamma \vdash T \Leftarrow \{\!\{ \vec{f} \}\!\}@r}{\Gamma \vdash_{\Uparrow} obj[fld] \to T}$$

$$\textsc{T-Update}\ \frac{\Gamma \vdash_{\Uparrow} obj \to T \qquad \Gamma \vdash_{\Uparrow} fld \to r \qquad T = \{\!\{ hid : \mathsf{Hidden}, abs : \mathsf{Absent}, \overrightarrow{r_f : T_f} \}\!\} \qquad r \cap (hid \cup abs) = \emptyset \qquad \forall r_f : r \cap r_f \neq \emptyset.\, \Gamma \vdash_{\Downarrow} e : T_f}{\Gamma \vdash_{\Uparrow} obj[fld\texttt{=}e] \to T}$$

$$\textsc{T-Delete}\ \frac{\Gamma \vdash_{\Uparrow} obj \to T \qquad T = \left\{\!\!\left\{ \begin{array}{l} hid : \mathsf{Hidden}, abs : \mathsf{Absent}, \\ \overrightarrow{f_p : !\, T_p}, \overrightarrow{f_i : \hat{}\, T_i}, \overrightarrow{f_m : ?\, T_m} \end{array} \right\}\!\!\right\} \qquad \Gamma \vdash_{\Uparrow} fld \to r \qquad r \subseteq \left(\bigcup_i f_{i_i} \right)}{\Gamma \vdash_{\Uparrow} \textbf{delete}\ obj[fld] \to T}$$

$$\textsc{T-TryCatch}\ \frac{\Gamma \vdash_{\Uparrow} e_1 \to T_1 \qquad \Gamma, x : \top \vdash_{\Uparrow} e_2 \to T_2}{\Gamma \vdash_{\Uparrow} \textbf{try}\ e_1\ \textbf{catch}(x)\ e_2 \to T_1 \cup T_2} \qquad \textsc{T-TryFinally}\ \frac{\Gamma \vdash_{\Uparrow} e_1 \to T_1 \qquad \Gamma \vdash_{\Uparrow} e_2 \to T_2}{\Gamma \vdash_{\Uparrow} \textbf{try}\ e_1\ \textbf{finally}\ e_2 \to T_2}$$

$$\textsc{T-Throw}\ \frac{\Gamma \vdash_{\Uparrow} e \to T}{\Gamma \vdash_{\Uparrow} \textbf{throw}\ e \to \bot}$$

$$\textsc{T-App-Fixed}\ \frac{\begin{array}{c} \Gamma \vdash_{\Uparrow} f \to [T_{this}]\, \overrightarrow{T_a} \ \to T_{ret} \qquad \Gamma \vdash_{\Downarrow} recv : T_{this} \\ \forall 1 \leq i \leq \min\left(\left|\vec{a}\right|, \left|\overrightarrow{T_a}\right|\right).\, \Gamma \vdash_{\Downarrow} a_i : T_{a_i} \\ \forall \left|\overrightarrow{T_a}\right| < i \leq \left|\vec{a}\right|.\, \Gamma \vdash \mathsf{Undef} <: T_{a_i} \end{array}}{\Gamma \vdash_{\Uparrow} f(recv, \vec{a}) \to T_{ret}}$$

$$\textsc{T-App-Var}\ \frac{\begin{array}{c} \Gamma \vdash_{\Uparrow} f \to [T_{this}]\, \overrightarrow{T_a} \times T_{var} \ldots \ \to T_{ret} \qquad \Gamma \vdash_{\Downarrow} recv : T_{this} \\ \forall 1 \leq i \leq \min\left(\left|\overrightarrow{T_a}\right|, \left|\vec{a}\right|\right).\, \Gamma \vdash_{\Downarrow} a_i : T_{a_i} \\ \forall \left|\overrightarrow{T_a}\right| < i \leq \left|\vec{a}\right|.\, \Gamma \vdash_{\Downarrow} a_i : T_{var} \qquad \forall \left|\vec{a}\right| < i \leq \left|\overrightarrow{T_a}\right|.\, \Gamma \vdash \mathsf{Undef} <: T_{a_i} \end{array}}{\Gamma \vdash_{\Uparrow} f(recv, \vec{a}) \to T_{ret}}$$

$$\textsc{T-IfTrue}\ \frac{\Gamma \vdash_{\Downarrow} c : \mathsf{True} \qquad \Gamma \vdash_{\Uparrow} e_1 \to T}{\Gamma \vdash_{\Uparrow} \textbf{if}\ (c)\ e_1\ \textbf{else}\ e_2 \to T} \qquad \textsc{T-IfFalse}\ \frac{\Gamma \vdash_{\Downarrow} c : \mathsf{False} \qquad \Gamma \vdash_{\Uparrow} e_2 \to T}{\Gamma \vdash_{\Uparrow} \textbf{if}\ (c)\ e_1\ \textbf{else}\ e_2 \to T}$$

$$\textsc{T-If}\ \frac{\Gamma \vdash_{\Downarrow} c : \mathsf{Bool} \qquad \Gamma \vdash_{\Uparrow} e_1 \to T_1 \qquad \Gamma \vdash_{\Uparrow} e_2 \to T_2}{\Gamma \vdash_{\Uparrow} \textbf{if}\ (c)\ e_1\ \textbf{else}\ e_2 \to T_1 \cup T_2}$$

$$\textsc{T-PrefixOp}\ \frac{op_p : T_e \ \to T \in \Gamma \qquad \Gamma \vdash_{\Downarrow} e : T_e}{\Gamma \vdash_{\Uparrow} op_p\, e \to T} \qquad \textsc{T-InfixOp}\ \frac{op_i : T_1 \times T_2 \ \to T \in \Gamma \qquad \Gamma \vdash_{\Downarrow} e_1 : T_1 \qquad \Gamma \vdash_{\Downarrow} e_2 : T_2}{\Gamma \vdash_{\Uparrow} e_1\, op_i\, e_2 \to T}$$

$$\textsc{T-Box-Num}\ \frac{\Gamma \vdash_{\Uparrow} e \to \mathsf{Num} \qquad \Gamma \vdash \mathsf{Number} \rightsquigarrow \mathsf{Ref}\, T}{\Gamma \vdash_{\Uparrow} e \to T} \qquad \textsc{T-Box-Str}\ \frac{\Gamma \vdash_{\Uparrow} e \to r \qquad \Gamma \vdash \mathsf{String} \rightsquigarrow \mathsf{Ref}\, T}{\Gamma \vdash_{\Uparrow} e \to T}$$

$$\textsc{T-App-Inter1}\ \frac{\Gamma \vdash_{\Uparrow} f \to T_1 \cap T_2 \qquad \Gamma \vdash_{\Uparrow} (\textbf{check}\ f\, T_1)(\vec{a}) \to T}{\Gamma \vdash_{\Uparrow} f(\vec{a}) \to T} \qquad \textsc{T-App-Inter2}\ \frac{\begin{array}{c} \Gamma \vdash_{\Uparrow} f \to T_1 \cap T_2 \qquad \Gamma \vdash_{\Uparrow} (\textbf{check}\ f\, T_2)(\vec{a}) \to T \\ \neg \exists T', \Gamma \vdash_{\Downarrow} (\textbf{check}\ f\, T_1)(\vec{a}) : T' \end{array}}{\Gamma \vdash_{\Uparrow} f(\vec{a}) \to T}$$

$$\textsc{T-Rec}\ \frac{\Gamma' = \Gamma, x_1 : T_1, \cdots, x_n : T_n \qquad \forall i.\, \Gamma' \vdash_{\Downarrow} e_i : T_i \qquad \Gamma' \vdash_{\Uparrow} b \to T_b}{\Gamma \vdash_{\Uparrow} \textbf{letrec}\ \overrightarrow{x : T = e}\ \textbf{in}\ b \to T_b}$$

$$\textsc{T-TypAbs}\ \frac{\Gamma, x <: T \vdash_{\Uparrow} e \to T'}{\Gamma \vdash_{\Uparrow} \Lambda x <: T\,.\, e \to \forall x <: T\,.\, T'} \qquad \textsc{T-TypApp}\ \frac{\Gamma \vdash_{\Uparrow} e \to T \qquad \Gamma \vdash T \rightsquigarrow \forall x <: S\,.\, T' \qquad \Gamma \vdash U <: S}{\Gamma \vdash_{\Uparrow} e\langle U \rangle \to T'[U/x]}$$

Figure 6: Type synthesis rules

SUBTYPING: $\Gamma \vdash S <: T$

S-REFL
$$\frac{}{\Gamma \vdash T <: T}$$

S-$\top$
$$\frac{}{\Gamma \vdash T <: \top}$$

S-$\bot$
$$\frac{}{\Gamma \vdash \bot <: T}$$

S-NULL-SRC
$$\frac{}{\Gamma \vdash \mathsf{Null} <: \mathsf{Src}\, T}$$

S-NULL-REF
$$\frac{}{\Gamma \vdash \mathsf{Null} <: \mathsf{Ref}\, T}$$

S-NULL-SNK
$$\frac{}{\Gamma \vdash \mathsf{Null} <: \mathsf{Sink}\, T}$$

S-SNK
$$\frac{\Gamma \vdash T <: S}{\Gamma \vdash \mathsf{Sink}\, S <: \mathsf{Sink}\, T}$$

S-REFSNK
$$\frac{\Gamma \vdash T <: S}{\Gamma \vdash \mathsf{Ref}\, S <: \mathsf{Sink}\, T}$$

S-REF
$$\frac{\Gamma \vdash T <: S \wedge \Gamma \vdash S <: T}{\Gamma \vdash \mathsf{Ref}\, S <: \mathsf{Ref}\, T}$$

S-REFSRC
$$\frac{\Gamma \vdash S <: T}{\Gamma \vdash \mathsf{Ref}\, S <: \mathsf{Src}\, T}$$

S-SRC
$$\frac{\Gamma \vdash S <: T}{\Gamma \vdash \mathsf{Src}\, S <: \mathsf{Src}\, T}$$

S-UNION-L
$$\frac{\Gamma \vdash S_1 <: T \wedge \Gamma \vdash S_2 <: T}{\Gamma \vdash S_1 \cup S_2 <: T}$$

S-UNION-R
$$\frac{\Gamma \vdash S <: T_1 \vee \Gamma \vdash S <: T_2}{\Gamma \vdash S <: T_1 \cup T_2}$$

S-INTER-R
$$\frac{\Gamma \vdash S <: T_1 \wedge \Gamma \vdash S <: T_2}{\Gamma \vdash S <: T_1 \cap T_2}$$

S-INTER-L
$$\frac{\Gamma \vdash S_1 <: T \vee \Gamma \vdash S_2 <: T}{\Gamma \vdash S_1 \cap S_2 <: T}$$

S-ARR-FIXEDFIXED
$$\frac{\begin{array}{c}|S_a| = |T_a| \qquad \Gamma \vdash S_r <: T_r \\ (S_{this} = \mathsf{Ref}\, S_t \vee S_{this} = \mathsf{Src}\, S_t) \\ (T_{this} = \mathsf{Ref}\, T_t \vee T_{this} = \mathsf{Src}\, T_t) \\ \Gamma \vdash S_t <: T_t \qquad \forall i.\, \Gamma \vdash T_{a_i} <: S_{a_i}\end{array}}{\Gamma \vdash [S_{this}]\, \vec{S_a} \;\to\; S_r <: [T_{this}]\, \vec{T_a} \;\to\; T_r}$$

S-ARR-FIXEDVAR
$$\frac{\begin{array}{c}|S_a| \geq |T_a| \qquad \Gamma \vdash S_r <: T_r \\ (S_{this} = \mathsf{Ref}\, S_t \vee S_{this} = \mathsf{Src}\, S_t) \\ (T_{this} = \mathsf{Ref}\, T_t \vee T_{this} = \mathsf{Src}\, T_t) \qquad \Gamma \vdash S_t <: T_t \\ \forall 1 \leq i \leq |T_a|.\, \Gamma \vdash T_{a_i} <: S_{a_i} \\ \forall |T_a| < i \leq |S_a|.\, \Gamma \vdash T_v \cup \mathsf{Undef} <: S_{a_i}\end{array}}{\Gamma \vdash [S_{this}]\, \vec{S_a} \;\to\; S_r <: [T_{this}]\, \vec{T_a} \times T_v \ldots \;\to\; T_r}$$

S-ARR-VARFIXED
$$\frac{\begin{array}{c}|S_a| \leq |T_a| \qquad \Gamma \vdash S_r <: T_r \\ (S_{this} = \mathsf{Ref}\, S_t \vee S_{this} = \mathsf{Src}\, S_t) \\ (T_{this} = \mathsf{Ref}\, T_t \vee T_{this} = \mathsf{Src}\, T_t) \qquad \Gamma \vdash S_t <: T_t \\ \forall 1 \leq i \leq |S_a|.\, \Gamma \vdash T_{a_i} <: S_{a_i} \\ \forall |S_a| < i \leq |T_a|.\, \Gamma \vdash T_{a_i} <: S_v\end{array}}{\Gamma \vdash [S_{this}]\, \vec{S_a} \times S_v \ldots \;\to\; S_r <: [T_{this}]\, \vec{T_a} \;\to\; T_r}$$

S-ARR-VARVAR
$$\frac{\begin{array}{c}(S_{this} = \mathsf{Ref}\, S_t \vee S_{this} = \mathsf{Src}\, S_t) \\ (T_{this} = \mathsf{Ref}\, T_t \vee T_{this} = \mathsf{Src}\, T_t) \qquad \Gamma \vdash S_t <: T_t \\ \Gamma \vdash S_r <: T_r \qquad \forall 1 \leq i \leq \min\left(|S_a|, |T_a|\right).\, \Gamma \vdash T_{a_i} <: S_{a_i} \\ \forall |S_a| < i \leq |T_a|.\, \Gamma \vdash T_{a_i} <: S_v \\ \forall |T_a| < i \leq |S_a|.\, \Gamma \vdash T_v \cup \mathsf{Undef} <: S_{a_i}\end{array}}{\Gamma \vdash [S_{this}]\, \vec{S_a} \times S_v \ldots \;\to\; S_r <: [T_{this}]\, \vec{T_a} \times T_v \ldots \;\to\; T_r}$$

S-μ-R
$$\frac{\Gamma \vdash S <: T[\mu\alpha\,.\,T/\alpha]}{\Gamma \vdash S <: \mu\alpha\,.\,T}$$

S-μ-L
$$\frac{\Gamma \vdash S[\mu\alpha\,.\,S/\alpha] <: T}{\Gamma \vdash \mu\alpha.S <: T}$$

S-TYVAR
$$\frac{\alpha <: S \in \Gamma \qquad \Gamma \vdash S <: T}{\Gamma \vdash \alpha <: T}$$

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

S-Λ
$$\frac{\Gamma, x <: \top, y <: \top \vdash S <: T}{\Gamma \vdash \Lambda x :: \star\,.\,S <: \Lambda y :: \star\,.\,T}$$

S-REGEX
$$\frac{r_1 \subseteq r_2}{\Gamma \vdash r_1 <: r_2}$$

S-OBJ
$$\frac{\begin{array}{c}\forall i, j.\, \Gamma \vdash f_{S_i} <:_f f_{T_j} \\ \forall i : f_{T_i} = r :\hat{}\, T_T.\ \left(\Gamma \vdash T_S \Leftarrow \{\!\{\vec{f_S}\}\!\}@r \implies \Gamma \vdash T_S <: T_T\right)\end{array}}{\Gamma \vdash \{\!\{\vec{f_S}\}\!\} <: \{\!\{\vec{f_T}\}\!\}}$$

OBJECT-FIELD SUBTYPING: $\Gamma \vdash f_1 <:_f f_2$

F-PRESENT
$$\frac{\begin{array}{c}r_1 \cap r_2 \neq \emptyset \\ \Gamma \vdash T_1 <: T_2\end{array}}{\Gamma \vdash r_1 :! T_1 <:_f r_2 :! T_2}$$

F-MAYBE
$$\frac{\begin{array}{c}r_1 \cap r_2 \neq \emptyset \\ \Gamma \vdash T_1 <: T_2\end{array}}{\Gamma \vdash r_1 :? T_1 <:_f r_2 :? T_2}$$

F-INHERIT
$$\frac{\begin{array}{c}r_1 \cap r_2 \neq \emptyset \\ \Gamma \vdash T_1 <: T_2\end{array}}{\Gamma \vdash r_1 :\hat{}\, T_1 <:_f r_2 :\hat{}\, T_2}$$

F-ABSENT
$$\frac{r_1 \cap r_2 \neq \emptyset}{\Gamma \vdash r_1 : \mathsf{Absent} <:_f r_2 : \mathsf{Absent}}$$

F-HIDDEN
$$\frac{}{\Gamma \vdash f <:_f r_2 : \mathsf{Hidden}}$$

F-PRESENTMAYBE
$$\frac{r_1 \cap r_2 \neq \emptyset \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash r_1 :! T_1 <:_f r_2 :? T_2}$$

F-PRESENTINHERIT
$$\frac{r_1 \cap r_2 \neq \emptyset \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash r_1 :! T_1 <:_f r_2 :\hat{}\, T_2}$$

F-ABSENTMAYBE
$$\frac{r_1 \cap r_2 \neq \emptyset}{\Gamma \vdash r_1 : \mathsf{Absent} <:_f r_2 :? T_2}$$

F-ABSENTINHERIT
$$\frac{r_1 \cap r_2 \neq \emptyset \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash r_1 : \mathsf{Absent} <:_f r_2 :\hat{}\, T_2}$$

F-DISJOINT
$$\frac{r_1 \cap r_2 = \emptyset}{\Gamma \vdash r_1 <:_f r_2}$$

Figure 7: Subtyping rules