Dependent Types for JavaScript

Ravi Chugh
University of California, San Diego
rchugh@cs.ucsd.edu

David Herman
Mozilla Research
dherman@mozilla.com

Ranjit Jhala
University of California, San Diego
jhala@cs.ucsd.edu

Abstract

We present Dependent JavaScript (DJS), a statically-typed dialect of the imperative, object-oriented, dynamic language. DJS supports the particularly challenging features such as run-time type-tests, higher-order functions, extensible objects, prototype inheritance, and arrays through a combination of *nested refinement types*, *strong updates* to the heap, and *heap unrolling* to precisely track prototype hierarchies. With our implementation of DJS, we demonstrate that the type system is expressive enough to reason about a variety of tricky idioms found in small examples drawn from several sources, including the popular book *JavaScript: The Good Parts* and the SunSpider benchmark suite.

1. Introduction

Dynamic languages like JavaScript, Python, and Ruby are widely popular for building both client and server applications, in large part because they provide powerful sets of features — run-time type tests, mutable variables, extensible objects, and higher-order functions. But as applications grow, the lack of static typing makes it difficult to achieve reliability, security, maintainability, and performance. In response, several authors have proposed type systems which provide static checking for various subsets of dynamic languages [5, 15, 22, 23, 30, 36].

Recently, we developed System D [8], a core calculus for dynamic languages that supports the above dynamic idioms but in a purely functional setting. The main insight in System D is to *dependently* type all values with formulas drawn from an SMT-decidable refinement logic. We use an SMT solver to reason about the properties it tracks well, namely, control-flow invariants and dictionaries with dynamic keys that bind scalar values. But to describe dynamic keys that bind rich values like functions, System D encodes function types as logical terms and *nests* the typing relation as an uninterpreted predicate within the logic. By dividing work between syntactic subtyping and SMT-based validity checking, the calculus supports fully automatic checking of dynamic features like run-time type tests, value-indexed dictionaries, higher-order functions, and polymorphism.

In this paper, we scale up the System D calculus to Dependent JavaScript (abbreviated to DJS), an explicitly typed dialect of a real-world, imperative, object-oriented, dynamic language. We bridge the vast gap between System D and JavaScript in three steps.

Step 1: Imperative Updates. The types of variables in JavaScript are routinely "changed" either by assignment or by incrementally adding or removing fields to objects bound to variables. The presence of mutation makes it challenging to assign precise types to variables, and the standard method of assigning a single "invariant" reference type that overapproximates all values held by the variable is useless in the JavaScript setting. We overcome this challenge by extending our calculus with *flow-sensitive heap types* (in the style of [2, 12, 14, 31, 32]) which allow the system to precisely track the heap location each variable refers to as well as aliasing relationships, thereby enabling *strong updates* through mutable variables. Our formulation of flow-sensitive heaps combined with higher-order functions and refinement types is novel, and allows DJS to express precise pre- and post-conditions of heaps, as in separation logic [16].

Step 2: Prototype Inheritance. Each JavaScript object maintains an implicit link to the "prototype" object from which it derives. To resolve a key lookup from an object at run-time, JavaScript *transitively* follows its prototype links until either the key is found or the root is reached without success. Thus, unlike in class-based languages, inheritance relationships are *computed* at run-time rather than provided as declarative specifications. The semantics of prototypes is challenging for static typing, because to track the type of a key binding, the system must statically reason about a potentially unbounded number of prototype links! In DJS, we solve this problem with a novel decomposition of the heap into a "shallow" part, for which we precisely track a finite number of prototype links, and a "deep" part, for which we do not have precise information, represented abstractly via a logical heap variable. We *unroll* prototype hierarchies in shallow heaps to precisely model the semantics of object operations, and we use *uninterpreted heap predicates* to reason abstractly about deep parts. In this way, we reduce the reasoning about unbounded, imperative, prototype hierarchies to the underlying decidable, first-order, refinement logic.

Step 3: Arrays. JavaScript arrays are simply objects whose keys are string representations of integers. Arrays are commonly used both as *heterogenous* tuples (that have a fixed number of elements of different types) as well as *homogenous* collections (that have an unbounded number of elements of the same type). The overloaded use of arrays, together with the fact that arrays are otherwise syntactically indistinguishable and have the same prototype-based semantics as non-array objects, makes it hard to statically reason about the very different ways in which they are used. In DJS, we use nested refinements to address the problem neatly by uniformly encoding tuples and collections with refinement predicates, and by using *intersection* types that simultaneous encode the semantics of tuples, collections, and objects.

Expressiveness of DJS. We demonstrate the expressiveness of DJS by using our implementation[1] to check a variety of properties found in small but subtle examples drawn from a variety of sources, including the popular book *JavaScript: The Good Parts* [10] and the SunSpider benchmark suite [34]. Our experiments show that several examples *simultaneously* require the gamut of features in DJS, but that many examples conform to recurring patterns that rely on particular aspects of the type system. We identify several ways in which future work can handle these patterns more specifically in order to reduce the annotation burden and performance for common cases, while falling back to the full expressiveness of DJS in general. Thus, we believe that DJS provides a significant step towards truly retrofitting JavaScript with a practical type system.

2. Overview

Let us begin with an informal overview of the semantics of JavaScript. We will emphasize the aspects that are the most distinctive and challenging from the perspective of type system design, and describe the key insights in our work that overcome these challenges.

JavaScript Semantics by Desugaring. Many corner cases of JavaScript are clarified by λ_{JS} [21], a syntax-directed translation, or *desugaring*, of JavaScript programs to a mostly-standard lambda-calculus with explicit references. As λ_{JS} is a core language with well-understood semantics and proof techniques, the translation paves a path to a typed dialect of JavaScript: define a type system for the core language and then type check desugared JavaScript programs.

We take this path by developing System !D (pronounced "D-ref"), a new calculus based on λ_{JS}. Although the operational semantics of System !D is straightforward, the *dynamic* features of the language ensure that building a type system expressive enough to support desugared JavaScript idioms is not. We solve this problem by scaling the purely functional technique of nested refinement types up to the imperative, object-oriented, setting of real-world JavaScript.

Terminology. JavaScript has a long history and an evolving specification. In this paper, we say "JavaScript" to roughly mean ECMAScript Edition 3, the standard version of the language for more than a decade [24]. We say ES5 to refer to Edition 5 of the language,[2] recently released by the JavaScript standards committee [9]. We say ES6 to refer to features proposed for the next version of the language, scheduled to be finalized within the next one or two years. Dependent JavaScript (DJS) includes a large set of core features common to all editions.

2.1 Base Types, Operators, and Control Flow

Consider the following function adapted from [8] and annotated in DJS. A function type annotation is written just above the definition inside a JavaScript comment demarcated by an additional : character. We typeset annotations in math mode for clarity, but the ASCII versions parsed by our type checker are quite similar.

```
/*: x:Top → {ν|ite Num(x) Num(ν) Bool(ν)} */
function negate(x) {
  if (typeof x == "number") { return 0 - x; }
  else                      { return !x;    }  }
```

The `typeof` operator is a facility used pervasively to direct control flow based on the run-time "tag" of a value. If the input to `negate` is a number, so is the return value. If not, the function uses an interesting feature of JavaScript, namely, that *all* values have a boolean interpretation. The values `false`, `null`, `undefined`, the empty string "", `0`, and the "not-a-number" value `NaN` are considered *falsy*, and evaluate to `false` when used in a boolean context; all other values are *truthy*. The operator `!` inverts "truthiness," so the else branch returns a boolean no matter what the type of `x` is. The ability to treat arbitrary values as booleans is commonly used, for example, to guard against non-`null` values.

The `negate` function demonstrates that even simple JavaScript programs depend heavily on sophisticated control-flow based reasoning. Syntactic type systems are capable of tracking control flow to a limited degree [22, 36], but none can handle complex invariants like the relationship between the input and output of `negate`. To have any chance of capturing such invariants, types must be able to *depend* on other program values. Powerful dependent type systems like Coq can express extremely rich invariants, but are too heavyweight for our goals since they require the programmer to discharge type checking obligations interactively.

Refinement Types. We adopt a more lightweight mechanism called *refinement types* that has been previously applied to purely functional dynamic languages [5, 8]. We demonstrate that refinement types afford us the expressiveness needed to precisely track control-flow invariants in the JavaScript setting and, unlike more powerful dependent systems, without sacrificing decidable type checking. In particular, once the programmer has written type annotations for

[1] Available at `ravichugh.com/nested`.

[2] Edition 4 was never standardized.

```
val typeof :: (* x:Top → {ν = tag(x)} *)
val      ! :: (* x:Top → {ν iff falsy(x)} *)
val   (||) :: (* x:Top → y:Top → {ite falsy(x) (ν = y) (ν = x)} *)
val   (&&) :: (* x:Top → y:Top → {ite truthy(x) (ν = y) (ν = x)} *)
val  (===) :: (* x:Top → y:{tag(ν) = tag(x)} → {ite (x = NaN ∨ y = NaN) (ν = false) (ν iff x = y)} *)
val   (==) :: (* x:Top → y:Top → {Bool(ν) ∧ (tag(x) = tag(y) ⇒ ν iff (x = y ∧ x ≠ NaN))} *)
val    (+) :: (* x:Str → y:Str → Str *)
val    (+) :: (* x:Num → y:Num → {Num(ν) ∧ ((Int(x) ∧ Int(y)) ⇒ (Int(ν) ∧ ν = x + y))} *)
val    fix :: (* ∀A. (A → A) → A *)
```

Figure 1. Excerpt from `basics.dref`

function definitions, type checking is carried out automatically via a combination of syntactic subtyping and SMT-based [11] logical validity checking.

In System !D, every value is described by a refinement type of the form $\{\nu \mid p\}$, read "ν such that p", where p is a formula that can mention ν. For example, 3 can be given the type $\{\nu \mid tag(\nu) = \text{“}\texttt{number}\text{”}\}$ and `true` the type $\{\nu \mid tag(\nu) = \text{“}\texttt{boolean}\text{”}\}$, where tag is an uninterpreted function symbol in the refinement logic, *not* a function in the programming language. We use the following abbreviations to make the refinement binder implicit and the types concise.

$$\begin{array}{rclrcl} \{p\} & \mathrel{\mathring{=}} & \{\nu \mid p\} & Num(x) & \mathrel{\mathring{=}} & tag(x) = \text{“}\texttt{number}\text{”} \\ Top(x) & \mathrel{\mathring{=}} & true & Bool(x) & \mathrel{\mathring{=}} & tag(x) = \text{“}\texttt{boolean}\text{”} \\ T & \mathrel{\mathring{=}} & \{T(\nu)\} & Str(x) & \mathrel{\mathring{=}} & tag(x) = \text{“}\texttt{string}\text{”} \end{array}$$

$$\mathbf{if}\ p\ \mathbf{then}\ q_1\ \mathbf{else}\ q_2 \mathrel{\mathring{=}} \mathbf{ite}\ p\ q_1\ q_2 \mathrel{\mathring{=}} (p \Rightarrow q_1) \wedge (\neg p \Rightarrow q_2)$$

$$x\ \mathbf{iff}\ p \mathrel{\mathring{=}} \mathbf{ite}\ p\ (x = \texttt{true})\ (x = \texttt{false})$$

$$falsy(x) \mathrel{\mathring{=}} x \in \{\texttt{false} \vee \texttt{null} \vee \texttt{undefined} \vee \text{“”} \vee 0 \vee \texttt{NaN}\}$$

$$truthy(x) \mathrel{\mathring{=}} \neg falsy(x)$$

Primitives. We use refinements to assign precise, and sometimes exact, types to System !D primitive functions, defined in the file `basics.dref` (Figure 1). Notice that `typeof` returns the tag of its input. Some examples beyond ones we have already seen include $tag(\texttt{null}) = \text{“}\texttt{object}\text{”}$ and $tag(\texttt{undefined}) = \text{“}\texttt{undefined}\text{”}$. The type of the negation operator `!` inverts "truthiness." The types of the operators `&&` and `||` are interesting, because as in JavaScript, they do not necessarily return booleans. The "guard" operator `&&` returns its second operand if the first is truthy, which enables the idiom `if (x && x.f) { ... }` that checks whether the object `x` and its "`f`" field are non-null. Dually, the "default" operator `||` returns its second operand if the first is falsy, which enables the idiom `x = x || default` to specify a default value. The `+` operator is specified as an *intersection* of function types and captures the fact that it performs both string concatenation and numerical addition, but does *not* type check expressions like 3 + "`hi`" that rely on the *implicit coercion* in JavaScript. We choose types for System !D primitives that prohibit implicit coercions since they often lead to subtle programming errors.

Equality. JavaScript provides two equality operators: `==` implicitly coerces the second operand if its tag differs from the first, and strict equality `===` does not perform any coercion.[3] To avoid reasoning about implicit coercions, we give a relatively weaker type to `==`, where the boolean result relates its operands *only if* they have the same tag.

Integers. JavaScript provides a single number type that has no minimum or maximum value. However, programmers and optimizing JIT compilers [38] often distinguish integers from arbitrary numbers. In System !D, we describe integers with the abbreviation $Int(x) \mathrel{\mathring{=}} Num(x) \wedge integer(x)$. We introduce the uninterpreted predicate $integer(x)$ in the types of integer literals, and functions like `+` propagate "integer-ness" where possible. Furthermore, numeric functions use the (decidable) theory of linear arithmetic to precisely reason about integers, which is important for dealing with arrays.

Tracking Control Flow. System !D precisely tracks control flow by recording that the guard of an if-expression is truthy (resp. falsy) while checking the then-branch (resp. else-branch), enabling System !D to verify the annotation for `negate` as follows. Because of the call to `typeof`, System !D tracks that $Num(\texttt{x})$ holds along the then-branch, so `x` can be safely passed to the subtraction operator which produces a number as required. For the else-branch, System !D records that $\neg Num(\texttt{x})$. The negation operator, which can be applied to any value, produces a value of type $\{\nu\ \mathbf{iff}\ \neg falsy(\texttt{x})\}$ which is a subtype of $Bool$. Thus, both branches satisfy the specification provided by the programmer.

2.2 Imperative Updates

JavaScript is an imperative language where variables can be reassigned arbitrary values. Consider the DJS function `also_negate` in Figure 2 that is like `negate` but first assigns the eventual result in the variable `x`, and its translation to System !D on the right (ignore the comments for now).

Several aspects of the translation warrant attention. First, since the formal parameter `x`, like all JavaScript variables, is mutable, the translation of the function body begins with an *explicit reference* cell `_x` initialized with `x`, and each

[3] Every value except `NaN` is strictly equal to itself.

```
function also_negate(x) {

 if (typeof x == "number")
   x = 0 - x;
 else
   x = !x;

 return x;
}
```

```
1 let also_negate = fun x ->                   (* Γ1 = ∅;  Σ1 = ∅ *)
2  let _x = ref x in                           (* Γ2 = x:Top;  Σ2 = (ℓx ↦ x) *)
3  if typeof (deref _x) == "number" then       (* Γ3 = Γ2, _x:Ref ℓx;  Σ3 = Σ2 *)
4   _x := 0 - (deref _x)      (* Γ4 = Γ3, Num(x);  Σ4 = ∃x4:Num. (ℓx ↦ x4) *)
5  else
6   _x := !(deref _x)  (* Γ6 = Γ3, ¬Num(x);  Σ6 = ∃x6:{ν iff falsy(x)}. (ℓx ↦ x6) *)
7  ;               (* Γ7 = Γ3;  Σ7 = ∃x′:{ite Num(x) Num(ν) Bool(ν)}. (ℓx ↦ x′) *)
8  deref _x in     (* Γ8 = Γ3;  Σ8 = Σ7 *)
9 let _also_negate = ref {"__code__": also_negate}
```

Figure 2. DJS function `also_negate`; Desugared to System !D; Verifying $x\!:\!Top \to \{\mathbf{ite}\ Num(x)\ Num(\nu)\ Bool(\nu)\}$

read of `x` is desugared to a dereference of `_x`.[4] Second, notice that scalar constants like `0` and `true` and operators like `typeof` and `==` are translated directly to corresponding ones in System !D. Third, notice that each assignment to the JavaScript variable `x` translates to a set reference (*i.e.* assignment) operation to update the contents of the heap cell. Finally, since every JavaScript function is actually an object, the translation stores the function value in a distinguished "`__code__`" field, which we assume is inaccessible to source programs.[5] For System !D to verify that `also_negate` satisfies the specification, it must precisely reason about heap updates in addition to control-flow as before.

Reference Types. The traditional way to handle references in the λ-calculus [28] is to (a) assign a reference cell some type $Ref\,T$, (b) require that only values of type T be stored in it, and then (c) conclude that dereferences produce values of type T. This approach supports so-called *weak updates*, because even if a stored value satisfies a stronger type S than T (*i.e.* if S is a subtype of T), subsequent dereferences produce values of the original, weaker type T. Put another way, this approach requires that the type assigned to a reference cell be a supertype of all the values written to the cell. Unfortunately, weak updates would preclude System !D from verifying `also_negate`. The initialization of `_x` on line 2 stores the parameter `x` which has type Top, so `_x` would be assigned type $Ref\ Top$. The assignments on lines 4 and 6 type check because the updated values satisfy the trivial type Top, but the dereference on line 8 produces a value with type Top, which does *not* satisfy the specified return type. Thus, we need a way to reason more precisely about heap updates.

Strong Updates. Allowing assignment to *change* the type of a reference is called *strong update*, which is sound only when a reference is guaranteed to point to a *single* heap cell and when there accesses through other *aliases* that refer to the same cell. The *Alias Types* approach [32] provides a means of managing these concerns. Rather than $Ref\,T$, a reference type is written $Ref\,\ell$, where ℓ is the (compile-time) name of a location in the heap, and a separate (compile-time) heap maps locations to types, for example, $(\ell \mapsto T)$. Strong updates are realized by allowing heaps to change *flow-sensitively*, and the aliasing problem is mitigated by maintaining the invariant that distinct location names ℓ and ℓ' do not alias. System !D employs this approach by using a *type environment* Γ that grows and shrinks as usual during type checking but remains flow-insensitive, and a *heap environment* Σ that can be strongly updated per program point.

Figure 2 shows how System !D checks the desugared version of `also_negate`. The figure shows, at each line i, the type environment Γ_i used to check the expression on the line, and the heap environment Σ_i that exists *after* checking the expression. After starting with the empty heap $\Sigma_1 = \varnothing$, the allocation on line 2 creates a fresh location ℓ_x in the new heap $\Sigma_2 \mathrel{\hat{=}} \Sigma_1 \oplus (\ell_x \mapsto \mathtt{x})$ and adds $\_x\!:\!Ref\,\ell_x$ to the type environment. We use the symbol $\oplus$ to construct unordered sets of heap bindings. To exploit the precision of dependent types, we map locations to *values* rather than types (*i.e.* $(\ell \mapsto \mathtt{x})$ rather than $(\ell \mapsto Top)$).

When checking the if-expression guard on line 3, the deference retrieves the initial value `x` from the heap Σ_2, so as a result of the tag-test, System !D adds $Num(\mathtt{x})$ to the type environment Γ_4 along the true-branch and $\neg Num(\mathtt{x})$ to Γ_6 along the false-branch. In the true-branch, the subtraction on line 4 is well-typed because $Num(\mathtt{x})$, and produces a number x_4 that is stored in the heap Σ_4 at location ℓ_x. In the false-branch, `x` is negated on line 6, producing a boolean x_6 that is stored in the heap Σ_6 at location ℓ_x. System !D combines the branches by *joining* the heaps Σ_4 and Σ_6, producing Σ_7 that describes the heap no matter which branch is taken. The dereference on line 8 retrieves x', a value of type $\{\mathbf{ite}\ Num(x)\ Num(\nu)\ Bool(\nu)\}$, as required by the return type annotation.

In this way, System !D syntactically tracks strong updates to the heap, while reducing subtyping obligations to implication queries in an ordinary, pure refinement logic [8] that does *not* model imperative updates.[6]

[4] Presentations of imperative languages often model assignable variables directly rather than with explicit references. Both approaches are equivalent in expressiveness as well as challenges for type checking; we choose the latter to make the presentation more similar to λ_{JS} [21] and System D [8].

[5] To eliminate the assumption, we could instead treat each function as a pair of a function value and an associated object. This approach poses no technical difficulty, but we follow the λ_{JS} encoding for simplicity.

[6] Existentials are not problematic as they only appear during checking [25].

2.3 Simple Objects

JavaScript's objects exhibit several interesting semantic properties. Consider the following object initialized with a single key (also known as field or property). We assume that `assert` is a pre-defined function that aborts when its argument is falsy; JavaScript does not provide such a function as built-in, but it is trivial to define.

```
var x = {"f": 1};
assert (x.f == 1 && x.g == undefined);
x.g = 2; delete x.f;
assert (x.g == 2 && x.f == undefined);
x.f.g; // raises exception
var k = "h"; x[k] = 3; assert (x[k] == 3);
```

Notice that when retrieving the non-existent "g" key from `x`, JavaScript returns `undefined` as opposed to raising an exception. Attempting to retrieve a key from `undefined`, or `null`, however, does raise an exception. Keys can be added or removed to objects, and can even be arbitrary *dynamically-computed* values, not just string literals, that are converted to strings if necessary. Dynamic keys are pervasive — objects are commonly used as hash tables with unknown sets of keys — but they are quite challenging to track inside a static type system.

Nested Refinements. To support dynamic keys, we adopt the System D primitives [8] for (functional) dictionary operations, shown in the first four lines of the file `objects.dref` (Figure 3). The primitive function application `get` d k retrieves the key k from dictionary d, where $sel(d,k)$ describes the exact binding as a value in the refinement logic; `set` d k y produces a new dictionary that extends d with a binding for k, shadowing previous bindings, if any, where $upd(d,k,y)$ describes the new dictionary; `del` d k produces a new dictionary with a binding removed, using the logical symbol bot (distinct from all source-level values) to denote its absence; and `mem` d k indicates the presence or absence of a binding, where we write the abbreviation $has(d,k) \mathrel{\mathring{=}} sel(d,k) \neq bot$.

The key innovation of *nested refinements* in System D allows *syntactic type terms* U (like function types) to be written within refinement formulas using an uninterpreted "has-type" predicate $x :: U$, while staying within the decidable McCarthy theory of arrays [27]. The has-type predicate allows System D to describe dictionaries that map *dynamic* keys to *arbitrary* values.[7] For example, we write $\{Dict(\nu) \wedge sel(\nu,k) :: Bool \to Bool\}$ to describe a dictionary d with key k that binds a boolean-to-boolean function, and $\{\nu = upd(d, \text{“g”}, 4)\}$ to describe a dictionary d' that is just like d but with an additional binding. We refer the reader to [8] for the technical development of nested refinements.

[7] Prior approaches such as [5] were limited to dynamic keys that store first-order (non-function) values.

Mutability and Dynamic Keys. The combination of nested refinements and strong updates allows us to precisely track objects with dynamic keys despite the presence of imperative updates. Consider the desugaring of our example above; we omit the assertions for clarity.

```
1 let _x = ref (ref {"f": 1}) in
2 _x := set (_deref (_deref _x)) "g" 2;
3 _x := del (_deref (_deref _x)) "f";
4 let _k = "h" in
5 _x := set (_deref (_deref _x))
6           (coerceToStr (deref _k)) 3;
```

The allocation on line 1 adds three bindings to the type environment — $d : \{\nu = upd(empty, \text{“f”}, 1)\}$, $ptr : Ref\,\ell$, and `_obj`$: Ref\,\ell'$, where ℓ and ℓ' are fresh locations — and produces the heap $\Sigma_1 \mathrel{\mathring{=}} (\ell' \mapsto ptr) \oplus (\ell \mapsto d)$. Notice that the dictionary is stored via an additional level of indirection to facilitate the encoding of *side-effecting* JavaScript object operations. The object extension on line 2 adds $d' : \{\nu = upd(d, \text{“g”}, 2)\}$ to the type environment and strongly updates the heap to $\Sigma_2 \mathrel{\mathring{=}} (\ell' \mapsto ptr) \oplus (\ell \mapsto d')$. The deletion on line 3 and the extension on line 5 (through a dynamic key) have similar effects on the static heap, thereby statically verifying the assertions.

2.4 Function Types

In order to fully understand the challenges of JavaScript objects, we must first pause to take a closer look at function types. The function types we have seen so far — for `negate` and the primitives in `basics.dref` — have not mentioned heaps, because their inputs and outputs are scalar values.

However, JavaScript objects are reference values, and are passed to and returned from functions through the heap. Thus, to account for heaps and side-effects, a System !D function type has the following form.

$$\forall [\overline{A}; \overline{L}; \overline{H}]\; x{:}T_1 / \hat{\Sigma}_1 \to T_2 / \hat{\Sigma}_2$$

This type describes a function that, given an argument of type T_1 in a calling context that satisfies the input heap type $\hat{\Sigma}_1$, produces an output value of type T_2 and a modified heap type $\hat{\Sigma}_2$. A function type can be parameterized by sequences of *type variables* A, *location variables* L, and *heap variables* H. A heap type $\hat{\Sigma}$ is like a heap environment Σ but maps locations to binder-type pairs rather than values (*e.g.* $(\ell \mapsto y{:}T)$ rather than $(\ell \mapsto v)$); the binders are useful for relating heaps across function boundaries. The binder x of the input type, as well as all of the binders in the input heap $\hat{\Sigma}_1$, are in scope in the output *world* $T_2/\hat{\Sigma}_2$. We often omit binders when they are not referred to. To match the structure of function types, function applications must instantiate type, location, and heap variables. However, our implementation infers instantiations in almost all cases using standard local type inference techniques (§ 6). When we write DJS examples in the sequel, we omit instantiations

```
val        set :: (* d:Dict → k:Str → y:Top → {ν = upd(d,k,y)} *)
val        del :: (* d:Dict → k:Str → {ν = upd(d,k,bot)} *)
val        has :: (* d:Dict → k:Str → {ν iff has(d,k)} *)
val        get :: (* d:Dict → k:Str → {ite has(d,k) (ν = sel(d,k)) (ν = undefined)} *)

val setPropObj :: (* (x:Ref, k:Str, y:Top)/(x ↦ ⟨d:Dict,ẋ⟩) → {ν = y}/(x ↦ ⟨d':{ν = upd(d,k,y)},ẋ⟩) *)
val delPropObj :: (* (x:Ref, k:Str)/(x ↦ ⟨d:Dict,ẋ⟩) → Bool/(x ↦ ⟨d':{ν = upd(d,k,bot)},ẋ⟩) *)
val hasPropObj :: (* (x:Ref, k:Str)/(x ↦ ⟨d:Dict,ẋ⟩) → {ν iff ObjHas(d,k,cur,ẋ)}/same *)
val getPropObj :: (* (x:Ref, k:Str)/(x ↦ ⟨d:Dict,ẋ⟩)
                    → {ite ObjHas(d,k,cur,ẋ) (ν = ObjSel(d,k,cur,ẋ)) (ν = undefined)}/same *)

val  getIdxArr :: (* ∀A. (x:Ref, i:Int)/(x ↦ ⟨a:Arr(A),ẋ⟩)
                    → {ite ¬packed(a) (ν :: A ∨ Undef(ν)) (ite (0 ≤ i < len(a)) (ν :: A) (Undef(ν)))}/same *)
val  getLenArr :: (* ∀A. (x:Ref, k:{ν = "length"})/(x ↦ ⟨a:Arr(A),ẋ⟩)
                    → {ite packed(a) (ν = len(a)) Int(ν)}/same *)
val getPropArr :: (* ∀A. (x:Ref, k:{Str(ν) ∧ ν ≠ "length"})/(x ↦ ⟨a:Arr(A),ẋ⟩)
                    → {ite HeapHas(H,ẋ,k) (ν = HeapSel(H,ẋ,k)) (ν = undefined)}/same *)

val    getElem :: (and (type getPropObj) (type getIdxArr) (type getLenArr) (type getPropArr))

val  setIdxArr :: (* ∀A. (x:Ref, i:Int, y:A)/(x ↦ ⟨a:Arr(A),ẋ⟩)
                    → {ν = y}/(x ↦ ⟨a':{ν :: Arr(A) ∧ arrSet(ν,a,i)},ẋ⟩) *)
```

Figure 3. Excerpt from `objects.dref`

at applications wherever our current implementation infers them. We sweeten function type syntax with some sugar:

- When used as an output heap, the token *same* refers to the sequence of locations in the corresponding input heap, where each binding records that the final value is exactly equal to the initial value.
- In an input world, a reference binding $x{:}Ref$ without a location introduces a location variable L that is quantified by the type, and x (a value of type $Ref\ L$) can be used as a location in heaps to refer to this variable L. Further, the dotted variable $\dot{x}$ introduces a location parameter, corresponding to the prototype of x.
- A heap variable H is implicitly added to a function type when it contains none, and H is added to both the input and output heaps.[8] In this case, the token *cur* refers to H.

For example, compare the type for `hasPropObj` (Figure 3) followed by its expansion.

$$\begin{aligned}&(x{:}Ref,\ k{:}Str)/(x \mapsto \langle d{:}Dict, \dot{x}\rangle)\\ \rightarrow\ &\{\nu \textbf{ iff } ObjHas(d,k,cur,\dot{x})\}/same\end{aligned}$$

$$\begin{aligned}&\forall L, L', H.\ (x{:}Ref\, L,\ k{:}Str)/H \oplus (L \mapsto \langle d{:}Dict, L'\rangle)\\ \rightarrow\ &\{\nu \textbf{ iff } ObjHas(d,k,H,L')\}/H \oplus (L \mapsto \langle d'{:}\{\nu = d\}, L'\rangle)\end{aligned}$$

2.5 Prototype-Based Objects

JavaScript sports a special form of inheritance, where each base object is equipped with a link to its *prototype object*. This link is set when the base object is created and cannot be changed or accessed by the program.[9] When trying to retrieve a key `k` not stored in an object `x` itself, JavaScript *transitively* searches the *prototype chain* of `x` until it either finds `k` or it reaches the root of the object hierarchy without finding `k`. The prototype chain does not play a role in the semantics of key update, addition, or deletion.[10]

For example, consider the initially empty object `child` created by the function `beget` (described in the sequel) with prototype object `parent`. The prototype of object literals, like `parent`, is the object stored in `Object.prototype` (note that the "`prototype`" key of `Object` is *not* the same as its prototype object). Thus, all keys in `parent` and `Object.prototype` are transitively accessible via `child`.

```
var parent = {"last": " Doe"};
var child = beget(parent);
child.first = "John";
assert (child.first + child.last == "John Doe");
assert ("last" in child == true);
assert (child.hasOwnProperty("last") == false);
```

The JavaScript operator `k in x` tests for the presence of `k` *anywhere* along the prototype chain of `x`, whereas the native function `Object.prototype.hasOwnProperty` tests only the "own" object itself. Keys routinely resolve through prototypes, so a static type system must precisely track them.

[8] This variable corresponds to the "frame" from separation logic [16].

[9] Many implementations, however, do expose the prototype of an object `x` with a non-standard `x.__proto__` property. We discuss this further in § 7.

[10] This is not true in ES5 with the addition of setters, as discussed in § 7.

Unfortunately, we cannot encode prototypes directly within the framework of refinement types and strong update, as the semantics of transitively traversing mutable and unbounded prototype hierarchies is beyond the reach of decidable, first-order reasoning.

Shallow and Deep Heaps. We solve this problem by syntactically reducing reasoning about prototype-based objects to the refinement logic. Our key insight is to decompose the heap into a "shallow" part, the bounded portion of the heap for which we have explicit locations, and a "deep" part, which is the potentially unbounded portion which we can represent by uninterpreted heap variables H. We explicitly track prototype links in the "shallow" heap by using bindings of the form $(\ell \mapsto \langle d, \ell' \rangle)$, where the prototype of the object at ℓ is stored at ℓ'. We cannot track prototype links explicitly in the "deep" heap, so instead we *summarize* information about deep prototype chains by using the abstract (uninterpreted) heap predicate $\mathit{HeapHas}(H, \ell, k)$ to encode the proposition that the object stored at location ℓ in H *transitively* has the key k, and the abstract (uninterpreted) heap function $\mathit{HeapSel}(H, \ell, k)$ to represent the corresponding value retrieved by lookup.

As an example, recall the `child` object and its prototype `parent`. Suppose that the prototype of `parent` is an unknown object `grandpa`, rather than `Object.prototype` as written. If `child`, `parent`, and `grandpa` are stored at locations ℓ_1, ℓ_2, and ℓ_3 with underlying "own" dictionary values d_1, d_2, and d_3, then we write the heap $\{\ell_1 \mapsto \langle d_1, \ell_2 \rangle, \ell_2 \mapsto \langle d_2, \ell_3 \rangle, \ell_3 \mapsto \langle d_3, \ell_4 \rangle, H\}$ — we use set notation to abbreviate the concatenation of heap bindings with $\oplus$. Despite not knowing what value is the prototype of `grandpa`, we name its location ℓ_4 that is somewhere in the deep part of the heap H.

Key Membership and Lookup. When describing simple objects, we used the original System D primitives (`mem` and `get`) to desugar key membership and lookup operations. But in fact, to account for the transitive semantics of key membership and lookup facilitated by prototype links, System !D uses the new primitives `hasPropObj` and `getPropObj` defined in `objects.dref` (Figure 3). These primitives differ from their purely functional System D counterparts in two ways: each operation goes through a *reference* to a dictionary on the heap, and the abstract predicates ObjHas and ObjSel are used in place of has and sel. These abstract predicates are defined over the *disjoint union* of the shallow and deep heaps and, intuitively, summarize whether an object transitively has a key and, if so, the value it binds.

Transitive Semantics via Unrolling. We encode the transitive lookup semantics in the abstract predicates by decomposing the predicates into shallow and deep disjuncts. In particular, we define the predicates by *unrolling* the shallow part of the heap by chasing prototype links until we "bottom out" with uninterpreted predicates for the deep heap.

Let us return to the example of the `child`, `parent` and `grandpa` prototype chain to understand how unrolling captures the transitive lookup. The DJS key membership test on the left desugars to System !D as follows.

`k in child`	`hasPropObj (deref _child) (deref _k)`

The result of the function call has the following type.

$$\{\nu \textbf{ iff } \mathit{ObjHas}(d_1, \mathtt{k}, \{(\ell_2 \mapsto \langle d_2, \ell_3 \rangle), (\ell_3 \mapsto \langle d_3, \ell_4 \rangle), H\}, \ell_2)\}$$

We expand this type by unrolling ObjHas to the following.

$$\{\nu \textbf{ iff } \mathit{has}(d_1, \mathtt{k}) \vee \mathit{has}(d_2, \mathtt{k}) \vee \mathit{has}(d_3, \mathtt{k}) \vee \mathit{HeapHas}(H, \ell_4, \mathtt{k})\}$$

The first three disjuncts correspond to looking for the key in the shallow heap, and the last is the uninterpreted predicate that summarizes whether the key exists in the deep heap. Similarly, key lookup in DJS is desugared as follows.

`child[k]`	`getPropObj (deref _child) (deref _k)`

We unroll the type of the System !D expression as follows.

$$\begin{array}{l} \{\textbf{if } \mathit{has}(d_1, \mathtt{k}) \textbf{ then } \nu = \mathit{sel}(d_1, \mathtt{k}) \textbf{ else} \\ \;\textbf{if } \mathit{has}(d_2, \mathtt{k}) \textbf{ then } \nu = \mathit{sel}(d_2, \mathtt{k}) \textbf{ else} \\ \;\textbf{if } \mathit{has}(d_3, \mathtt{k}) \textbf{ then } \nu = \mathit{sel}(d_3, \mathtt{k}) \textbf{ else} \\ \;\textbf{ite } \mathit{HeapHas}(H, \ell_4, \mathtt{k})\ (\nu = \mathit{HeapSel}(H, \ell_4, \mathtt{k}))\ \mathit{Undef}(\nu)\} \end{array}$$

Thus, our technique of decomposing the heap into shallow and deep parts, followed by heap unrolling, captures the *exact* semantics of prototype-based object operations modulo the unknown portion of the heap. Thus, System !D precisely tracks objects in the presence of mutation and prototypes.

Additional Primitives. The new update and deletion primitives `setPropObj` and `delPropObj` (Figure 3) affect only the "own" object, since the prototype chain does not participate in the semantics. We model native JavaScript functions like `Object.prototype.hasOwnProperty` with type annotations in the file `prelude.js` (Figure 4). Notice that the function type for objects (the first in the intersection) checks only the "own" object for the given key.

Constructors. JavaScript provides the expression form `new Foo(args)` as a second way of *constructing* objects, in addition to object literals whose prototypes are set to `Object.prototype`. The semantics are straightforward, but quite different than the traditional `new` syntax suggests. Here, if `Foo` is any function (object), then a fresh, empty object is created with prototype object `Foo.prototype`, and `Foo` is called with the new object bound to `this` (along with the remaining arguments) to finish its initialization. We desugar constructors and `new` with standard objects and functions (following λ_{JS} [21]) *without* adding any special System !D constructs or primitive functions.

```
var __hasOwn =
  /*: (this:Ref, k:Str)/(this ↦ ⟨d:Dict, thiṡ⟩) → {ν iff has(d,k)}/same
    ∧ ∀A. (this:Ref, i:Int)/(this ↦ ⟨a:Arr(A), thiṡ⟩) → {ite packed(a) (ν iff 0 ≤ i < len(a)) Bool(ν)}/same
    ∧ ∀A. (this:Ref, k:Str)/(this ↦ ⟨a:Arr(A), thiṡ⟩) → {ν iff k = "length"}/same */ "#extern";

function Object() { ... }; Object.prototype = {"hasOwnProperty": __hasOwn, "constructor": Object, ... };

var __push = /*: ∀A. (this:Ref, x:A)/(this ↦ ⟨a:Arr(A), thiṡ⟩)
                  → Int/(this ↦ ⟨a′:{ν :: Arr(A) ∧ arrSize(ν,a,1)}, thiṡ⟩) */ "#extern";
var  __pop = /*: ∀A. (this:Ref, x:A)/(this ↦ ⟨a:Arr(A), thiṡ⟩)
                  → {ite packed(a) (ν :: A) (ν :: A ∨ Undef(ν))}
                  / (this ↦ ⟨a′:{ν :: Arr(A) ∧ arrSize(ν,a,−1)}, thiṡ⟩) */ "#extern";

function Array() { ... }; Array.prototype = {"push": __push, "pop": __pop, "constructor": Array, ... };
```

Figure 4. Excerpt from `prelude.js`

Inheritance. Several inheritance relationships, including ones that simulate traditional classes, can be encoded with the construction mechanism, as shown in the popular book *JavaScript: The Good Parts* [10]. Here, we examine the *prototypal pattern*, a minimal abstraction which wraps construction to avoid the unusual syntax and semantics that leads to common errors; we discuss the rest in § 6. The function `beget` (the basis for `Object.create` in ES5) returns a fresh empty object with prototype `o`.

```
1  /*: ∀L. o:Ref/(o ↦ ⟨d:Dict, ȯ⟩)
2       → Ref L/(L ↦ ⟨{ν = empty}, o⟩) ⊕ (o ↦ same) */
3  function beget(o) {
4    /*: #ctor this:Ref → {ν = this} */
5    function F() { return this; };
6    F.prototype = o;
7    return new /*:L*/ F();                          }
```

The `#ctor` on line 4 instructs desugaring to: initialize the function object with a "`prototype`" key that stores an empty object literal (since it will be called as a constructor); and expand the type annotation as follows to require that `this` initially be empty, as is common for all constructors.

$$this{:}Ref\,/\,(this \mapsto \langle\{\nu = empty\}, \dot{this}\rangle) \to \{\nu = this\}\,/\,same$$

The assignment on line 6 strongly updates `Foo.prototype` (overwriting its initial empty object) with the argument `o`. Thus, the object constructed (at location L) on line 7 has prototype `o`, so `beget` has the ascribed type. In most cases, `new` can be used without a location annotation and a fresh one is chosen. In this case, we annotate line 7 with L (from the type of `beget`), which our implementation does not infer because there is no input corresponding to L.

2.6 Arrays

The other workhorse data structure of JavaScript are arrays, which are really just objects with integer "indices" converted to ordinary string keys. However, arrays pose several tricky challenges as they are commonly used both as finite tuples as well as unbounded collections.

```
var arr = [17, "hi", true];
arr[3] = 3; arr.push(4);
assert (arr.length == 5 && arr[5] == undefined);
```

As for any object, retrieving a non-existent key returns `undefined` rather than raising an "out-of-bounds" exception. Like other objects, arrays are extensible simply by writing "past the end." Array literals objects have prototype `Array.prototype`, which includes a `push` (resp. `pop`) function for adding an element to (resp. removing an element from) the end of an array.

Loops are used to iterate over arrays of unknown size. But since lookups may return `undefined`, it is important to track when an access is "in-bounds." JavaScript bestows upon arrays an unusual "`length`" property, rather than a method, to help. Reading it returns the largest integer key of the array, which is *not* necessarily its "size" because it may contain "holes" or even non-integer keys. Furthermore, assigning a number `n` to the "`length`" of an array either *truncates* it if `n` is less than its current length, or *extends* it (by padding with holes) if it is greater. Despite the unusual semantics, programmers commonly use arrays as if they are traditional "packed" arrays with integer "indices" zero to "size" minus one. The type system must reconcile this discrepancy.

Array Types. We introduce a new syntactic type term $Arr(T)$ and maintain the following four properties for every value a that satisfies the has-type predicate $a :: Arr(T)$. We refer to strings that do *not* coerce to integers as "safe," and we use an uninterpreted predicate *safe* to describe such strings (*e.g.* $safe(\text{"foo"})$ whereas $\neg safe(\text{"17"})$).

(A1) a contains the special "`length`" key.

(A2) All other "own" keys of a are (strings that coerce to) integers.

(A3) For all integers i, either a maps the key i to a value of type T, or it has no binding for i.

(A4) All inherited keys of a are safe (*i.e.* non-integer) strings.

An array can have arbitrary objects in its prototype chain, so to ensure (A4), we require that *all* non-array objects bind only safe strings. This sharp distinction between between array objects (that bind integer keys) and non-array objects (that bind safe string keys) allows System !D to avoid reasoning about string coercions, and does not significantly limit expressiveness because programs typically conform to this division anyway.[11] To enforce this restriction, the type for keys manipulated by primitives in `objects.dref` and `prelude.js` is actually $\mathit{SafeStr}$, rather than Str as shown in Figure 3 and Figure 4, where $\mathit{SafeStr} \mathrel{\hat{=}} \{\mathit{Str}(\nu) \wedge \mathit{safe}(\nu)\}$.

Packed Arrays. Arrays a that additionally satisfy the uninterpreted predicate $\mathit{packed}(a)$ enjoy the following property, where $\mathit{len}(a)$ is an uninterpreted function symbol.

(A5) For all integers i, if i is between zero and $\mathit{len}(a)$ minus one, then a maps i to a value of type T. Otherwise, a has no binding for i.

Tuple Arrays. Using additional predicates, System !D gives precise types to array literals, which are often used as finite tuples in idiomatic code. For example, we can describe a pair of integers with the type $\{\nu :: \mathit{Arr}(\mathit{Int}) \wedge \mathit{packed}(\nu) \wedge \mathit{len}(\nu) = 2\}$ and a *heterogeneous* pair with $\{\nu :: \mathit{Arr}(\mathit{Top}) \wedge \mathit{packed}(\nu) \wedge \mathit{len}(\nu) = 2 \wedge \mathit{Str}(\mathit{sel}(\nu, \text{“0”})) \wedge \mathit{Bool}(\mathit{sel}(\nu, \text{“1”}))\}$. Thus, the technique of nested refinements allows us to smoothly reason about arrays both as packed homogenous collections and as heterogenous tuples.

Array Primitives. We define several array-manipulating primitives in `objects.dref` (some of which we show in Figure 3) that maintain and use the array invariants above. For key lookup on arrays, we define three primitives: `getIdxArr` looks for the integer key i on the own object a and ignores the prototype chain of a because (A4) guarantees that a will not inherit i, and returns a value subject to the properties (A3) and (A5) that govern its integer key bindings; `getLenArr` handles the special case when the string key k is “`length`”, which (A1) guarantees is bound by a, and returns the the true length of the array only if it is packed; and `getPropArr` deals with all other (safe) string keys k by reading from the prototype chain of the array (re-using the heap unrolling mechanism) ignoring its own bindings because of (A2).

For array updates, we define `setIdxArr` that uses the following macros to preserve packedness (A5) when possible.

$$\begin{aligned}\mathit{arrSet}(a', a, i) \mathrel{\hat{=}}\ & \textbf{if}\ 0 \le i < \mathit{len}(a)\ \textbf{then}\ \mathit{arrSize}(a', a, 0)\ \textbf{else}\\ & \textbf{if}\ i = \mathit{len}(a)\ \textbf{then}\ \mathit{arrSize}(a', a, 1)\ \textbf{else}\ \mathit{true}\end{aligned}$$

$$\begin{aligned}\mathit{arrSize}(a', a, n) \mathrel{\hat{=}}\ & \mathit{packed}(a) \Rightarrow\\ & (\mathit{packed}(a') \wedge \mathit{len}(a') = \mathit{len}(a) + n)\end{aligned}$$

[11] We discuss an alternative approach in § 7.

In particular, the updated array a' is packed if: (1) the original array a is packed; and (2) the updated index i is either within the bounds of a (in which case, the length of a' is the same as a) or just past the end (so the length of a' is one greater than a). In similar fashion, we specify the remaining primitives for update and deletion to maintain the array invariants, and the ones for key membership to use them, but we do not show them in Figure 3.

In `prelude.js` (Figure 4), we use precise types to model the native `push` and `pop` methods of `Array.prototype` (which maintain packedness, as above), as well as the behavior of `Object.prototype.hasOwnProperty` on arrays (the last two cases of the intersection type). Thus, the precise dependent types we ascribe to array-manipulating operations maintain invariants (A1) through (A5) and allow DJS to precisely track array operations.

Desugaring. It may seem that we need to use separate primitive functions for array and non-array object operations, even though they are syntactically indistinguishable in JavaScript. Nevertheless, we are able to desugar DJS based purely on expression syntax (and *not* type information) by unifying key lookup within a single primitive `getElem` and giving it a type that is the *intersection* of the (three) array lookup primitives and the (one) non-array lookup primitive `getPropObj`. We define `getElem` in Figure 3, where we specify the intersection type using `and` and `type` as syntactic sugar to refer to the previous type annotations. We define similar unified primitives for `setElem`, `hasElem`, and `delElem` (not shown in Figure 3). Desugaring uniformly translates object operations to these unified general primitives, and type checking of function calls ensures that the appropriate cases of the intersection type apply.

2.7 Collections

As discussed in § 2.2, strong updates are sound only for references that point to *exactly one* object, which is far too restrictive as real programs manipulate collections of objects. In this section, we describe *weak references* in DJS to refer to multiple objects, a facility that enables programming with arrays of mutable objects as well as recursive types.

Weak References. In the following example, we iterate over an array of passenger objects and compute the sum of their weights; we use a default value `max_weight` when a passenger does not list his weight (ignore the unfamiliar annotations for now).

```
/*: (ℓ̃pass ↦ frzn) → same */
for (i=0; i < passengers.length; i++) {
  var p = passengers[i];
  /*: #thaw p */
  if (p.weight) { sum += p.weight;   }
  else          { sum += max_weight; }
  /*: #freeze p */
}
```

We could describe the array `passengers` with the type $Ref\,\ell$ for a location ℓ. However, this type is not very useful as it denotes an array of references to a *single* object.

Weak Locations. To refer to an *arbitrary number* (zero or more) objects of the same type, we adopt the Alias Types [32] solution, which categorizes some locations as *weak* to describe an arbitrary number of locations that satisfy the same type, and syntactically ensures that weak locations are weakly updated.

We introduce a new kind of heap binding $(\tilde{\ell} \mapsto \langle T, \ell' \rangle)$, where $\tilde{\ell}$ is a weak location, all objects that might reside there satisfy T, and ℓ' is the strong location of the prototype of all objects that reside at location $\tilde{\ell}$. There is *no* heap binder for weak locations since there is not a single value to describe.

In our example, we can use $(\tilde{\ell}_{pass} \mapsto \langle T_{pass}, \ell_{op} \rangle)$ to describe passenger objects, where ℓ_{op} is the location of `Object.prototype` and T_{pass} is the dictionary type $\{Dict(\nu) \wedge has(\nu, \text{“}\texttt{weight}\text{”}) \Rightarrow Num(sel(\nu, \text{“}\texttt{weight}\text{”}))\}$. If we assign the type $\{\nu :: Arr(Ref\,\tilde{\ell}_{pass}) \wedge packed(\nu)\}$, to `passengers`, then `p` has type $Ref\,\tilde{\ell}_{pass}$, and thus each (desugared) use of `p` is a dictionary of type T_{pass}. This type is quite unsatisfying, however, because the conditional establishes that along the then-branch, `p` *does* possess the key and therefore should be assigned the more precise type $\{Num(sel(d, \text{“}\texttt{weight}\text{”}))\}$.

Thaw / Freeze. To solve this problem, we adopt a mechanism found in derivatives of Alias Types (*e.g.* [2, 12, 14, 31]) that allows a weak location to be *temporarily* treated as strong. A weak location $\tilde{\ell}$ is said to be *frozen* if all references $Ref\,\tilde{\ell}$ use the location only at its weak (invariant) type. The type system can *thaw* a location, producing a strong reference $Ref\,\ell_k$ (with a fresh name) that can be used to strongly update the type of the cell. While a location is thawed, the type system prohibits the use of weak references to the location, and does not allow further thaw operations. When the thawed (strong) reference is no longer needed, the type system checks that the original type has been restored, *re-freezes* the location, and discards the thawed location. Soundness of the approach depends on the invariant that each weak location has at most one corresponding thawed location at a time.

In our example, we do not need to temporarily violate the type of `p`, but the thaw/freeze mechanism does help us *relate* the two accesses to `p`. The *thaw state* annotation above the loop declares that before each iteration of the loop (including the first one), the location $\tilde{\ell}_{pass}$ must be frozen. The thaw annotation on line 4 changes the type of `p` to a strong reference to a fresh thawed location ℓ_1, which stores a *particular* dictionary on the heap (named with a binder) that is retrieved by both subsequent uses of `p`. Thus, we can relate the key membership test to the lookup, and track that `p.weight` produces a number. The freeze annotation on line 7 restores the invariant required before the next iteration. We describe this technique formally in § 4.

Recursive Types. We reuse the weak location mechanism to describe recursive data structures. Consider the following adapted from the SunSpider [34] benchmark `access-binary-trees.js`, annotated in DJS.

```
1  /*: #define T_tn {"i":Num, "l", "r":Ref ℓ̃_tn} */
2  /*: #weak (ℓ̃_tn ↦ ⟨T_tn, ℓ_tnp⟩) */
3  /*: #ctor (this:Ref, left,right:Ref ℓ̃_tn, item:Num)
4             /(ℓ̃_tn ↦ frzn) → Ref ℓ̃_tn / same */
5  function TreeNode(left, right, item) {
6    this.l = left; this.r = right; this.i = item;
7    /*: #freeze this */
8    return this;
9  }
10 /*: this:Ref ℓ̃_tn → Num */
11 TreeNode.prototype.itemCheck = function f() {
12   // thaw/freeze annotations inferred
13   if (this.l == null) return this.item;
14   else { return this.i
15              + f.apply(this.l)
16              - f.apply(this.r); }
17 }
```

The source-level macro on line 1 introduces T_{tn} to abbreviate the type of `TreeNode`s, using traditional record type syntax instead of the underlying McCarthy operators. Line 2 defines the weak location for `TreeNode`s, using the predictable location ℓ_{tnp} created by desugaring for the object `TreeNode.prototype`. The constructor annotation itself declares that the return type is a reference to one of these recursive objects, which System !D verifies by checking that on line 6 the appropriate fields are added to the strong, initially-empty object `this` before it is frozen and returned.

Recursive Traversal. There are two differences compared to the original version of `itemCheck`, which cannot be type checked in DJS. First, we name the function being defined (notice the `f` on line 11), a JavaScript facility for recursive definitions. Second, we write `f.apply(this.r)` instead of `this.r.itemCheck()` as in the original, where the native JavaScript function `apply` allows a caller to explicitly supply a receiver argument. The trouble with the original call is that it goes through the heap (in particular, the prototype chain of `this`) to resolve the recursive function being defined. This function will be stored in a strong object, and we have no facility (*e.g.* mu-types) for strong objects with recursive types; our only mechanism is for weak objects. If we write `f.apply(this.r)`, however, the recursive function `f` is syntactically manifest, and we can translate the definition with a call to the standard `fix` primitive (Figure 1). In § 5, we describe how we handle a limited form of `apply` that is sufficient for our idiomatic recursive definitions in DJS. We expect that we can add a more powerful mechanism for recursive types that supports the original code as written, but we leave this to future work.

2.8 Rest of the Paper

We have now completed our tour of Dependent JavaScript. Next, we formally define the syntax of System !D in § 3 and the type system in § 4. In, § 5, we present the syntax of DJS and its desugaring to System !D. We discuss our implementation and results in § 6, directions for future work in § 7, and related work in § 8.

3. Syntax and Semantics of System !D

We now introduce the formal syntax of values, expressions, and types of System !D, defined in Figure 5.

Values. Values v include variables x, constants c, lambdas $\lambda x.\ e$, (functional) dictionaries v_1 `++` $v_2 \mapsto v_3$, and run-time heap locations r. The set of constants c includes base values (numbers, booleans, strings, the empty dictionary $\{\}$, `null`, `undefined`, `NaN`, *etc.*) and the primitive functions from `basics.dref` and `objects.dref` (`typeof`, `get`, `getElem`, *etc.*). Logical values w are all values and applications of primitive function symbols F, such as addition + and dictionary selection sel, to logical values.

Expressions. Expressions e include values, function application, if-expressions, and let-bindings. The ascription form e `as` T allows source-level type annotations. Since function types will be parameterized by type, location, and heap variables, the syntax of function application requires that these be instantiated. Reference operations include reference allocation, dereference, and update, and the run-time semantics maintains a separate heap that maps locations to values. The expression `newobj` $\ell\ v\ v'$ stores the value v at a fresh location r — where the name ℓ is a compile-time abstraction of a set of run-time location names that includes r – with its prototype link set to v', which should be a location. The `thaw` $\ell\ v$ operation converts a weak reference to a strong strong one; `freeze` $\tilde{\ell}\ \theta\ v$ converts a strong reference to a weak one, where thaw state θ is used by the type system for bookkeeping. An expression $@x : e$ labels the enclosed expression, and a break expression `break` $@x\ v$ terminates execution of the innermost expression labeled $@x$ within the function currently being evaluated and produces the result v. If no such labeled expression is found, evaluation becomes stuck. Label and break expressions are included to translate the control flow operations of DJS. Exceptions are arbitrary objects that can be raised and handled as usual.

The operational semantics is standard, based on λ_{JS} with minor differences. For example, we make prototype links manifest in the syntax of heaps (to facilitate heap unrolling in the type system), whereas λ_{JS} stores them inside objects in a distinguished "`__proto__`" field. We refer the reader to [21] for the full details.

We use an A-normal expression syntax so that we need only define substitution of values (not arbitrary expressions) into types. We use a more general syntax for examples throughout this paper, and our implementation desugars expressions into A-normal form. We use tuple syntax $(v_0,\ \ldots,\ v_n)$ as sugar for the dictionary with fields "`0`" through "`n`" bound to the component values.

Types and Formulas. Values in System !D are described by *refinement types* of the form $\{x\,|\,p\}$ where x may appear free in the formula p,[12] and *existential types* $\exists x{:}T.\ S$ where x may appear free in S. Existentials are created during type checking, and, by convention (rather than introducing more syntactic categories), do not appear in programs. The language of *refinement formulas* includes predicates P, such as the equality and dictionary predicate has, and the usual logical connectives. Similar to the syntax for expression tuples, we use $(T_1,\ \ldots,\ T_n)$ as sugar for the dictionary type with fields "`0`" through "`n`" with the corresponding types.

Locations. To track which locations can be used to instantiate variables, we distinguish between strong and weak location constants a and $\tilde{a}$, and strong and weak location variables L and $\tilde{L}$. Strong locations ℓ and weak locations $\tilde{\ell}$ can be constants or variables. We use m to range over arbitrary location constants and M for arbitrary location variables.

Type Terms and Type Predicates. As in System D, we use an uninterpreted *has-type predicate* $w :: U$ in formulas to describe values that have complex types, represented by *type terms* U, which includes function types, type variables, null, reference, and array types. A reference type names a strong or weak location in the heap. In § 2.4, we discussed how function types are parameterized by sequences of type, location, and heap variables, and describe input and output *worlds* (type-heap pairs). At that point in the presentation, we had only seen strong locations, so arrows were parameterized with strong location variables L. Since System !D includes weak locations as well, function types are parameterized by arbitrary location variables M and also include an explicit *weak heap* Ψ. We have not shown any examples that abstract over weak locations in this paper.

Heap Types. A *heap type* $\hat{\Sigma}$ is an unordered set of *heap variables* H and *heap bindings* $\hat{h}$ concatenated with the $\oplus$ operator. We syntactically require that each heap has exactly one heap variable, so we write a heap type as the pair $(H, \hat{h})$, where H is the "deep" part for which we have no information and $\hat{h}$ is the "shallow" part for which have precise location information. The heap binding $(\ell \mapsto x{:}T)$ represents the fact that the value at location ℓ has type T; the binder x refers to this value in the types of other heap bindings. The binding $(\ell \mapsto \langle x{:}T, \ell'\rangle)$ additionally records a prototype link ℓ'. The binding $(\tilde{\ell} \mapsto \theta)$ records the current *thaw state* of weak location $\tilde{\ell}$, to help maintain the invariant that it has at most one thawed location at a time.

[12] The presentation in [8] required that the binder be ν; here, we allow the binder to be arbitrary, but use ν when it is convenient. Also, unlike in the previous presentation, we eliminate the stratification between polymorphic and monomorphic types; in System !D, polymorphism is expressed with the type parameters of function types.

$$
\begin{array}{rccl}
\textbf{Values} & v & ::= & x \mid c \mid v_1 \texttt{++} v_2 \mapsto v_3 \mid \lambda x.\, e \mid r \\
\textbf{Expressions} & e & ::= & v \mid [\overline{T};\overline{m};\overline{\Sigma}]\, v_1\, v_2 \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid e\ \texttt{as}\ T \\
& & \mid & \texttt{ref}\ \ell\ v \mid \texttt{deref}\ v \mid v_1 := v_2 \mid \texttt{newobj}\ \ell\ v\ v' \mid \texttt{freeze}\ \tilde{\ell}\ \theta\ v \mid \texttt{thaw}\ \ell\ v \\
& & \mid & @x : e \mid \texttt{break}\ @x\ e \mid \texttt{throw}\ e \mid \texttt{try}\ e_1\ \texttt{catch}\ (x)\ e_2 \mid \texttt{try}\ e_1\ \texttt{finally}\ e_2 \\
\textbf{Types} & S, T & ::= & \{x \,|\, p\} \mid \exists x{:}T.\, S \\
\textbf{Formulas} & p, q & ::= & P(\overline{w}) \mid w :: U \mid \mathit{HeapHas}(H, \ell, w) \mid p \wedge q \mid p \vee q \mid \neg p \\
\textbf{Logical Values} & w & ::= & v \mid F(\overline{w}) \mid \mathit{HeapSel}(H, \ell, w) \\
\textbf{Syntactic Type Terms} & U & ::= & \forall[\overline{A};\overline{M};\overline{H}]\, \Psi / x{:}T_1 / \hat{\Sigma}_1 \to W \mid A \mid \mathit{Null} \mid \mathit{Ref}\,\ell \mid \mathit{Ref}\,\tilde{\ell} \mid \mathit{Arr}(T) \\
\textbf{Heap Bindings} & \hat{h} & ::= & (\ell \mapsto x{:}T) \mid (\ell \mapsto \langle x{:}T, \ell' \rangle) \mid (\tilde{\ell} \mapsto \theta) \mid \hat{h}_1 \oplus \hat{h}_2 \mid \varnothing \\
\textbf{Weak Heap Bindings} & \Psi & ::= & (\tilde{\ell} \mapsto \langle T, \ell \rangle) \mid \Psi_1 \oplus \Psi_2 \mid \varnothing
\end{array}
$$

$$
\begin{array}{rcclrclrcl}
\textbf{Worlds} & W & ::= & T_2 / \hat{\Sigma}_2 & x, y, z & \in & \mathit{Identifiers} & A, B & \in & \mathit{TypeVariables} \\
\textbf{Heaps} & \hat{\Sigma} & ::= & (H, \hat{h}) & a & \in & \mathit{LocationConstants} & H & \in & \mathit{HeapVariables} \\
\textbf{Strong Locations} & \ell & ::= & a \mid L & r & \in & \mathit{DynamicHeapLocations} & L & \in & \mathit{LocationVariables} \\
\textbf{Weak Locations} & \tilde{\ell} & ::= & \tilde{a} \mid \tilde{L} & F & \in & \mathit{LogicalFunctionSymbols} & P & \in & \mathit{LogicalPredicateSymbols} \\
\textbf{Thaw States} & \theta & ::= & \texttt{frzn} \mid \texttt{thwd}\ \ell & m & ::= & \ell \mid \tilde{\ell} & M & ::= & L \mid \tilde{L}
\end{array}
$$

$$
c \in \mathit{ValueConstants} \supset \{\, \texttt{true}, \texttt{false}, \texttt{null}, \texttt{undefined}, 1, 2, \text{“}\texttt{hanna}\text{”}, (\texttt{==}), \texttt{!}, \texttt{typeof}, \texttt{get}, \texttt{getElem}, \texttt{fix} \,\}
$$

Figure 5. Syntax of System !D

Uninterpreted Heap Symbols. To describe invariants about the deep part of a heap, System !D introduces two uninterpreted *heap symbols*. The predicate $\mathit{HeapHas}(H, \ell, k)$ represents the fact that the chain of objects in H starting with ℓ has the key k. Similarly, the function symbol $\mathit{HeapSel}(H, \ell, k)$ refers to the value retrieved when looking up key k in the heap H starting with ℓ.

4. Type Checking

In this section, we present the well-formedness, typing, and subtyping relations of System !D. The type system reuses the System D [8] subtyping algorithm to factor subtyping obligations between a first-order SMT solver and syntactic subtyping rules. The novel technical developments are: the formulation of flow-sensitive heap types in a dependent setting; the use of uninterpreted heap symbols to regain precision in the presence of imperative, prototype-based objects; the encoding of array primitives to support idiomatic use of JavaScript arrays; and the use of refinement types to assign precise types to JavaScript operators.

Environments. Type checking uses several environments.

$$
\begin{array}{rcl}
\Gamma & ::= & \varnothing \mid \Gamma, x{:}T \mid \Gamma, p \mid \Gamma, A \mid \Gamma, M \mid \Gamma, H \\
& \mid & \Gamma, (\tilde{\ell} \mapsto \langle T, \ell \rangle) \\
\Sigma & ::= & (H, h) \\
h & ::= & \varnothing \mid h_1 \oplus h_2 \mid (\ell \mapsto v) \mid (\ell \mapsto \langle v, \ell' \rangle) \mid (\tilde{\ell} \mapsto \theta) \\
\Omega & ::= & \varnothing \mid \Omega, @x{:}(T/\Sigma)
\end{array}
$$

A type environment Γ binding records either: the derived type for a variable; a formula p to track control flow along a conditional branch; a polymorphic variable introduced by a function type; or the description of a weak location (which does not change flow-sensitively), namely, that every object stored at $\tilde{\ell}$ satisfies type T and has prototype link ℓ. A heap environment Σ is just like a heap type, except a strong location ℓ binds the value v it stores (as opposed to the *type* of v). A label environment Ω binding records the world that the expression labeled $@x$ is expected to satisfy.

4.1 Well-Formedness

We highlight a few aspects of the largely straightforward well-formedness relations, defined in in Appendix A.

Types, Formulas, and Logical Values. We require that types be *well-formed* within the current type environment, where formulas are boolean propositions and mention only variables that are currently in scope.

Heap Types. Locations in a heap type $\hat{\Sigma}$ must either be location constants or location variables bound by the type environment, and may not be bound multiple times. All heap binders may refer to each other. Thus, the values in a heap can be regarded as a dependent tuple.

Function Types. Checking that a function type is well-formed proceeds in several steps. First, the polymorphic variables are added to the environment. Second, the weak heap Ψ is checked. Next, the locations of Ψ are added to the environment, and input type T_1 is checked. Then, the input heap type $\hat{\Sigma}_1$ is checked in an environment extended with the input binder x. Finally, the procedure Binders collects all binders from the input heap to be added to the environment,

Subtyping $\boxed{\Gamma \vdash T_1 \sqsubseteq T_2}$

$$\text{[S-Refinement]}\ \frac{y \text{ fresh} \quad \Gamma, p[y/x] \Rightarrow q[y/x]}{\Gamma \vdash \{x \,|\, p\} \sqsubseteq \{x \,|\, q\}} \qquad \text{[S-Exists]}\ \frac{\Gamma, x{:}T \vdash S_1 \sqsubseteq S_2}{\Gamma \vdash \exists x{:}T.\ S_1 \sqsubseteq S_2}$$

Implication $\boxed{\Gamma \Rightarrow p}$

$$\frac{\mathsf{CNF}(p) = \wedge_i\ (p_i \Rightarrow Q_i) \qquad \forall i.\ \exists q \in Q_i.\ \Gamma, p_i \Rightarrow q}{\Gamma \Rightarrow p}\ \text{[I-Cnf]}$$

$$\text{[I-Valid]}\ \frac{\mathsf{Valid}([\![\Gamma]\!] \Rightarrow p)}{\Gamma \Rightarrow p} \qquad \text{[I-ImpSyn]}\ \frac{\mathsf{Valid}([\![\Gamma]\!] \Rightarrow w :: U') \qquad \Gamma \vdash U' <: U}{\Gamma \Rightarrow w :: U}$$

Syntactic Subtyping $\boxed{\Gamma \vdash U_1 <: U_2}$

$$\frac{\begin{array}{c} U_1 = \forall[\overline{A}; \overline{M}; \overline{H}]\ \Psi / x{:}T_1 / (H, \hat{h}_1) \to W_1 \\ U_2 = \forall[\overline{A}; \overline{M}; \overline{H}]\ \Psi / x{:}T_2 / (H, \hat{h}_2) \to W_2 \\ \Gamma \vdash T_2 \sqsubseteq T_1 \quad \Gamma, x{:}T_2 \vdash \hat{h}_2 \sqsubseteq \hat{h}_1;\ \pi \quad \Gamma, x{:}T_2, [\![\hat{h}_2]\!] \vdash \pi W_1 \sqsubseteq W_2 \end{array}}{\Gamma \vdash U_1 <: U_2}$$

$$\frac{}{\Gamma \vdash \mathit{Arr}(T) <: \mathit{Arr}(T)} \qquad \frac{}{\Gamma \vdash A <: A} \qquad \frac{}{\Gamma \vdash \mathit{Ref}\,\ell <: \mathit{Ref}\,\ell}$$

$$\frac{}{\Gamma \vdash \mathit{Null} <: \mathit{Ref}\,\tilde{\ell}} \qquad \frac{}{\Gamma \vdash \mathit{Ref}\,\tilde{\ell} <: \mathit{Ref}\,\tilde{\ell}}$$

Figure 6. Subtyping for System !D

Value Typing $\boxed{\Gamma;\ \Sigma;\ \Omega \vdash v :: T}$

$$\text{[T-Const]}\ \frac{}{\Gamma;\ \Sigma;\ \Omega \vdash c :: ty(c)} \qquad \text{[T-Var]}\ \frac{\Gamma(x) = S}{\Gamma;\ \Sigma;\ \Omega \vdash x :: \{y \,|\, y = x\}}$$

$$\frac{\mathsf{StaticLoc}(r) = m \qquad m \in dom(\Sigma)}{\Gamma;\ \Sigma;\ \Omega \vdash r :: \{x \,|\, x = r \wedge x :: \mathit{Ref}\, m\}}\ \text{[T-Loc]}$$

$$\text{[T-Extend]}\ \frac{\Gamma;\ \Sigma;\ \Omega \vdash (v_1,\ v_2,\ v_3) :: (\mathit{Dict},\ \mathit{Str},\ T)}{\Gamma;\ \Sigma;\ \Omega \vdash v_1 \mathbin{++} v_2 \mapsto v_3 :: \{x \,|\, x = v_1 \mathbin{++} v_2 \mapsto v_3\}}$$

$$\text{[T-Fun]}\ \frac{\begin{array}{c} U = \forall[\overline{A}; \overline{M}; \overline{H}]\ \Psi / x{:}T_1 / \hat{\Sigma}_1 \to W_2 \qquad \Gamma \vdash U \qquad \Omega_1 = \varnothing \\ \mathsf{Snapshot}(\hat{\Sigma}_1) = (\overline{z}{:}\overline{S}, \Sigma_1) \qquad \Gamma_1 = \Gamma, \overline{A}, \overline{M}, \overline{H}, \Psi, x{:}T_1, \overline{z}{:}\overline{S} \\ \Gamma_1;\ \Sigma_1;\ \Omega_1 \vdash e :: T_2 / \Sigma_2 \qquad \Gamma_1 \vdash T_2 / \Sigma_2 \vDash W_2;\ \pi \end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash \lambda x.\ e :: \{y \,|\, y = \lambda x.\ e \wedge y :: U\}}$$

World Satisfaction $\boxed{\Gamma \vdash T / \Sigma \vDash W;\ \pi}$

$$\frac{\Gamma \vdash T \sqsubseteq S \qquad \Gamma \vdash \Sigma \vDash \hat{\Sigma};\ \pi \qquad \Gamma \Rightarrow \pi[\![\hat{\Sigma}]\!]}{\Gamma \vdash T / \Sigma \vDash S / \hat{\Sigma};\ \pi'}$$

Figure 7. Value type checking for System !D

so that the output world $T_2 / \hat{\Sigma}_2$ can refer to them and, thus, express precise relationships with the input heap.

4.2 Subtyping

Several relations, defined in Figure 6, comprise subtyping.

Subtyping, Implication, and Syntactic Subtyping. As in System D, subtyping on refinement types reduces to implication of refinement formulas, which are discharged by a combination of uninterpreted, first-order reasoning and syntactic subtyping. If the SMT solver alone cannot discharge an implication obligation (I-Valid), the formula is rearranged into conjunctive normal form (I-Cnf), and goals of the form $w :: U$ are discharged by a combination of uninterpreted reasoning and syntactic subtyping (I-ImpSyn). We write $[\![T]\!]$ for the *embedding* of a type as a formula, a straightforward definition [8] that lifts to environments $[\![\Gamma]\!]$ and heap bindings $[\![\hat{h}]\!]$. Our treatment of existential types follows the algorithmic (decidable) approach in [25]. In particular, when on the left side of an obligation, the existential binding is added to the environment; there is *no* support for existentials on the right. The way in which type checking introduces existentials guarantees that they always appear on the left.

References and Arrays. As in Alias Types [32], we enforce the invariant that distinct strong locations do not alias, so references to them are *never* related by subtyping. In contrast, weak locations describe zero or more locations, and it is safe to treat `null` as a subtype of *any* weak location. That is, weak references are *nullable* but strong ones are not. Arrays are invariant in their type parameter, as usual, but can be related with additional predicates. For example, $\{\nu :: \mathit{Arr}(\mathit{Int}) \wedge len(\nu) = 2\}$ *is* a subtype of $\{\nu :: \mathit{Arr}(\mathit{Int})\}$.

Heaps. The world subtyping judgment $\Gamma \vdash T_1 / (H, \hat{h}_1) \sqsubseteq T_1 / (H, \hat{h}_2)$ (in Appendix A) checks that T_1 is a subtype of T_2 and that the heaps agree on the "deep" part H. Then, it checks that the structure of the "shallow" parts match (modulo permutation of bindings, checked by the $\equiv$ operator) and creates a substitution π of binders from $\hat{h}_2$ to $\hat{h}_1$. Finally, the heap bindings, which can be though of as dependent tuples, are embedded as formulas and checked by implication.

4.3 Value Typing

The value typing judgment $\Gamma;\ \Sigma;\ \Omega \vdash v :: T$ (Figure 7) verifies that the value v has type T in the given environments. Since values do not produce any effects, this judgment does *not* produce an output heap environment. Each primitive constant c has a type, denoted by $ty(c)$, that is used by T-Const. In our implementation, the standard prelude (`basics.dref`, `objects.dref`, and `prelude.dref`) comprise $ty(c)$. The rules T-Var and T-Extend are standard. The T-Loc rule assigns run-time location r (which appears during evaluation, but not in source programs) a reference type corresponding to its compile-time location, using the mapping StaticLoc. The rule T-Fun uses an empty label environment to type check function bodies, so that break expressions cannot cross function boundaries. The procedure Snapshot strips all of the binders from the input heap type $\hat{\Sigma}_1$ to add to the type environment and produces a heap environment Σ_1 for type checking the body. Dually, to check that the resulting world T_2/Σ_2 satisfies W_2, the world satisfaction relation (along with its helper, heap satisfaction, defined in Appendix A) checks that all bindings have the appropriate structure (modulo permutation of bindings, related by the $\equiv$ relation) and that the values in Σ_2 satisfy the types in W_2.

4.4 Expression Typing

The expression typing judgment $\Gamma;\ \Sigma;\ \Omega \vdash v :: T/\Sigma'$ (Figure 8) verifies that the evaluation of expression e produces a value of type T *and* a new heap environment Σ'. Aside from the rules for label and break expressions, label environments Ω play no interesting role. We consider several sets of typing rules in turn.

Imperative Operations. In System !D, we require that *all* imperative operations go through strong locations. Alternatively, we could define analogs of the following rules for weak locations, but we choose not to since we provide a mechanism for temporarily *thawing* weak locations anyway. The rules T-REF, T-DEREF, and T-SETREF manipulate strong locations that bind "bare" values with no prototype link. To check the reference allocation `ref` ℓ v, the rule T-REF ensures that ℓ is not already bound in the heap, and then adds a binding that records exactly the value being stored. The rule T-DEREF checks that the given value is a reference to a location on the heap, and then retrieves the stored value; this is the imperative analog to the "selfifying" T-VAR rule. The rule T-SETREF strongly updates a heap location. The rule T-NEWOBJ stores the value v_1 in the heap at location ℓ_1 *along with* a prototype link to the location ℓ_2 that v_2 refers to. Although no typing rules manipulate locations with prototype links, several primitive functions do (*e.g.* `getElem`, `setElem`, *etc.*).

Thaw / Freeze. To safely allow a weak location $\tilde{\ell}$ to be treated *temporarily* as strong, System !D ensures that $\tilde{\ell}$ has at most one corresponding *thawed* location at a time; if there is none, we say $\tilde{\ell}$ is *frozen*. The rule T-THAW thaws $\tilde{\ell}$ to a strong location ℓ (which we syntactically require be distinct from all other thawed locations for $\tilde{\ell}$) and updates the heap environment with thaw state `thwd` ℓ to track the correspondence. Subtyping allows `null` weak references, so the output type is `null` if the original reference is; otherwise, it is a reference to ℓ. Finally, the new heap also binds a value x of type T, the invariant for all values stored at $\tilde{\ell}$, and the output type introduces an *existential* so that x is in scope in the new heap. The rule T-FREEZE serves two purposes, to merge a strong location ℓ into a weak (frozen) location $\tilde{\ell}$ and to *re-freeze* a thawed (strong) location ℓ that originated from $\tilde{\ell}$, as long as the heap value stored at ℓ satisfies the invariant required by $\tilde{\ell}$. Compared to the presentation in [2], we have combined freeze and re-freeze into a single `freeze` expression that includes an explicit thaw state θ.

The result of thawing a weak location is *either* a strong reference or `null`. Although we could statically require that all strong references be non-`null` before use (to rule out the possibility of null-dereference exceptions), we choose to allow `null` references to facilitate idiomatic programming. Therefore, we modify the input type for the object primitives in `objects.dref` to allow a `null` argument. For example, consider the updated input type for `hasPropObj` below, where $T? \stackrel{\circ}{=} \{T(\nu) \vee \nu = \texttt{null}\}$. Notice that we add the predicate $x \neq \texttt{null}$ to the *output* type, because if `hasPropObj` evaluates without raising an exception, then x is guaranteed to be non-`null`. In this way, System !D precisely tracks the invariants of thawed objects (*cf.* `passengers` from § 2.7).

$$\begin{aligned}&(x{:}\mathit{Ref}?,\ k{:}\mathit{Str})/(x \mapsto \langle d{:}\mathit{Dict}, \dot{x}\rangle)\\ &\to \{x \neq \texttt{null} \wedge (\nu \textbf{ iff } \mathit{ObjHas}(d, k, \mathit{cur}, \dot{x}))\}/\mathit{same}\end{aligned}$$

Function Application. The T-APP rule does quite a bit of heavy lifting to check the application $[\overline{T}; \overline{m}; \overline{\Sigma}]\ v_1\ v_2$. The procedure $\mathsf{Freshen}$ generates fresh binders for the output heap type so that the output heap bindings at different call sites do not collide. The number of type, location, and heap parameters must match the number of type, location, and heap variables of the function type of v_1. Furthermore, the premise $\Gamma \vdash [\overline{m}/\overline{M}]$ ensures that the list of locations provides strong (resp. weak) locations for strong (resp. weak) variables, and that the list contains no duplicates to ensure the soundness of strong updates [32]. The substitution of parameters for polymorphic variables proceeds in three steps. First, the type variables $\overline{A}$ are instantiated with the type parameters $\overline{T}$ using the procedure Inst, which instantiates type variables inside has-type predicates. Second, the location variables $\overline{M}$ are replaced with the parameters $\overline{m}$ by ordinary substitution. Third, the heap variables $\overline{H}$ are instantiated with the parameters $\overline{\Sigma}$ using a procedure HInst that substitutes heap environment bindings for heap variables. The substitution of heaps presents two complications. First, $\mathit{HeapHas}$ and $\mathit{HeapSel}$ may refer to arbitrary heaps rather than just heap variables, as required. These *pre-types* (and *pre-formulas*, *pre-heaps*, *etc.*) are unrolled to types using the procedure Unroll, discussed shortly. Second, since the domains of the heap parameters may overlap, well-formedness checks are required to ensure that locations are not bound multiple times after instantiation. After these substitutions, $\Gamma \models \Psi$ checks that the environment contains all heap bindings required by the weak heap of the function type.

From this point, the rest of the rule is fairly familiar. First, the argument type T_2 and current heap Σ are checked to satisfy the input world $T'_{11}/\hat{\Sigma}'_{11}$. If so, the substitution π maps binders from the input heap to the corresponding ones in the current heap Σ. Finally, the substitution for heap and argument binders is applied to the output world, and its heap bindings are collected using $\mathsf{Snapshot}$ and added to the output type using existentials.

Heap Symbol Unrolling. After heap instantiation, pre-types contain symbols of the form $\mathit{HeapHas}(\Sigma, \ell, v)$ and $\mathit{HeapHas}(\Sigma, \ell, v)$, which must be unrolled to eliminate heap

Expression Typing $\boxed{\Gamma;\ \Sigma;\ \Omega \vdash e :: T/\Sigma'}$

$$\text{[T-Ref]}\ \frac{\ell \notin \mathit{dom}(\Sigma) \qquad \Gamma;\ \Sigma;\ \Omega \vdash v :: T}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{ref}\ \ell\ v :: \{x \,|\, x :: \mathit{Ref}\,\ell\}/\Sigma \oplus (\ell \mapsto v)}
\qquad
\frac{\Gamma;\ \Sigma;\ \Omega \vdash v :: \{y \,|\, y :: \mathit{Ref}\,\ell\} \qquad \Sigma \equiv \Sigma_0 \oplus (\ell \mapsto v')}{\Gamma;\ \Sigma;\ \Omega \vdash\ \texttt{deref}\ v\ :: \{y \,|\, y = v\}/\Sigma}\ \text{[T-Deref]}$$

$$\text{[T-Setref]}\ \frac{\begin{array}{c}\Gamma;\ \Sigma;\ \Omega \vdash v_1 :: \{y \,|\, y :: \mathit{Ref}\,\ell\} \\ \Gamma;\ \Sigma;\ \Omega \vdash v_2 :: T \\ \Sigma \equiv \Sigma_0 \oplus (\ell \mapsto v) \qquad \Sigma' = \Sigma_0 \oplus (\ell \mapsto v_2)\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash v_1 := v_2 :: \{x \,|\, x = v_2\}/\Sigma'}
\qquad
\frac{\begin{array}{c}\ell_1 \notin \mathit{dom}(\Sigma) \qquad \Gamma;\ \Sigma;\ \Omega \vdash v_1 :: T \\ \Gamma;\ \Sigma;\ \Omega \vdash v_2 :: \{y \,|\, y :: \mathit{Ref}\,\ell_2\} \qquad \Sigma \equiv \Sigma_0 \oplus (\ell_2 \mapsto \langle v_2, \ell_3 \rangle) \\ \Sigma' = \Sigma \oplus (\ell_1 \mapsto \langle v_1, \ell_2 \rangle)\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{newobj}\ \ell_1\ v_1\ v_2 :: S_2/\Sigma'}\ \text{[T-NewObj]}$$

$$\text{[T-Freeze]}\ \frac{\begin{array}{c}\Gamma;\ \Sigma;\ \Omega \vdash v :: \{x \,|\, x :: \mathit{Ref}\,\ell\} \qquad \Psi(\tilde{\ell}) = (T, \ell') \\ \Sigma \equiv \Sigma_0 \oplus (\tilde{\ell} \mapsto \theta) \oplus (\ell \mapsto \langle v', \ell' \rangle) \qquad \theta = \texttt{frzn} \text{ or } \theta = \texttt{thwd}\ \ell \\ \Gamma;\ \Sigma;\ \Omega \vdash v' :: T \qquad \Sigma' = \Sigma_0 \oplus (\tilde{\ell} \mapsto \texttt{frzn})\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{freeze}\ \tilde{\ell}\ \theta\ v :: \mathit{SafeRef}\ \tilde{\ell}/\Sigma'}
\qquad
\frac{\begin{array}{c}\Gamma;\ \Sigma;\ \Omega \vdash v :: \mathit{Ref}\,\tilde{\ell} \qquad \Psi(\tilde{\ell}) = (T, \ell') \\ \Sigma \equiv \Sigma_0 \oplus (\tilde{\ell} \mapsto \texttt{frzn}) \\ \Sigma' = \Sigma_0 \oplus (\tilde{\ell} \mapsto \texttt{thwd}\ \ell) \oplus (\ell \mapsto \langle x, \ell' \rangle) \\ S = \{y \,|\, \texttt{ite}\ (v = \texttt{null})\ (y = \texttt{null})\ (y :: \mathit{Ref}\,\ell)\}\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{thaw}\ \ell\ v :: \exists x{:}T.\ S/\Sigma'}\ \text{[T-Thaw]}$$

$$\text{[T-Val]}\ \frac{\Gamma;\ \Sigma;\ \Omega \vdash v :: T}{\Gamma;\ \Sigma;\ \Omega \vdash v :: T/\Sigma}
\qquad
\frac{\Gamma;\ \Sigma;\ \Omega \vdash e :: T/\Sigma}{\Gamma;\ \Sigma;\ \Omega \vdash e\ \texttt{as}\ T :: T/\Sigma}\ \text{[T-As]}
\qquad
\frac{\Gamma;\ \Sigma;\ \Omega \vdash e :: S/\Sigma' \qquad \Gamma \vdash S \sqsubseteq T \qquad \Gamma \vdash T}{\Gamma;\ \Sigma;\ \Omega \vdash e :: T/\Sigma'}\ \text{[T-Sub]}$$

$$\text{[T-If]}\ \frac{\begin{array}{c}\Gamma;\ \Sigma;\ \Omega \vdash v :: S \qquad \Gamma, \mathit{truthy}(v);\ \Sigma;\ \Omega \vdash e_1 :: T_1/\Sigma_1 \\ \Gamma, \mathit{falsy}(v);\ \Sigma;\ \Omega \vdash e_2 :: T_2/\Sigma_2 \qquad T/\Sigma' = T_1/\Sigma_1 \sqcup T_2/\Sigma_2\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 :: T/\Sigma'}
\qquad
\frac{\begin{array}{c}\Gamma;\ \Sigma;\ \Omega \vdash e_1 :: T_1/\Sigma_1 \\ \Gamma, x{:}T_1;\ \Sigma_1;\ \Omega \vdash e_2 :: T_2/\Sigma_2 \qquad \Gamma \vdash T_2/\Sigma_2\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 :: T_2/\Sigma_2}\ \text{[T-Let]}$$

[T-App]

$$\frac{\begin{array}{c}\Gamma;\ \Sigma;\ \Omega \vdash v_1 :: \{y \,|\, y :: \forall[\overline{A};\overline{M};\overline{H}]\ \Psi/x{:}T_{11}/\hat{\Sigma}_{11} \to W_{12}\} \qquad \Gamma;\ \Sigma;\ \Omega \vdash v_2 :: T_2 \\ \Gamma \vdash [\overline{T}/\overline{A}] \qquad \Gamma \vdash [\overline{m}/\overline{M}] \qquad \Gamma \vdash [\overline{\Sigma}/\overline{H}] \qquad W'_{12} = \mathsf{Freshen}(W_{12}) \\ (T'_{11}, \hat{\Sigma}'_{11}, W''_{12}, \Psi') = \mathsf{Unroll}(\mathsf{HInst}(\mathsf{LInst}(\mathsf{TInst}((T_{11}, \hat{\Sigma}_{11}, W'_{12}, \Psi), \overline{A}, \overline{T}), \overline{M}, \overline{m}), \overline{H}, \overline{\Sigma})) \\ \Gamma \vdash \hat{\Sigma}'_{11} \qquad \Gamma \vdash W''_{12} \qquad \Gamma \vDash \Psi' \qquad \Gamma \vdash T_2/\Sigma \vDash T'_{11}/\hat{\Sigma}'_{11};\ \pi \qquad W''_{12} = (T_{12}, \hat{\Sigma}_{12}) \qquad \pi' = \pi[v_2/x] \qquad \mathsf{Snapshot}(\pi'\hat{\Sigma}_{12}) = (\overline{z}{:}\overline{S}, \Sigma_{12})\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash [\overline{T};\overline{m};\overline{\Sigma}]\ v_1\ v_2 :: \exists \overline{z}{:}\overline{S}.\ \pi' T_{12}/\Sigma_{12}}$$

$$\begin{array}{rcl}
\mathsf{Unroll}(\mathit{HeapHas}((H,h),\ell,k)) & = & \mathsf{UnrollHas}(H,h,\ell,k) \\
\mathsf{Unroll}(\psi(\mathit{HeapSel}((H,h),\ell,k))) & = & \mathsf{UnrollSel}(\psi,H,h,\ell,k) \\
\mathsf{UnrollHas}(H, h \oplus (\ell \mapsto \langle d, \ell' \rangle), \ell, k) & = & \mathit{has}(d,k) \vee \mathsf{UnrollHas}(H,h,\ell',k) \\
\mathsf{UnrollSel}(\psi, H, h \oplus (\ell \mapsto \langle d, \ell' \rangle), \ell, k) & = & \textbf{if}\ \mathit{has}(d,k)\ \textbf{then}\ \psi(\mathit{sel}(d,k)) \\
 & & \textbf{else}\ \psi(\mathit{HeapSel}(H,\ell,k))
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{UnrollHas}(H,h,\circ,k) & = & \mathit{false} \\
\mathsf{UnrollHas}(H,h,\ell,k) & = & \mathit{HeapHas}(H,\ell,k) \\
\mathsf{UnrollSel}(\psi,H,h,\circ,k) & = & \texttt{undefined} \\
\mathsf{UnrollSel}(\psi,H,h,\ell,k) & = & \psi(\mathit{HeapSel}(H,\ell,k))
\end{array}$$

Figure 8. Expression type checking for System !D

constraints from these symbols. The procedure Unroll recursively walks pre-formulas, pre-types, and pre-heaps and applies UnrollHas and UnrollSel, which unroll heaps by following prototype links within heap bindings, thus precisely matching the semantics of object key membership and lookup. We write the location $\circ$ for the root of the prototype hierarchy. We use the notation $\psi(p)$ to refer to a *formula context* ψ, a formula with a hole, filled with p.

Existentials. We have seen that T-APP and T-THAW conveniently use existentials binders into scope; T-LET does the same to describe the variable x that goes out of scope after the body expression is checked. Alternatively, the more traditional approach (*e.g.* [8]) requires that the variable be eliminated (*e.g.* via subsumption), but using existentials follows the standard implementation strategy [25] more closely. The T-IF rule uses a *join* operator (not shown) that combines the output worlds of the branches by rearranging existentials to ensure that the resulting world is *prenex-quantified*. This ensures that existentials only appear on the left side of subtyping obligations. For example, the join of $(\exists x_1{:}T_1.\ \mathit{Top}/(\ell \mapsto x_1))$ and $(\exists x_2{:}T_2.\ \mathit{Top}/(\ell \mapsto x_2))$ is $(\exists y{:}\{T_1(\nu) \vee T_2(\nu)\}.\ \mathit{Top}/(\ell \mapsto y))$.

Exceptions and Labels. We omit these rules from Figure 8, but they are standard. The T-LABEL rule for $@x : e$ binds $@x$ to an expected world W in the label environment Ω' used to check e, and the T-BREAK rule for `break` $@x\ e'$ checks that e' satisfies the world $\Omega'(@x)$.

4.5 Type Soundness

Many standard *stuck* states are *not* stuck in JavaScript: a function can be applied with any number of arguments; an operator can be used with any values because of implicit coercion; and, all property lookups succeed (possibly producing `undefined`). Nonetheless, several (non-exceptional) stuck states remain: applying a non-function value; and retrieving a property for a non-object value. A progress statement for DJS would state that well-typed programs do not get stuck and can only fail with exceptions due to retrieving a property from `undefined` or `null`.[13]

We expect that System !D satisfies progress and preservation theorems, but we have not yet proven them. The process will likely be tedious but not require new proof techniques. Unlike System D, which introduced the problematic *nesting* of syntactic types inside uninterpreted formulas, System !D does not introduce any new mechanisms in the refinement logic. Furthermore, several variations of Alias Types [23, 32, 37], even in a dependent setting [31], have been proven sound, and we expect to re-use their techniques to prove the soundness of System !D.

5. Desugaring DJS to System !D

In Figure 9, we present the explicitly-typed abstract syntax of DJS along with desugaring rules $\langle\!\langle\ \mathtt{e}\ \rangle\!\rangle = e$ that translate DJS expressions $\mathtt{e}$ to System !D expressions e. Most of the desugaring rules follow λ_{JS} [21] closely, so we limit our discussion to the aspects most relevant to DJS; we refer the reader to their work for more details. We use the metavariable $I \mathrel{\hat{=}} [\overline{T};\overline{m};\overline{\Sigma}]$ to range over instantiation parameters for function application.

Objects. We use $pro(e) \mathrel{\hat{=}} \mathtt{getProp}\ (\langle\!\langle \mathtt{e} \rangle\!\rangle,\ \text{“}\mathtt{prototype}\text{”})$ to set the prototypes of fresh object and array literals, translated to `newobj` which creates values with prototype links. Our implementation inserts a fresh location if none is provided. The instantiation parameters are usually inferred by the type checker (§ 6).

Function Application. The rules DS-FUNCCALL and DS-METHCALL desugar “direct calls” and “method calls”, where $\ulcorner \mathtt{e} \urcorner \mathrel{\hat{=}} \mathtt{getElem}\ (\mathtt{e},\ \text{“}\mathtt{\_\_code\_\_}\text{”})$. Notice that non-receiver arguments are packed into a single “arguments” tuple. We write `window` for the “global object” supplied as the receiver for direct calls. JavaScript provides two native functions `apply` and `call` in `Function.prototype` that allow the caller to explicitly supply the receiver argument. We do not provide general support `apply` and `call` in DJS, because they require mechanisms beyond the scope of our (already-large) type system; for example, the latter accepts an arbitrary number of arguments. The primary benefit of (non-constructor) functions as objects in JavaScript is that they inherit `apply` and `call` from `Function.prototype`, but since we do not support them, we sacrifice little expressiveness if the type system treats every non-constructor function as a *scalar* function value, rather than an object with the function stored in “`__code__`”. Furthermore, we can then support the limited use of `apply` required for our recursive function idioms § 2.7 using the rule DS-APPLY that *syntactically* looks for “`apply`” and explicitly sets the receiver. Because the type systems prohibits (non-constructor) functions from being used as objects, there is no danger that the `apply` be “hijacked” by overwriting the “`apply`” property.

$$\langle\!\langle\ \mathtt{c}\ \rangle\!\rangle = \mathtt{c} \qquad \langle\!\langle\ \mathtt{x}\ \rangle\!\rangle = \mathtt{deref}\ \mathtt{\_x} \qquad \langle\!\langle\ \mathtt{e}_1 = \mathtt{e}_2\ \rangle\!\rangle = \langle\!\langle \mathtt{e}_1 \rangle\!\rangle := \langle\!\langle \mathtt{e}_2 \rangle\!\rangle$$

$$\langle\!\langle\ \mathtt{var\ x = e;\ e'}\ \rangle\!\rangle = \mathtt{let}\ \mathtt{\_x} = \mathtt{ref}\ a_x\ \langle\!\langle \mathtt{e} \rangle\!\rangle\ \mathtt{in}\ \langle\!\langle \mathtt{e'} \rangle\!\rangle$$

$$\langle\!\langle\ /\!*\!:I*/\ \mathtt{e}_1[\mathtt{e}_2]\ \rangle\!\rangle = /\!*\!:I*/\ \mathtt{getElem}\ (\langle\!\langle \mathtt{e}_1 \rangle\!\rangle,\ \langle\!\langle \mathtt{e}_2 \rangle\!\rangle)$$

$$\langle\!\langle\ /\!*\!:I*/\ \mathtt{e}_1[\mathtt{e}_2] = \mathtt{e}_3\ \rangle\!\rangle = /\!*\!:I*/\ \mathtt{setElem}\ (\langle\!\langle \mathtt{e}_1 \rangle\!\rangle,\ \langle\!\langle \mathtt{e}_2 \rangle\!\rangle,\ \langle\!\langle \mathtt{e}_3 \rangle\!\rangle)$$

$$\langle\!\langle\ /\!*\!:I*/\ \mathtt{delete}\ \mathtt{e}_1[\mathtt{e}_2]\ \rangle\!\rangle = /\!*\!:I*/\ \mathtt{delElem}\ (\langle\!\langle \mathtt{e}_1 \rangle\!\rangle,\ \langle\!\langle \mathtt{e}_2 \rangle\!\rangle)$$

$$\langle\!\langle\ /\!*\!:I*/\ \mathtt{e}_2\ \mathtt{in}\ \mathtt{e}_1\ \rangle\!\rangle = /\!*\!:I*/\ \mathtt{hasElem}\ (\langle\!\langle \mathtt{e}_1 \rangle\!\rangle,\ \langle\!\langle \mathtt{e}_2 \rangle\!\rangle)$$

$$\langle\!\langle\ /\!*\!:\ell*/\ \{\ \overline{\mathtt{e}_\mathtt{k}}\!:\!\overline{\mathtt{e}_\mathtt{v}}\ \}\ \rangle\!\rangle = \qquad \text{[DS-ObjLit]}$$
$$\mathtt{newobj}\ \ell\ \{\langle\!\langle \mathtt{e}_{\mathtt{k}0} \rangle\!\rangle \mapsto \langle\!\langle \mathtt{e}_{\mathtt{v}0} \rangle\!\rangle\ \mathtt{++}\ \cdots\ \}\ (pro(\mathtt{Object}))$$

$$\langle\!\langle\ /\!*\!:\ell*/\ [\ \overline{\mathtt{e}}\ ]\ \rangle\!\rangle = \qquad \text{[DS-ArrLit]}$$
$$\mathtt{newobj}\ \ell\ \{\text{“}0\text{”} \mapsto \langle\!\langle \mathtt{e}_0 \rangle\!\rangle\ \mathtt{++}\ \cdots\ \}\ (pro(\mathtt{Array}))$$

$$\langle\!\langle\ /\!*\!:I*/\ \mathtt{e}(\mathtt{e}_1,\ \ldots,\ \mathtt{e}_\mathtt{n})\ \rangle\!\rangle = \qquad \text{[DS-FuncCall]}$$
$$/\!*\!:I*/\ \ulcorner\langle\!\langle \mathtt{e} \rangle\!\rangle\urcorner\ (\mathtt{window},\ (\langle\!\langle \mathtt{e}_1 \rangle\!\rangle,\ \ldots,\ \langle\!\langle \mathtt{e}_\mathtt{n} \rangle\!\rangle))$$

$$\langle\!\langle\ /\!*\!:I*/\ \mathtt{e}[\mathtt{e}'](\mathtt{e}_1,\ \ldots,\ \mathtt{e}_\mathtt{n})\ \rangle\!\rangle = \qquad \text{[DS-MethCall]}$$
$$\begin{array}{l} \mathtt{let}\ obj = \langle\!\langle \mathtt{e} \rangle\!\rangle\ \mathtt{in} \\ \mathtt{let}\ m = \mathtt{getElem}\ (obj,\ \langle\!\langle \mathtt{e}' \rangle\!\rangle)\ \mathtt{in} \\ /\!*\!:I*/\ \ulcorner m \urcorner\ (obj,\ (\langle\!\langle \mathtt{e}_1 \rangle\!\rangle,\ \ldots,\ \langle\!\langle \mathtt{e}_\mathtt{n} \rangle\!\rangle)) \end{array}$$

$$\langle\!\langle\ /\!*\!:I*/\ \mathtt{e.apply}(\mathtt{e}_1,\ \ldots,\ \mathtt{e}_\mathtt{n})\ \rangle\!\rangle = \qquad \text{[DS-Apply]}$$
$$/\!*\!:I*/\ \ulcorner\langle\!\langle \mathtt{e} \rangle\!\rangle\urcorner\ (\langle\!\langle \mathtt{e}_1 \rangle\!\rangle,\ (\langle\!\langle \mathtt{e}_2 \rangle\!\rangle,\ \ldots,\ \langle\!\langle \mathtt{e}_\mathtt{n} \rangle\!\rangle))$$

$$\langle\!\langle\ \mathtt{function}\ \mathtt{F}(\overline{\mathtt{x}})\ /\!*\!:\#\mathtt{ctor}\ T*/\ \{\ \mathtt{e}\ \}\ \rangle\!\rangle = \qquad \text{[DS-Ctor]}$$
$$\begin{array}{l} \mathtt{let}\ f = \lambda(this, arguments). \\ \quad \mathtt{let}\ (\mathtt{\_x}_0,\ \ldots) = (\mathtt{ref}\ a_{x_0}\ (\mathtt{get}\ arguments\ \text{“}0\text{”}),\ \ldots)\ \mathtt{in} \\ \quad @return : \langle\!\langle \mathtt{e} \rangle\!\rangle\ \mathtt{in} \\ \mathtt{let}\ p = \mathtt{newobj}\ a_{Fproto}\ \{\}\ (pro(\mathtt{Object}))\ \mathtt{in} \\ \mathtt{let}\ d = \{\ \text{“}\mathtt{\_\_code\_\_}\text{”} = f\ \mathtt{as}\ \langle\!\langle T \rangle\!\rangle;\ \text{“}\mathtt{prototype}\text{”} = p\ \}\ \mathtt{in} \\ \mathtt{newobj}\ a_F\ d\ (pro(\mathtt{Function})) \end{array}$$

$$\langle\!\langle\ \mathtt{new}\ /\!*\!:\ell_{new}\ I*/\ \mathtt{e}(\mathtt{e}_1,\ \ldots,\ \mathtt{e}_\mathtt{n})\ \rangle\!\rangle = \qquad \text{[DS-New]}$$
$$\begin{array}{l} \mathtt{let}\ foo = \langle\!\langle \mathtt{e} \rangle\!\rangle\ \mathtt{in} \\ \mathtt{let}\ obj = \mathtt{newobj}\ \ell_{new}\ \{\}\ (foo\ \text{“}\mathtt{prototype}\text{”})\ \mathtt{in} \\ /\!*\!:I*/\ \ulcorner foo \urcorner\ (obj,\ (\langle\!\langle \mathtt{e}_1 \rangle\!\rangle,\ \ldots,\ \langle\!\langle \mathtt{e}_\mathtt{n} \rangle\!\rangle)) \end{array}$$

$$\langle\!\langle\ /\!*\!:T*/\ \mathtt{while}\ (\mathtt{e}_\mathtt{cond})\ \{\ \mathtt{e}_\mathtt{body}\ \}\ \rangle\!\rangle = \qquad \text{[DS-While]}$$
$$\begin{array}{l} @break : \mathtt{letrec}\ loop :: T = \lambda(). \\ \qquad\qquad \mathtt{if}\ \langle\!\langle \mathtt{e}_\mathtt{cond} \rangle\!\rangle\ \mathtt{then}\ (\langle\!\langle \mathtt{e}_\mathtt{body} \rangle\!\rangle; loop\ ()) \\ \qquad\qquad \mathtt{else}\ \mathtt{undefined}\ \mathtt{in}\ loop\ () \end{array}$$

$$\langle\!\langle\ \mathtt{return\ e}\ \rangle\!\rangle = \mathtt{break}\ @return\ \langle\!\langle \mathtt{e} \rangle\!\rangle \qquad \text{[DS-Return]}$$

$$\langle\!\langle\ \mathtt{break}\ \rangle\!\rangle = \mathtt{break}\ @break\ \mathtt{undefined} \qquad \text{[DS-Break]}$$

$$\langle\!\langle\ /\!*\!:\#\mathtt{thaw}\ \ell\ \mathtt{e}*/\ \rangle\!\rangle = \mathtt{thaw}\ \ell\ \langle\!\langle \mathtt{e} \rangle\!\rangle \qquad \text{[DS-Thaw]}$$

$$\langle\!\langle\ /\!*\!:\#\mathtt{freeze}\ \tilde{\ell}\ \theta\ \mathtt{e}*/\ \rangle\!\rangle = \mathtt{freeze}\ \tilde{\ell}\ \theta\ \langle\!\langle \mathtt{e} \rangle\!\rangle \qquad \text{[DS-Freeze]}$$

$$\langle\!\langle\ \mathtt{assert(e)}\ \rangle\!\rangle = \langle\!\langle \mathtt{e} \rangle\!\rangle\ \mathtt{as}\ \{\nu = \mathtt{true}\} \qquad \text{[DS-Assert]}$$

Figure 9. Desugaring DJS to System !D (selected rules)

Functions and Construction. DS-CTOR uses fresh locations a_F and $a_{F_{proto}}$ for a constructor function object and its “`prototype`” object used when constructing objects (DS-NEW); our implementation chooses the predictable locations

[13] We can provide the stronger guarantees that only bound keys are retrieved and only non-`null` objects are accessed (thus ruling out the possibility of `null` dereference exceptions) simply by changing the types of object primitives appropriately.

Adapted Benchmark	Un	Ann	Queries	Time
JS: The Good Parts				
`prototypal`	18	36	731	2
`pseudoclassical`	15	23	706	2
`functional`	19	43	862	8
`parts`	11	20	605	3
SunSpider				
`string-fasta`	10	18	263	1
`access-binary-trees`	34	50	2389	23
`access-nbody`	129	201	4225	39
V8				
`splay`	17	36	571	1
Google Closure Library				
`typeOf`	15	31	1975	52
Other				
`negate`	9	9	296	1
`passengers`	9	19	310	3
`counter`	16	24	272	1
`dispatch`	4	8	219	1
Totals	306	518	13424	137

Figure 10. Benchmarks (Un: LOC without annotations; Ann: LOC with annotations; Queries: Number of Z3 queries; Time: Running time in seconds)

`lFObj` and `lFooProto`. The function body is labeled to facilitate how DS-RETURN desugars `return` statements. We omit a similar rule for recursive (non-constructor) functions. The syntax of construction mentions a location ℓ_{new} for the new object and instantiations for the application, all of which is usually inferred by the type checker (§ 6).

Loops. Looping constructs desugar to recursive functions (we write `letrec` as syntactic sugar for the standard encoding using `fix`). As such, a (function type) annotation describes the invariants that hold before and after each iteration. A label around the desugared loop facilitates how DS-BREAK desugars `break` statements; we elide the similar mechanism for `continue`.

6. Evaluation

In this section, we describe our implementation, the benchmarks we have annotated and type checked so far that demonstrate the expressiveness of DJS, and identify several ways for future work to improve the tool.

6.1 Implementation

We have implemented a type checker for DJS, available at `ravichugh.com/nested`, that is currently approximately 6600 (non-whitespace, non-comment) lines of OCaml code. We borrow the λ_{JS} [21] JavaScript parser, use their desugaring as a starting point for our own, and use the Z3 SMT solver [11] to discharge logical validity queries. We specify the System !D primitive functions in the files `basics.dref` and `objects.dref`, and JavaScript built-in functions like `Object.prototype.hasOwnProperty` in `prelude.js` (desugared to `prelude.dref`). These three files comprise a standard prelude included with every desugared DJS program for type checking.

Local Inference. Function definitions require explicit type annotations, and we employ *bidirectional type checking* [29] techniques to infer types for local expressions. At a function application, we infer missing type and location parameters by "greedily" matching the types of arguments against any $Arr(T)$ and $Ref\,L$ type terms in the the declared input type and input heap. Because these type terms are invariant in their parameters, the greedy choice is always the right one. For a function type with exactly one heap variable H (like all the ones we have encountered) and input heap type $(H, \hat{h})$, we infer the corresponding heap argument by simply collecting all locations in the current heap environment that do *not* match the explicit location bindings in $\hat{h}$. In our benchmarks, we are able to omit most type and location arguments and all heap arguments.

6.2 Benchmarks

To demonstrate the expressiveness of DJS, we have annotated and checked several small examples — inspired by *JavaScript: The Good Parts* [10], Google Closure Library [18], and the SunSpider [34] and V8 [19] benchmarks — that exercise a variety of invariants, besides those demonstrated by previous examples (*e.g.* `negate`, `passengers`, *etc.*). We also ported the `counter` and `dispatch` examples from System D [8] to DJS to demonstrate the nesting of function types inside objects with dynamic keys. Figure 10 summarizes our results, where for each example: "Un" is the number of (non-whitespace, non-comment) lines of code in the *unannotated* benchmark; "Ann" is the lines of code in the annotated DJS version (including comments because they contain DJS annotations); "Time" is the running time rounded to the nearest second; and "Queries" is the number of validity queries issued to Z3 during type checking.

Expressiveness. We highlight some of the features of DJS that our benchmarks leverage. Besides the prototypal pattern discussed in § 2.5, Crockford [10] presents three additional inheritance patterns using JavaScript's construction mechanism. Each of these examples relies on the support for imperative, prototype-based objects in DJS.

The behavior of the `typeOf` function is like the `typeof` operator except that it returns the more informative result "`null`" for `null` and "`array`" for arrays; the operator returns "`object`" in both cases. The type specification for `typeOf` depends on the ability to express intersections of function types in DJS, and verifying it requires control-flow tracking in the presence of mutation as well as a precise specification for the native (ES5) function `Array.isArray`, which we model in `prelude.js`.

The `makeCumulative` function in `string-fasta.js` iterates over an object with an unknown number of keys that all store integers, and sums them in place within the object. While iterating over the keys of the object, the function uses a variable to store the key from the *previous* iteration, a subtle invariant that DJS is able to express by describing the heap before and after each iteration. Compared to the original version, we allow the bindings to store arbitrary values and use a tag-test to sum only the integer bindings. To specify the original version requires universally quantified formulas, which DJS avoids to retain decidable type checking.

The `splay` benchmark defines the following interesting tree node constructor. Rather than initializing each "own" object with `null` left and right subtrees, the constructor's prototype object stores the defaults.

```
function Node(k,v) { this.k = k; this.v = v; }
Node.prototype.left = null;
Node.prototype.right = null;
```

After construction, however, `Node`s are often extended with explicit subtrees. Using the flexibility of refinements, we assign each `Node` a type with the predicate $has(\nu, \text{"left"}) \Rightarrow sel(\nu, \text{"left"}) :: Ref\,\tilde{\ell}$, where $\tilde{\ell}$ is the weak location that describes `Node`s, to ensure that retrieving the "`left`" key produces another `Node` regardless of whether it is stored on the object or not (and similarly for "`right`").

Our largest example is `access-nbody`, which defines a constructor function `NBodySystem` that creates a container object to store an array of `Body` objects. The prototypes of both constructors are augmented with methods, and the thaw/freeze mechanism is heavily used while iterating over the array of `Body` objects to read and write their fields.

6.3 Annotation Burden

As Figure 10 shows, our annotated benchmarks are approximately 1.7 times as large (70% overhead) as their unannotated versions on average. In our experience, a significant majority of the annotation burden is boilerplate — unrelated to the interesting typing invariants — that fall into the following five patterns. Our implementation includes preliminary support for several patterns by tracking a limited amount of type information during desugaring (that require *no* changes to type checking), which has already significantly reduced the annotation overhead. There is plenty of room for further improvements in future work, however.

Closures. If a function refers to a variable from an outer scope, its heap type must explicitly list its location and type. In the following example, the desugarer uses the predictable locations `&pi` and `&e` when desugaring `pi` and `e`, and the function type must contain the binding for `&pi`.

```
var pi = 3.14, e = 2.718;
/*: Top/(&pi ↦ n:Num) → Num/same */
function getPi() { return pi; }
```

To ease this burden, we collect the free variables in each function definition and automatically add the corresponding heap bindings that are missing. In situations where we cannot insert a suitably precise type for a location, we allow the programmer to annotate a variable declaration `var i = /*: T */ e` and we propagate T to functions that refer to `i`.

Loops. Because loops desugar to functions, they require a heap type annotation (like for arbitrary closures) to describe invariants that hold before and after every iteration. We infer heap types for basic patterns like the following.

```
/*: (&i ↦ {Int(ν) ∧ i ≥ 0}) ⊕ (&sum ↦ Num)
      ⊕ (&ns ↦ Ref ℓa)
      ⊕ (ℓa ↦ ⟨{ν :: Arr(Num) ∧ packed(ν)}, ℓap⟩) */
for (i=0; i < ns.length; i++) { sum += ns[i]; }
```

Thaw / Freeze. Every weak reference must first be thawed before access, which quickly becomes burdensome. As a simple aid, we surround an access to a weak reference with thaw and freeze operations, which is sufficient for simple cases involving reads and weak updates. For more complex invariants, like the relationship *between* accesses to a weak reference (as in the `passengers` example from § 2.7), a single thaw and freeze pair must surround *both* accesses. In the future, we plan to insert these operations at basic block and function boundaries in the style of [31] so that objects are tracked with strong references as long as possible.

Untampered Natives. Functions that use JavaScript primitive functions like `Object.prototype.hasOwnProperty` and `Array.prototype.push` and expect them *not* to be overwritten, must explicitly constrain their input heaps as such. In most cases, programmers expect natives to remain "untampered," so desugaring could augment all function types with these constraints.

Constructor Prototypes. The purpose of a constructor `C` is to allow its instances to inherit the properties of `C.prototype`, but functions like `useC` that use such an instance must then explicitly list the type of `C.prototype`.

```
/*: #define T_C {Dict(ν) ∧ ··· } */
/*: #ctor this:Ref/(this ↦ ⟨Emp, a_Cpro⟩)
        → {ν = this}/(this ↦ ⟨T_C, a_Cpro⟩) */
function C() { ...; return this; }
C.prototype.f = /*: T_f */ ...;
C.prototype.g = /*: T_g */ ...;

/*: x:Ref/(x ↦ ⟨T_C, a_Cpro⟩) ⊕
      (a_Cpro ↦ ⟨T_f(sel(ν, "f")) ∧ T_g(sel(ν, "g")), ℓop⟩)
  → Top/same */
function useC(x) { ... x.f ... }
```

This is a predictable pattern that should be easy to incorporate into desugaring, though we have not yet done so.

6.4 Performance

The running time of our type checker is acceptable for small examples, but less so as the number of queries to the SMT solver increases. We have not yet spent much effort to improve performance, but we have implemented a few optimizations that have already reduced the number of SMT queries. First, even though desugaring works perfectly well *without* any type information, we use DJS type annotations to translate object and array operations to specific primitives, where possible, rather than the more general ones (*e.g.* `getPropObj` and `getIdxArr` rather than `getElem`) so that type checking has fewer cases to try, and we insert type and location parameters so that they need not be inferred. Second, we modify the T-VAR rule, which normally assigns the "selfified" type $\{\nu = x\}$ to variable x that is already bound in Γ. Although this precision is crucial, the variable x often has a simple syntactic type (*e.g.* $\mathit{Ref}\,\ell$) that is "hidden" behind the equality. Instead, if $\Gamma(x)$ is of the form $\{\nu :: U \wedge p\}$, we assign $\{\nu :: U \wedge p \wedge \nu = x\}$ so that subsequent typing rules can *syntactically* look for U rather than going through additional SMT queries as in the general case [8]. We expect that syntactically handling more common cases will further improve performance. For example, even though the dynamic keys are crucial in certain situations, many examples use object types with finite and fixed key names, which the type checker should be able to handle with far fewer queries to the SMT solver than in the current implementation.

7. Conclusion and Future Work

In this paper, we have shown how to scale up prior work on System D — a type system for dynamic languages in a functional setting — to the real-world JavaScript setting — with imperative updates, prototype-based objects, and arrays — through a combination of strong updates and prototype chain unrolling. We have demonstrated that our new system, System !D, is expressive enough to support the invariants from a series of small but varied examples drawn from existing JavaScript benchmarks. We have found that the full range of features in DJS are indeed required, but that many examples fall into patterns that do not simultaneous exercise all features. Therefore, we believe that future work on desugaring and on type checking can treat common cases specially in order to reduce the annotation burden and running time, and fall back to the full expressiveness of the system when necessary. We believe that Dependent JavaScript is the most promising approach, to date, for supporting real-world dynamic languages like JavaScript.

Features for Future Work. DJS already supports a large subset of JavaScript that can be used for projects where all the code is controlled (*e.g.* server-side applications), and future work on integrating with run-time environments could allow DJS code to run alongside full untyped JavaScript. Next, we describe several features that we currently do not support, in addition to general use of `apply` and `call` as discussed in § 5.

To allow mutation of prototype links via the non-standard "`__proto__`" property, we could add a `setproto` expression to the language and detect cycles during heap unrolling.

The `eval` statement allows a string to be parsed and executed, which is useful but dangerous if misused. Since DJS is flow-sensitive, we can constraint `eval` with heap invariants before and after the statement, and then perform *staged type checking* in the style of [7] at run-time.

ES5 introduces optional per-object and per-property attributes (for example, to prevent modifications or deletions) that can likely be incorporated into our encoding of dictionaries. One benefit of such an extension is that the type system could reason more precisely about *which* objects are in a prototype chain. For example, we could then allow non-array objects to bind unsafe strings as long as we prevent them from appearing in the prototype chain of arrays, thus weakening the distinction we impose between array and non-array objects (§ 2.6). A second benefit is that native objects could be marked as unmodifiable, statically enforcing the pattern they are usually "untampered" as discussed in § 6.

ES5 getters and setters interpose on object reads and writes. Since this is a deep change to the semantics of object operations (invoking arbitrary functions), adding general support for these will likely be heavyweight. Interestingly, one can think of our treatment of the special array "`length`" property (§ 2.6) as a built-in getter/setter.

Each function has an implicit `arguments` array that binds all parameters supplied by the caller, regardless of how many formals the function defines. Current ES6 proposals include a modified version, where an *explicit* parameter can bind a variable number of arguments *beyond* those named by formals, similar in style to Python.

The `x instanceof Foo` operator checks whether or not `Foo.prototype` is *somewhere* along the prototype chain of `x`. We could add a primitive to match these semantics.

Scalar values can be explicitly coerced by wrapper functions, such as `Boolean`, in addition to the implicit coercion we have discussed.

Undesirable Features. The last three features we discuss regularly compete for the title of worst among several "warts" in the language (*e.g.* [10]) that lead to confusing code and hard-to-detect bugs. Incidentally, the λ_{JS} translations of all three are straightforward and can be supported in DJS, but we see no reason to given their demerits.

The `with` statement adds the fields of an object to the current scope of a block, allowing them to be accessed without qualification. There is hardly a good reason to use this feature, and it is banned in ES5 "strict" mode.

All `var` declarations are implicitly lifted to the top of the enclosing function, resulting in "function scope" rather than lexical scope. Although simple to detect when `var`-lifting kicks in, we opt for the latter. ES6 will likely add an explicit `let` binding form that is not subject to lifting. In DJS, `var` is essentially the new `let` form, but we stick with the traditional syntax for familiarity.

For a "method call" `x.f(y)`, the receiver `x` is supplied for the `this` argument to the function, but for a "direct call" `x(y)`, JavaScript implicitly supplies the global object for `this`, masking common errors. We choose to statically reject direct calls to functions that require a `this` parameter.

8. Related Work

In this section, we discuss topics related to types for imperative dynamic languages, and hence strong updates and inheritance. The reader may refer to [8] for background on the challenging idioms of even functional dynamic languages and the solution that nested refinements provide.

Location Sensitive Types. The way we handle reference types draws from the approach of *Alias Types* [32], in which strong updates are enabled by describing reference types with abstract location names and by factoring reasoning into a flow-insensitive tying environment and a flow-sensitive heap. *Low-level liquid types* [31] employs their approach in the setting of a first-order language with dependent types. In contrast, our setting includes higher-order functions, and our formulation of heap types gives variable names to unknown heaps to reason about prototypes and gives names to *all* heap values, which enables the specification of precise relationships between values of *different* heaps; the heap binders of [31] allow only relationships between values in a *single* heap to be described.

The original Alias Types work also includes support for *weak references* that point to zero or more values, for which strong updates are not sound. Several subsequent proposals [2, 3, 13, 14, 31, 33] allow strong updates to weak references under certain circumstances to support temporary invariant violations. We adapt the *thaw/freeze* mechanism from [2] and [31] with mostly cosmetic changes.

Prototype Inheritance. Unlike early class-based languages, such as Smalltalk and C++, the (untyped) language Self allows objects to be extended after creation and feature prototype, or delegation, inheritance. Static typing disciplines for class-based languages (*e.g.* [1]) explicitly preclude object extension to retain soundness in the presence of *width subtyping*, the ability to forget fields of an object. To mitigate the tension between object extension and subtyping, several proposals [6, 17] feature quite a different flavor: the fields of an object are split into a "reservation" part, which may be added to an object but cannot be forgotten, and a "sealed part" that can be manipulated with ordinary subtyping. Our approach provides additional precision in two important respects. First, we precisely track prototype hierarchies, whereas the above approaches flatten them into a single collection of fields. Second, we avoid the separation of reservation and sealed fields but still allow subtyping, since "width subtyping" in System !D is simply logical implication over refinement formulas; forgetting a field — discarding a $has(d, k)$ predicate — does not imply that $\neg has(d, k)$, which guards the traversal of the prototype chain.

Typed Subsets of JavaScript. Several (syntactic) type systems for various JavaScript subsets have been proposed. Among the earliest is [35], which identifies silent errors that result from implicit type coercion and the fact that JavaScript returns `undefined` when trying to look up a non-existent key from an object. The approach in [4] distinguishes between *potential* and *definite* keys, similar to the reservation and sealed discussed above; this general approach has been extended with flow-sensitivity and polymorphism [39]. The notion of *recency types*, similar to Alias Types, was applied to JavaScript in [23], in which typing environments, in addition to heap types, are flow-sensitive. Prototype support in [23] is limited to the finite number of prototype links tracked by the type system, whereas the *heap symbols* in System !D enable reasoning about *entire* prototype hierarchies. Unlike System !D, all of the above systems provide global type inference; our system does not have principal types, so we can only provide local type inference [29]. ADsafety [30] is a type system for ADsafe, a JavaScript sandbox, that restricts access to some fields, tracking strings precisely [20] to describe sets of field names. Although expressive enough to check ADsafe, which heavily uses large object literals, they do not support strong update and so cannot reason about object extension. Unlike System !D, none of the above systems include dependent types, which are required to express truly dynamic object keys and precise control-flow based invariants.

Recent work on JavaScript verification uses separation logic [16] to track precise flow-sensitive invariants. They support only first-order programs, and the expressiveness of their logic takes them beyond automatic verification, thus requiring properties to be manually proved.

JavaScript Semantics. We chose the JavaScript "semantics-by-translation" of λ_{JS} [21] since it targets a conventional core language that has been convenient for our study. An alternate semantics [26] inherits unconventional aspects of the language specification [24] (*e.g.* "scope objects"), which complicates the formulation of static reasoning.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] A. Ahmed, M. Fluet, and G. Morrisett. L^3: a linear language with locations. *Fundamenta Informaticae*, 77(4):397–449, June 2007.

[3] A. Aiken, J. Kodumal, J. S. Foster, and T. Terauchi. Checking and inferring local non-aliasing. In *PLDI*, 2003.

[4] C. Anderson, S. Drossopoulou, and P. Giannini. Towards Type Inference for JavaScript. In *ECOOP*, pages 428–452, June 2005.

[5] G. M. Bierman, A. D. Gordon, C. Hritcu, and D. E. Langworthy. Semantic subtyping with an smt solver. In *ICFP*, 2010.

[6] V. Bono and K. Fisher. An imperative, first-order calculus with object extension. In *ECOOP*, 1998.

[7] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *Proceedings of PLDI 2009*, pages 50–62, 2009.

[8] R. Chugh, P. M. Rondon, and R. Jhala. Nested refinements: A logic for duck typing. In *POPL*, 2012.

[9] E. T.-. Committee. `http://www.ecmascript.org/community.php`.

[10] D. Crockford. *JavaScript: The Good Parts*. Yahoo! Press, 2008.

[11] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[12] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI 01*. ACM, 2001.

[13] M. Fahndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*. ACM, 2002.

[14] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI*, 2002.

[15] M. Furr, J. hoon (David) An, J. S. Foster, and M. W. Hicks. Static type inference for ruby. In *SAC*, pages 1859–1866, 2009.

[16] P. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for javascript. In *POPL*, 2012.

[17] P. D. Gianantonio, F. Honsell, and L. Liquori. A lambda calculus of objects with self-inflicted extension. In *OOPSLA*, 1998.

[18] Google. Closure library. `https://developers.google.com/closure/library`.

[19] Google. V8 benchmark. `http://v8.googlecode.com/svn/data/benchmarks/`.

[20] A. Guha, J. G. Politz, and S. Krishnamurthi. Fluid object types. 2011.

[21] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of javascript. In *ECOOP*, 2010.

[22] A. Guha, C. Softoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *ESOP*, 2011.

[23] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In *ECOOP*, 2010.

[24] E. International. *ECMAScript Language Specification, ECMA-262, 3rd ed.* 1999.

[25] K. W. Knowles and C. Flanagan. Compositional reasoning and decidable checking for dependent contract types. In *PLPV*, pages 27–38, 2009.

[26] S. Maffeis, J. Mitchell, and A. Taly. An operational semantics for JavaScript. In *APLAS*, 2008.

[27] J. McCarthy. Towards a mathematical science of computation. In *IFIP*, 1962.

[28] B. C. Pierce. *Types and Programming Languages*. 2002.

[29] B. C. Pierce and D. N. Turner. Local type inference. In *POPL*, 1998.

[30] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. Adsafety: type-based verification of javascript sandboxing. In *USENIX Security*, 2011.

[31] P. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL*, 2010.

[32] F. Smith, D. Walker, and J. Morrisett. Alias types. In *ESOP*, 2000.

[33] J. Sunshine, K. Naden, S. Stork, J. Aldrich, and E. Tanter. First-class state change in plaid. In *OOPSLA*, 2011.

[34] SunSpider. Javascript benchmark. `http://www.webkit.org/perf/sunspider/sunspider.html`.

[35] P. Thiemann. Towards a type system for analyzing javascript programs. In *ESOP*, 2005.

[36] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *ICFP*, 2010.

[37] D. Walker and J. Morrisett. Alias types for recursive data structures. 2000.

[38] S. yu Guo and B. Hackett. Fast and precise hybrid type inference for javascript. In *PLDI*, 2012.

[39] T. Zhao. Polymorphic type inference for scripting languages with object extensions. In *DLS*, 2011.

A. Type Checking: Additional Definitions

A.1 Well-Formedness

Well-Formed Types $\boxed{\Gamma \vdash T}$

$$\frac{\Gamma, x{:}\,\mathit{Top} \vdash p}{\Gamma \vdash \{x \,|\, p\}} \qquad \frac{\Gamma \vdash T \qquad \Gamma, x{:}\,\mathit{Top} \vdash S}{\Gamma \vdash \exists x{:}T.\, S}$$

Well-Formed Formulas (selected rules) $\boxed{\Gamma \vdash p}$

$$\frac{\Gamma \vdash w \qquad \Gamma \vdash U}{\Gamma \vdash w :: U} \qquad \frac{\Gamma;\, \Sigma \vdash \overline{w}}{\Gamma;\, \Sigma \vdash P(\overline{w})} \qquad \frac{H \in \Gamma \qquad \Gamma \vdash \ell \qquad \Gamma \vdash w}{\Gamma \vdash \mathit{HeapHas}(H, \ell, w)}$$

Well-Formed Locations $\boxed{\Gamma \vdash m}$

$$\frac{}{\Gamma \vdash a} \qquad \frac{}{\Gamma \vdash \tilde{a}} \qquad \frac{M \in \Gamma}{\Gamma \vdash M}$$

Well-Formed Type Terms $\boxed{\Gamma \vdash U}$

$$\frac{\begin{array}{c}\Gamma_1 = \Gamma, \overline{A}, \overline{M}, \overline{H} \qquad \Gamma_1 \vdash \Psi \qquad \Gamma_2 = \Gamma_1, \Psi \qquad \Gamma_2 \vdash T_1 \\ \Gamma_3 = \Gamma_2, x{:}\,\mathit{Top} \qquad \Gamma_3 \vdash \hat{\Sigma}_1 \qquad \hat{\Sigma}_1 = (H', \hat{h}) \\ \Gamma_4 = \Gamma_3, H', \mathsf{Binders}(\hat{h}) \qquad \Gamma_4 \vdash T_2 \qquad \Gamma_4 \vdash \hat{\Sigma}_2\end{array}}{\Gamma \vdash \forall [\overline{A}; \overline{M}; \overline{H}]\ \Psi / x{:}T_1 / \hat{\Sigma}_1 \to T_2 / \hat{\Sigma}_2}$$

$$\frac{A \in \Gamma}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash T}{\Gamma \vdash \mathit{Arr}(T)} \qquad \frac{\Gamma \vdash m}{\Gamma \vdash \mathit{Ref}\, m} \qquad \frac{}{\Gamma \vdash \mathit{Null}}$$

Well-Formed Weak Heaps $\boxed{\Gamma \vdash \Psi}$

$$\frac{}{\Gamma \vdash \varnothing} \qquad \frac{\Gamma \vdash \Psi \qquad \Gamma \vdash \tilde{\ell} \qquad \Gamma \vdash T \qquad \Gamma \vdash \ell}{\Gamma \vdash \Psi, (\tilde{\ell} \mapsto \langle T, \ell \rangle)}$$

Well-Formed Heaps and Heap Bindings $\boxed{\Gamma \vdash \hat{\Sigma}}$ $\boxed{\Gamma \vdash \hat{h}}$

$$\frac{H \in \Gamma \qquad \Gamma, \mathsf{Binders}(\hat{h}) \vdash \hat{h}}{\Gamma \vdash (H, \hat{h})}$$

$$\frac{\Gamma \vdash \ell \qquad \Gamma \vdash T}{\Gamma \vdash (\ell \mapsto x{:}T)} \qquad \frac{\Gamma \vdash \ell \qquad \Gamma \vdash T \qquad \Gamma \vdash \ell'}{\Gamma \vdash (\ell \mapsto \langle x{:}T, \ell' \rangle)} \qquad \frac{}{\Gamma \vdash \varnothing}$$

$$\frac{\Gamma \vdash \tilde{\ell}}{\Gamma \vdash (\tilde{\ell} \mapsto \texttt{frzn})} \qquad \frac{\Gamma \vdash \tilde{\ell} \qquad \Gamma \vdash \ell}{\Gamma \vdash (\tilde{\ell} \mapsto \texttt{thwd}\ \ell)} \qquad \frac{\Gamma \vdash \hat{h}_1 \qquad \Gamma \vdash \hat{h}_2}{\Gamma \vdash \hat{h}_1 \oplus \hat{h}_2}$$

A.2 World Subtyping

World Subtyping $\boxed{\Gamma \vdash W_1 \sqsubseteq W_2}$

$$\frac{\Gamma \vdash T_1 \sqsubseteq T_2 \qquad \Gamma \vdash \hat{h}_1 \sqsubseteq \hat{h}_2;\ \pi \qquad \Gamma, [\![\hat{h}_1]\!] \Rightarrow \pi[\![\hat{h}_2]\!]}{\Gamma \vdash T_1 / (H, \hat{h}_1) \sqsubseteq T_2 / (H, \hat{h}_2)}$$

Heap Binding Matching $\boxed{\Gamma \vdash \hat{h}_1 \sqsubseteq \hat{h}_2;\ \pi}$

$$\frac{}{\Gamma \vdash \varnothing \sqsubseteq \varnothing;\ [\,]} \qquad \frac{\hat{h}_1 \equiv \hat{h}_1' \qquad \hat{h}_2 \equiv \hat{h}_2' \qquad \Gamma \vdash \hat{h}_1' \sqsubseteq \hat{h}_2';\ \pi}{\Gamma \vdash \hat{h}_1 \sqsubseteq \hat{h}_2;\ \pi}$$

$$\frac{}{\Gamma \vdash (\tilde{\ell} \mapsto \theta) \sqsubseteq (\tilde{\ell} \mapsto \theta);\ [\,]}$$

$$\frac{}{\Gamma \vdash (\ell \mapsto x{:}T) \sqsubseteq (\ell \mapsto y{:}S);\ [x/y]}$$

$$\frac{}{\Gamma \vdash (\ell \mapsto \langle x{:}T, \ell' \rangle) \sqsubseteq (\ell \mapsto \langle y{:}S, \ell' \rangle);\ [x/y]}$$

A.3 Heap Satisfaction

Heap Satisfaction $\boxed{\Gamma \vdash \Sigma \vDash \hat{\Sigma};\ \pi}$ $\boxed{\Gamma \vdash h \vDash \hat{h};\ \pi}$

$$\frac{h \equiv h' \qquad \hat{h} \equiv \hat{h}' \qquad \Gamma \vdash h \vDash \hat{h};\ \pi}{\Gamma \vdash (H, h) \vDash (H, \hat{h});\ \pi} \qquad \frac{}{\Gamma \vdash \varnothing \vDash \varnothing;\ [\,]}$$

$$\frac{\Gamma \vdash h \vDash \hat{h};\ \pi}{\Gamma \vdash h \oplus (\ell \mapsto v) \vDash \hat{h} \oplus (\ell \mapsto x{:}T);\ \pi[v/x]}$$

$$\frac{\Gamma \vdash h \vDash \hat{h};\ \pi}{\Gamma \vdash h \oplus (\ell \mapsto \langle v, \ell' \rangle) \vDash \hat{h} \oplus (\ell \mapsto \langle x{:}T, \ell' \rangle);\ \pi[v/x]}$$

$$\frac{\Gamma \vdash h \vDash \hat{h};\ \pi}{\Gamma \vdash h \oplus (\tilde{\ell} \mapsto \theta) \vDash \hat{h} \oplus (\tilde{\ell} \mapsto \theta);\ \pi}$$