Dependent Types for JavaScript

Ravi Chugh and Ranjit Jhala

University of California, San Diego

Abstract. Dynamic languages like JavaScript, Python, and Ruby feature run-time tag-tests, higher-order functions, extensible objects, and imperative updates. In recent work on System D, we showed how to support many of these features in a functional setting. In this work, we define a new calculus, System !D (pronounced "D-ref"), that extends the previous one in two ways. First, we add flow-sensitive heap types to reason about *strong updates* to mutable variables. Second, we incorporate *heap unrolling* and *uninterpreted heap symbols* to precisely reason about *prototype-based objects*. We demonstrate that System !D is expressive by defining DJS, a large JavaScript subset that translates to System !D for type checking. Through several patterns advocated in the popular book *JavaScript: The Good Parts*, we show that System !D is a practical typed intermediate language for real-world, imperative dynamic languages with higher-order functions and extensible, prototype-based objects.

1 Introduction

Dynamic languages like JavaScript, Python, and Ruby are widely popular for building both client and server applications, in large part because of their combination of run-time tag tests, extensible objects, higher-order functions, and imperative variables. But as applications grow, the lack of static typing makes it difficult to achieve reliability, security, extensibility, and performance. In response, several authors have proposed syntactic type systems which provide static checking for various subsets of dynamic languages [16,22,12,26]. However, a major stumbling block in this line of work has been the inability to simultaneouly support dynamic idioms like run-time tag tests, ad-hoc unions, higher-order functions and objects (dictionaries) with *dynamic* value-dependent fields (keys).

Recently, we developed System D, a calculus for dynamic languages that supports the above dynamic idioms via a novel *dependent type* system [6]. Despite its expressiveness, System D type checking is fully automatic due to a novel divsion of the work into syntactic subtyping and SMT-based validity checking.

Unfortunately, System D is a purely functional calculus. Thus, it is inapplicable to most popular dynamic languages, which make heavy use of mutation and some form of inheritance (*prototypes* in JavaScript, *classes* in Python and Ruby). In this paper, we extend System D to System !D (pronounced "D-ref"), a calculus that supports mutation and prototype-based objects, and use it as a foundation for developing DJS, a dependent type system for JavaScript.

Imperative Updates. The presence of mutable variables makes it hard to retain the expressiveness of System D; as opposed to functional dictionary extension which produces new values, object extension mutates values, and a naïve treatment of reference types requires that they be invariant. To overcome this challenge (§ 2), we extend System D with *flow-sensitive heap types*, in the style of Alias Types [11], which allow the system to track *strong updates* through mutable variables. The formulation of flow-sensitive heap types in a dependent type system with higher-order functions is novel, and allows the system to express precise pre- and post-conditions of heaps as in separation logic.

Prototype-Based Objects. The semantics of JavaScript objects depends on *transitively* traversing prototype hierarchies among objects, which, in addition to mutation, takes reasoning about objects (unlike functional dictionaries) beyond the reach of decidable, first-order logics. Thus, we must track prototype hierarchies syntactically, but in many cases only finite portions of prototype hierarchies are known. To overcome this challenge (§ 3), we split heaps into "deep" parts, for which we have no information, and "shallow" parts, for which we track prototype links. We *unroll* prototype hierarchies in shallow heaps to precisely model the semantics of object operations, and we use new *uninterpreted heap symbols* to reason abstractly about deep heaps. In this way, we reduce the reasoning about objects to the underlying (decidable) refinement logic of functional dictionaries [20] and uninterpreted functions.

Expressiveness. To demonstrate the expressiveness of System !D (§ 4), we define Dependent JavaScript (DJS), a typed dialect of a large JavaScript subset that translates (à la λ_{JS} [15]) to System !D for type checking. We show how to type check examples from [6] ported to DJS, as well as the inheritance patterns found in *JavaScript: The Good Parts* [8].

Contributions. We define a type system for DJS that provides strong guarantees about well-typed programs: key lookups on objects always succeed, only functions are applied and are provided the correct number of arguments, the arguments to primitive functions have the correct types, and no other standard "stuck" states are reached. Although, as in JavaScript, we could return `undefined` for missing key lookups and perform implicit coercion between base types, we choose a more restrictive semantics for DJS because these features are straightforward and orthogonal to the current investigation, namely, imperative updates and prototypes. After a tour of our approach in the next three sections, we formally define the system in § 5 and § 6 and discuss related work in § 7. The translation from DJS to System !D can be found in § C of the appendix. We have a preliminary implementation, available at `http://cseweb.ucsd.edu/~rchugh/research/nested`. that type checks the examples in this paper.

2 Imperative Updates

In this section, we work through several DJS examples to motivate the extensions we make to System D to support imperative updates; additional changes required to support prototypes are the topic of § 3. For each DJS example, we show its translation in the new calculus, where it is type checked. We draw several examples directly from previous work [6] and port them to DJS to emphasize the new challenges.

2.1 Explicit References and Heap Types

As in most imperative languages, like Java and C#, all variables in JavaScript are mutable references, so every read (resp. write) implicitly performs a dereference (resp. reference update). We choose to support reference variables by extending a standard functional lambda-calculus with explicit references for two reasons. First, since System D is a type system for a lambda-calculus with dictionaries, we can use it directly as a starting point from which to add support for references. Second, once we have developed a suitable type system for this core language, we can type check programs in DJS by first translating them to this core language, following the approach of λ_{JS} [15]. In this way, we focus our effort on the interesting technical challenges of extending System D to an imperative setting, and avoid working with an unconventional presentation of JavaScript language semantics [19,17].

To retain the precision of dependent types in an imperative setting, we adopt a *flow-sensitive* treatment of reference types in the style of Alias Types [11], in which it is sound to *strongly update* the types of references that are guaranteed to point to exactly one heap cell.[1] With this approach, we distinguish between a conventional flow-insensitive typing environment Γ and a flow-sensitive *heap type* Σ, which records mappings from *abstract locations* to types.

Scalar Versus Reference Values. To warm up, consider the following example in DJS and its translation on the right.

```
1:  var k = "g";          |  let _k = ref "g" in
2:  var x = {"f" : k};    |  let _x = ref (ref {"f" : !_k}) in
```

Variable definitions in DJS are translated to bindings to a fresh references, and uses are translated to dereferences, such as `!_k`. The translation of a value depends on whether it is a *scalar value* (*e.g.* integer, boolean, string) or a *reference value* (*e.g.* object), a distinction made in languages like JavaScript, Java, and C# but not in pure-object languages like Python and Ruby. Scalar values like `"g"` are translated directly, while reference values like the object literal `{"f" : k}` are wrapped in reference cells. This distinction allows functions in the translation to mutate arguments that are reference values.

[1] We discuss the case when a reference may point to multiple objects in § 7.

In addition to deriving a (flow-insensitive) type for each expression, the type system also produces a (flow-sensitive) heap type. In the following, we write Σ_i to refer to the heap after checking line i of the example above.

$$\Sigma_1 \triangleq (\ell_k \mapsto dk\!:\!\{\nu \,|\, \nu = \text{“}\texttt{g}\text{”}\})$$
$$\Sigma_2 \triangleq \Sigma_1 \oplus (\ell \mapsto d\!:\!\{\nu \,|\, \nu = upd(empty, \text{“}\texttt{f}\text{”}, dk)\}) \oplus (\ell_x \mapsto Ref\,\ell)$$

The type system derives the type $\_\texttt{k} :: Ref\,\ell_k$, and the single *heap constraint* in Σ_1 records that the (fresh) heap location ℓ_k stores a value of type $\{\nu \,|\, \nu = \text{“}g\text{”}\}$. The binder dk refers to the value stored in the heap and is added to the (flow-insensitive) type environment. In general, whenever a value of type T is stored in the heap, we generate a fresh binder x and add $x\!:\!T$ to the typing environment so that subsequent typing derivations can refer to previous heap values.

After the second line, the type system derives the type $\_\texttt{x} :: Ref\,\ell_x$ and extends the heap using the $\oplus$ operator with two constraints: that a dictionary d equal to the object literal from line 2 is stored at (fresh) location ℓ, and that a value of type $Ref\,\ell$ is stored at (fresh) location ℓ_x. Notice that the former constraint refers to the binder dk from the previous heap constraint, and the latter constraint does not introduce a binder since it will not be needed in subsequent typing derivations.

2.2 Simple Objects

We now extend the example above to demonstrate how to translate operations on objects in DJS. For now, we assume that DJS objects are “simple” in that they are references to ordinary (functional) dictionaries; “full” objects backed by prototypes will be considered in § 3. Throughout the rest of the paper, we use the following notation to make refinement types more concise.

$$\{p\} \triangleq \{\nu \,|\, p\}$$

Key Lookup. The first operation we consider is key lookup.

```
3:  var xf = x.f;          |  let _xf = ref (get !!_x "f") in
```

The dictionary referred to by the DJS variable `x` is stored via two levels of indirection in the heap Σ_2, so it is dereferenced *twice* and then supplied to the following System D primitive that performs lookup on functional dictionaries.

$$\texttt{get} :: d_0\!:\!Dict \rightarrow k_0\!:\!\{Str(\nu) \wedge has(d_0, \nu)\} \rightarrow \{\nu = sel(d_0, k_0)\}$$

The expression `!!_x` corresponds to the dictionary d from Σ_2, which has an `"f"` binding, so the call to `get` type checks and produces a value of type $\{\nu = dk\}$, which in the current type environment is equivalent to $\{\nu = \text{“}\texttt{g}\text{”}\}$. Thus, for line 3 the type system derives $\_\texttt{xf} :: Ref\,\ell_{xf}$ and the heap $\Sigma_3 \triangleq \Sigma_2 \oplus (\ell_{xf} \mapsto \{\nu = \text{“}\texttt{g}\text{”}\})$.

Object Extension, Key Deletion, and Key Membership. Continuing with our example, the following demonstrates how the remaining operations on objects are translated.

```
 4:  x[k] = 3;              |  !_x := set !!_x !_k 3;
 5:  assert (x[k] = 3);     |  assert (get !!_x !_k = 3);
 6:  delete x.f;            |  !_x := del !!_x "f";
 7:  assert (!("f" in x)); |  assert (not (mem !!_x "f"));
```

The translation uses the corresponding functional primitives in System D.

$$\texttt{mem} :: d_0 : \mathit{Dict} \rightarrow k_0 : \mathit{Str} \rightarrow \{\mathit{Bool}(\nu) \wedge (\nu = \texttt{true} \Leftrightarrow \mathit{has}(d_0, k_0))\}$$
$$\texttt{set} :: d_0 : \mathit{Dict} \rightarrow k_0 : \mathit{Str} \rightarrow x_0 : \mathit{Top} \rightarrow \{\nu = \mathit{upd}(d_0, k_0, x_0)\}$$
$$\texttt{del} :: d_0 : \mathit{Dict} \rightarrow k_0 : \mathit{Str} \rightarrow \{\nu = \mathit{upd}(d_0, k_0, \mathit{bot})\}$$

On line 4, the DJS object is extended with a *dynamic key*, that is, an arbitrary program value and not just a string literal; this poses no problem, since System D [6] enables the specification of dynamic dictionaries. As opposed to a functional setting where a new dictionary would be created, the extension *mutates* the existing dictionary on the heap, for which we use the reference update operator `:=`. In particular, the new heap is the result of *strongly updating* Σ_3 with the following heap constraint for location ℓ, where the fresh binder d' describes the new dictionary as an extension of d, the previous value at the location.

$$\begin{aligned}\Sigma_4 &\triangleq \Sigma_3[\ell \mapsto d' : \{\nu = \mathit{upd}(d, dk, 3)\}] \\ &= \Sigma_3[\ell \mapsto d' : \{\nu = \mathit{upd}(\mathit{upd}(\mathit{empty}, \text{“}\texttt{f}\text{”}, \text{“}\texttt{g}\text{”}), \text{“}\texttt{g}\text{”}, 3)\}]\end{aligned}$$

The last equality follows by replacing d and dk with equivalent values given the current type environment. The system can, hence, prove that the call to `get` on line 5 is safe (because d' has a binding for “`g`”) and returns a value of type $\{\nu = 3\}$. Like object extension, the key deletion on line 6 results in a strong update, producing $\Sigma_6 \triangleq \Sigma_4[\ell \mapsto d'' : \{\nu = \mathit{upd}(d', \text{“}\texttt{f}\text{”}, \mathit{bot})\}]$. Thus, the system proves the assertion on line 7 that d'' does not have a binding for “`f`”.

Aliasing. Factoring references into flow-insensitive reference types and flow-sensitive heap types in the style of [11] makes it easy to track strong updates in the presence of aliasing.

```
 8:  var y = x;             |  let _y = ref !_x in
 9:  y.f = "hi";            |  !_y := set !!_y "f" "hi";
10:  assert (x.f == "hi"); |  assert (get !!_x "f" = "hi");
```

After the declaration on line 8, the heap is of the form below. Because `_x` $:: \mathit{Ref}\,\ell_x$, `_y` $:: \mathit{Ref}\,\ell_y$, and both ℓ_x and ℓ_y refer to the dictionary stored at ℓ, the update through `_y` on line 9 is reflected when reading through `_x` on line 10, and, hence, the assertion is proven.

$$\Sigma_8 \triangleq \cdots \oplus (\ell \mapsto d'' : \cdots) \oplus (\ell_x \mapsto \mathit{Ref}\,\ell) \oplus (\ell_y \mapsto \mathit{Ref}\,\ell)$$

2.3 Functions

In this section, we describe the translation of functions with examples ported from a functional setting [6] to the imperative setting.

Input and Output Worlds. Since the types of values are spread across the typing environment and a heap type, function types must now also describe constraints on the input and output heap. We write the arrow $x{:}T_1 \,/\, \Sigma_1 \to T_2 \,/\, \Sigma_2$ to describe functions that can be called with a value of type T_1 in calling contexts where the heap type *matches* Σ_1, and return a value of type T_2 with an updated heap Σ_2. We refer to type-heap pairs as *worlds*. Similar to the frame rule in separation logic, the heap types Σ_1 and Σ_2 need only describe the parts of the heap manipulated by the function. At a function call, the parts of the current heap not matched by Σ_1 remain unchanged after the call. The binder x may appear free in Σ_1, T_2, and Σ_2; we omit x when it does not. We also omit Σ_1 and Σ_2 when they are both the empty heap $\emptyset$.

Dependent Unions. The following highlights several interesting aspects of the translation.

```
1:  function negate(x) {          |  let _negate = ref (fun x ->
2:                                |    let _x = ref x in
3:    if (typeof(x) == "Int") {   |    if tagof !_x = "Int" then
4:      x = 0 - x;                |      _x := 0 - !_x
5:    } else {                    |    else
6:      x = !x;                   |      _x := not !_x
7:    }                           |    ;
8:    return x;                   |    !_x
9:  };                            |  )
```

The function definition in DJS creates a mutable variable, essentially the same as `var negate = function(x) { ... }`, so the translation starts by binding a new reference on line 1. The translation of the function body begins by creating a local reference on line 2 initialized to the formal, since `negate` may (and indeed, does) mutate the DJS variable `x`, and these updates do not affect the *value* supplied as an argument. The `typeof` operator translates directly to the corresponding System D primitive $\texttt{tagof} :: x{:}\mathit{Top} \to \{\nu = \mathit{tag}(x)\}$.

Not only are the input and output values of `negate` *untagged unions* that can be discriminated by run-time tag-tests, the tag of the output value *depends* on that of the input, a relationship captured by the following type in System D.

$$\texttt{negate} :: x{:}\{\mathit{tag}(\nu) = \text{“}\texttt{Int}\text{”} \vee \mathit{tag}(\nu) \text{ “}\texttt{Bool}\text{”}\} \to \{\mathit{tag}(\nu) = \mathit{tag}(x)\}$$

In the new system, we can verify the same type signature, where we omit the input and output heaps because they are empty. After line 1, the heap is $\Sigma_1 \triangleq (\ell_x \mapsto x{:}\{\mathit{Int}(\nu) \vee \mathit{Bool}(\nu)\})$, where the binder x is chosen since it refers to the value parameter. From the tag-test of `!_x`, which corresponds to

x, on line 3, the type system records the fact that $tag(x) =$ "`Int`" while checking line 4. Because the dereference on line 4 *also* corresponds to x, the type system verifies that the addition expression is safe, and that the assignment produces the new heap $\Sigma_4 \triangleq (\ell_x \mapsto x' : \{tag(x) =$ "`Int`" $\Rightarrow tag(\nu) =$ "`Int`"$\})$. Similar reasoning is performed for the else-branch, producing the heap $\Sigma_6 \triangleq (\ell_x \mapsto x' : \{tag(x) =$ "`Bool`" $\Rightarrow tag(\nu) =$ "`Bool`"$\})$. The type of the ℓ_x binding on line 7, corresponding to the entire if-expression, is the result of conjoining the two previous implications, and, hence, the dereference on line 8 has type $\{tag(\nu) = tag(x)\}$. The constraint for ℓ_x can be discarded by *heap subsumption*, so the output world of the function is $\{tag(\nu) = tag(x)\}/\emptyset$ as required.

Location Polymorphism. The previous function operates on *scalar* values; the following is a function that operates on *reference* values.

```
function getCount(t,c) {  |  let _getCount = ref (fun (t,c) ->
                          |    let (_t,_c) = (ref t, ref c) in
  if (c in t) {           |    if mem !!_t !_c then
    return toInt(t[c]);   |      !_toInt (get !!_t !_c)
  } else {                |    else
    return 0;             |      0
  }                       |
};                        |  )
```

All objects in DJS are passed to functions as references, so the translated object operations require two dereferences of the parameter. Since the reference is to a value on the heap, the arrow must explicitly name its location and type in the heap. We would like the function to work with references to *any* location, so we write the following arrow quantified by a *location parameter*.

$$\texttt{getCount} :: \forall L.\ (t : Ref\ L,\ c : Str)/(L \mapsto d : Dict) \rightarrow Int/(L \mapsto d' : \{\nu = d\})$$

Each caller must instantiate L appropriately to match the reference parameter that it passes in. With this signature, and a constant `toInt` $::$ $Top \rightarrow Int$, the type system can prove that the key lookup operation is safely guarded by the key membership test, that the output value is always an integer, and that the output heap is *exactly* the same as the input heap (*i.e.* unmodified), described by the constraint $(L \mapsto d' : \{\nu = d\})$.

Notation. We introduce two new forms of syntactic sugar to improve the clarity of function types in the sequel. First, we use the macro *same* as an output heap to refer 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. Second, since we will often quantify arrows by the locations of their object arguments, we use the following notation to abbreviate the above type.

$$\texttt{getCount} :: (t : Ref,\ c : Str)/(t \mapsto d : Dict) \rightarrow Int/same$$

The binding $t : Ref$ without a location introduces a new location variable L that is quantified by the arrow, and the overloaded use of t as a location in the heap refers to this variable L.

Effects. Neither of the previous functions has an observable effect on callers; although each function allocates local references, they are inaccessible from call sites. The following function *does* mutate the heaps of its callers.

```
function incCount(t,c) {  |  let _incCount = ref (fun (t,c) ->
                          |    let (_t,_c) = (ref t, ref c) in
  var i = getCount(t,c);  |    let _i = ref (!_getCount(!_t,!_c)) in
  t[c] = 1 + i;           |    !_t := set !!_t !_c (1 + !_i)
};                        |  )
```

The function *mutates* the object argument rather than creating a copy of the object and extending it, so the updated object type is reflected in the output heap.[2] In the function type below, the macro $EqMod(d', d, \{c\})$ encodes the fact that the dictionary d' is equal to d at all keys except c; additionally, the predicate $Int(sel(d', c))$ records the fact that d' has an integer c field.

$$\texttt{incCount} :: (t\!:\!Ref,\ c\!:\!Str)\,/\,(t \mapsto d\!:\!Dict) \\ \rightarrow\ Top\,/\,(t \mapsto d'\!:\!\{EqMod(\nu, d, \{c\}) \wedge Int(sel(\nu, c))\})$$

Higher-Order Functions and Parametric Polymorphism. The following example captures an idiom common to object-based languages, where a function is read from an object and then applied to the object itself.

```
function dispatch(x,f) {  |  let _dispatch = ref (fun (x,f) ->
                          |    let (_x,_f) = (ref x, ref f) in
  var fn = x[f];          |    let _fn = ref (get !!_x !_f) in
  return fn(x);           |    !_fn !_x
};                        |  )
```

The key innovation of nested refinements in System D allows the specification of arrow types within dynamic dictionaries, and this naturally carries over to the imperative setting. The following function type is quantified by *type parameters* in addition to a location parameter as in previous examples.

$$\texttt{dispatch} :: \forall A, B.\ (x\!:\!Ref,\ f\!:\!Str) \\ /\ (x \mapsto \{\nu :: A \wedge sel(\nu, f) :: Ref\, x\,/\,(x \mapsto A) \rightarrow B\,/\,same\}) \rightarrow B\,/\,same$$

3 Prototype-Based Objects

In the previous section, we showed how to extend System D to an imperative setting without sacrificing precision by factoring mutable references into flow-sensitive heap types. One simplification that we made, however, was to treat objects in JavaScript as simply references to dictionaries, when in fact they are also backed by *prototype* chains. In this section, we describe how we mirror the actual JavaScript semantics for objects in DJS, and show how to extend the type system with precise support for them, resulting in our final calculus, System !D.

[2] In fact, JavaScript does not provide a language primitive for deep copying an object, although user programs can approximate it.

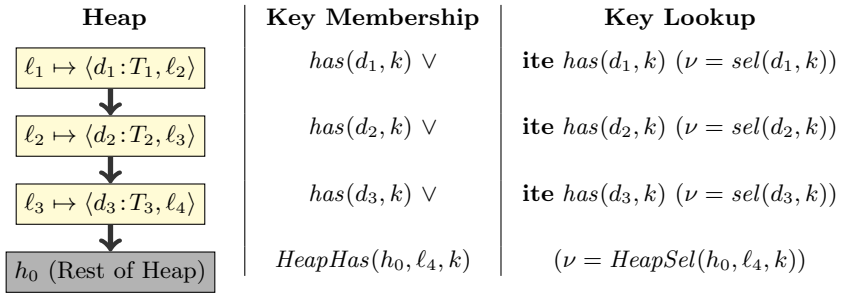

Fig. 1. Prototype Chain Unrolling

Prototype Semantics. JavaScript features a *prototype-based*, or *delegation-based*, inheritance model where every object is created with a link to some parent, or prototype, object. When trying to look up a key k that an object does not contain, its *prototype chain* is transitively traversed looking for k until some object along the chain contains k, or until the root of the object hierarchy is reached without finding k. Unfortunately, we cannot encode these object semantics on top of McCarthy's theory of arrays [20]; both the presence of mutation through the heap and the hierarchies induced by prototype chains take us beyond the reach of decidable, first-order reasoning.

Instead, we add new constants `objHas`, `objGet`, and `objSet` to encode the semantics of object operations, and we extend the type system in several ways. First, we augment heap constraints to track prototype links between locations. Second, since we cannot always reason about a prototype chain in its entirely, we introduce *heap variables* so we can reason abstractly about *entire* heaps. Finally, to reason about key membership and lookup, we *unroll* "shallow" parts of the heap (the locations for which we have constraints) to match the semantics of the operations and use *uninterpreted heap symbols* to reason about "deep" parts of the heap. In this way, we can precisely encode the semantics of imperative, prototype-based objects within a decidable first-order refinement logic.

We use a simple example to make these three extensions more concrete. The DJS function `beget`, which we will define in the sequel, creates a fresh object and sets its prototype link to the function argument.

```
var x3 = /* some obj */; var x2 = beget(x3); var x1 = beget(x2);
```

The types for the corresponding variables in the translation are $\_\mathtt{x}_1 :: \mathit{Ref}\,\ell_1$, $\_\mathtt{x}_2 :: \mathit{Ref}\,\ell_2$, and $\_\mathtt{x}_3 :: \mathit{Ref}\,\ell_3$, and the new heap type is shown in the first column of Figure 1. A heap binding may now be a *pair* where the second component is a prototype link. For example, the constraint $(\ell_1 \mapsto \langle d_1 : T_2, \ell_2 \rangle)$ records the fact that the prototype object of $\_\mathtt{x}_1$ is $\_\mathtt{x}_2$. Notice that the constraint for $\_\mathtt{x}_3$ specifies the location ℓ_4 of its prototype, even though that location is in the deep part of the heap named by the variable h_0, for which we have no precise information. To describe whether $\_\mathtt{x}_1$ has the key k, we unroll the shallow heap and obtain the formula in the second column of Figure 1. The unrolling bottoms

out with the uninterpreted *heap predicate* $\mathit{HeapHas}(h_0, \ell_4, k)$, which encodes the fact that the chain of objects in heap h_0 starting from location ℓ_4 has the key k. Similarly, to describe the result of reading key k from $\_\mathtt{x}_1$ we unroll the heap and obtain the conjunction of guarded implications in the third column, using the "if-then-else" macro **ite** $p\ q_1\ q_2 \triangleq (p \Rightarrow q_1) \wedge (\neg p \Rightarrow q_2)$. The uninterpreted *heap function* $\mathit{HeapSel}(h_0, \ell, k)$ represents the value produced by reading the key k from the chain of objects in heap h_0 starting at location ℓ_4.

Heap Polymorphism. In addition to type and location parameters as before, function types can be quantified by heap parameters and are now written $\forall \overline{A}, \overline{L}, \overline{h}.\ x{:}T_1 \,/\, \Sigma_1 \to T_2 \,/\, \Sigma_2$. Heap variables may be conjoined with heap constraints using the $\oplus$ operator. We introduce additional syntactic sugar to lighten the notation. First, we allow functions to be implicitly quantified by a single heap parameter that is threaded through the function signature.

$$\forall \overline{A}, \overline{L}.\ x{:}T_1 \,/\, \Sigma_1 \to T_2 \,/\, \Sigma_2 \triangleq \forall \overline{A}, \overline{L}, h.\ x{:}T_1 \,/\, h \oplus \Sigma_1 \to T_2 \,/\, h \oplus \Sigma_2$$

Second, to make the use of heap symbols compatible with this new notation, we use the token *cur* to name the implicitly quantified heap parameter. Finally, in addition to the sugar for implicit location polymorphism, the syntax $\dot{x}$ introduces an additional location parameter corresponding to the prototype of x. For example, we write $x{:}\mathit{Ref} \,/\, (x \mapsto \langle d{:}T, \dot{x}\rangle) \to T' \,/\, \mathit{same}$ as an abbreviation for $\forall L, L'.\ x{:}\mathit{Ref} \,/\, (L \mapsto \langle d{:}T, L'\rangle) \to T' \,/\, \mathit{same}$.

Key Membership. The DJS key membership test `"f" in x1` translates to `objHas(!_x1,"f")` in System !D.

$$\begin{aligned}\mathtt{objHas} :: (x{:}\mathit{Ref},\ k{:}\mathit{Str}) \ /\ (x \mapsto \langle d{:}\mathit{Dict}, \dot{x}\rangle)\\ \to \{\nu = \mathtt{true} \Leftrightarrow \mathit{ObjHas}(d, k, \mathit{cur}, \dot{x})\} \ /\ \mathit{same}\end{aligned}$$

The macro $\mathit{ObjHas}(d, k, \mathit{cur}, \dot{x})$ abbreviates $\mathit{has}(d, k) \vee \mathit{HeapHas}(\mathit{cur}, \dot{x}, k)$. The result of the call to `objHas`, after unrolling as prescribed above, is a value with the following type.

$$\{\nu = \mathtt{true} \Leftrightarrow (\mathit{has}(d_1, k) \vee \mathit{has}(d_2, k) \vee \mathit{has}(d_3, k) \vee \mathit{HeapHas}(h_0, \ell_4, k))\}$$

Like JavaScript, DJS also includes a "has-own" key membership test that looks only in the object itself. For this, System !D uses an `objHasOwn` primitive with a type just like for `objHas` except that the predicate in the return type is $\mathit{has}(d, k)$.

Key Lookup. The DJS key lookup `x1["f"]` translates to `objGet(!_x1,"f")` in System !D.

$$\begin{aligned}\mathtt{objGet} :: (x{:}\mathit{Ref},\ k{:}\mathit{Str}) \ /\ (x \mapsto \langle d{:}\{\mathit{ObjHas}(\nu, k, \mathit{cur}, \dot{x})\}, \dot{x}\rangle)\\ \to \{\nu = \mathit{ObjSel}(d, k, \mathit{cur}, \dot{x})\} \ /\ \mathit{same}\end{aligned}$$

In the return type, we use the macro $\nu = \mathit{ObjSel}(d, k, \mathit{cur}, \dot{x})$ to abbreviate **ite** $(\mathit{has}(d, k))\ (\nu = \mathit{sel}(d, k))\ (\nu = \mathit{HeapSel}(\mathit{cur}, \dot{x}, k))$. Because key lookup in DJS does not return `undefined` when the key is missing, the return type does not involve `undefined`. The call to `objGet` produces a value with the type from the heap unrolling in the third column of Figure 1.

Object Extension and Key Deletion. Since these operations do not affect the prototype chain, the corresponding System !D primitives are straightforward.

$$\begin{aligned}
\texttt{objSet} :: &(x\!:\!\mathit{Ref},\ k\!:\!\mathit{Str},\ y\!:\!\mathit{Top})\ /\ (x \mapsto \langle d\!:\!\mathit{Dict}, \dot{x}\rangle) \\
&\rightarrow \{\nu = y\}\ /\ (x \mapsto \langle d'\!:\!\{\nu = \mathit{upd}(d,k,y)\}, \dot{x}\rangle) \\
\texttt{objDel} :: &(x\!:\!\mathit{Ref},\ k\!:\!\mathit{Str})\ /\ (x \mapsto \langle d\!:\!\mathit{Dict}, \dot{x}\rangle) \\
&\rightarrow \mathit{Top}\ /\ (x \mapsto \langle d'\!:\!\{\nu = \mathit{upd}(d,k,\mathit{bot})\}, \dot{x}\rangle)
\end{aligned}$$

4 Inheritance

Now that we are equipped to precisely track imperative, extensible, prototype-based objects in System !D, we turn our attention to how to actually create prototype hierarchies in DJS. We work through several concrete examples in DJS that are inspired by the inheritance patterns that Crockford advocates in his book [8]. For brevity, many examples in this section summarize the corresponding ones in the book but retain the main patterns being showcased. For clarity, we omit all location and heap instantiations from the DJS code examples and their System !D translations; we discuss these annotations in § C.

4.1 Pattern: Prototypal

We first consider the "prototypal pattern," which is a minimal abstraction that wraps JavaScript's mechanism for setting up prototype chains. The following function takes an object `o` and creates a fresh object whose prototype is `o`.

```
function beget(o) {
  var F = function() { return this; };
  F.prototype = o;
  return new F(); };
```

We explain the semantics of this example in terms of its translation to System !D.

```
1:  let _beget = ref (fun o ->
2:    let _o = ref o in
3:    let _F = ref (ref { "code"      = (fun this -> this);
4:                        "prototype" = newobj _Object}) in
5:    objSet (!_F, "prototype", !_o);
6:    let ctor  = objGet (!_F, "code") in
7:    let proto = objGet (!_F, "prototype") in
8:    ctor (newobj proto)
9:  )
```

First, a local *constructor function* `_F` is defined to initialize fresh objects. The function is actually an object with two bindings, "`code`" and "`prototype`". The "`code`" field is privileged (not accessible to source programs) and stores the

constructor function value itself (line 3). Notice that the implicit `this` parameter of DJS functions are made explicit in System !D. On line 4, the "`prototype`" field of the constructor object is initialized to a fresh empty object literal; the prototype of every object literal is a pre-defined object called `_Object`. On line 5, the "`prototype`" field is immediately overwritten with `o`. The translation of the DJS expression `new F()` is spread across lines 6 through 8. First, the names `ctor` and `proto` are bound to the constructor object's "`code`" and "`prototype`" fields; because of the object update on line 5, `proto` is equal to `o`. Next, the System !D expression `newobj proto` creates a fresh object on the heap with prototype link set to the location of `proto`. Finally, `ctor` is called with the new object as an argument to complete its initialization. `ctor` is the function value on line 3 which simply returns its parameter, so the return value of the `beget` function (line 8) is an empty object with prototype `o`.

To type check `beget`, there are two function type signatures that we must provide. The first is for the constructor function `F`, which we annotate as follows, where $T_{emp} \triangleq \{\nu = empty\}$.

$$\texttt{F} :: [ctor]\ this : Ref\,/\,(this \mapsto \langle d{:}T_{emp}, \dot{this}\rangle) \rightarrow \{\nu = this\}\,/\,same$$

The DJS keyword $[ctor]$ before the function type indicates that the function is a constructor, for which the translation creates a function object (with "`code`" and "`prototype`" fields) as opposed to scalar functions that are translated directly to function values.[3] The type captures the fact that its argument is an empty object (as will be the case for all constructor functions) and that it returns the object without modification.

Second, we annotate `beget` with the following type. Notice that we use the macro *same* to record that a particular location in the output heap is unmodified, and that we omit binders from heap constraints when not needed.

$$\texttt{beget} :: \forall L.\ o{:}Ref\,/\,(o \mapsto \langle Dict, \dot{o}\rangle) \rightarrow Ref\,L\,/\,(L \mapsto \langle T_{emp}, o\rangle) \oplus (o \mapsto same)$$

The location parameters L, o, and $\dot{o}$ are used to type check the expression on line 8 (though we have elided them from the source code for clarity). In particular, L is supplied to the `newobj` expression to indicate where the fresh object should be allocated, and L and o are supplied as location parameters to the `ctor` function, which has the constructor type above. After substituting location parameters and supplying the reference to the fresh object, the type system verifies that the output of the call to `ctor` is the output world required by the `beget` specification.

Using The Pattern. To use the prototypal pattern, we start with an object.

```
var herb = {
  name : "Herb",
  get_name : function() { return "Hi, I'm " ^ this.name; } };
```

[3] Although we could treat *all* functions as objects, as in JavaScript, we treat them separately to emphasize the difference and because the lack of separation in JavaScript is often described as a poor design choice [8].

We specify the following type for the `get_name` "method"; the type makes clear that the function is not specific to the `herb` object at all, and can indeed be detached and used with any other object that satisfies the invariants on `this`.

$$this : Ref\,/\,(this \mapsto \langle d : \{Str(ObjSel(\nu, \text{“}\texttt{name}\text{”}, cur, \dot{this}))\}, \dot{this}\rangle) \rightarrow Str\,/\,same$$

We now call `beget` to create a new object that derives from `herb`. The type system proves that the new object has the `get_name` function (because its prototype does) and the result of calling it stores a string in `s`.

```
var henrietta  = beget(herb);
henrietta.name = "Henrietta";
var s          = henrietta.get_name();
```

Furthermore, when `herb.get_name` is updated, the type system strongly updates its type using the techniques in § 2, so it proves that the subsequent call to `get_name` through the child object produces an integer stored in `i`.

```
herb.get_name  = function() { return 42; };
var i          = henrietta.get_name();
```

4.2 Pattern: Pseudoclassical

The "pseudoclassical pattern" organizes constructors in a way that, on the surface, resembles traditional classes.

```
var Mammal = function(name) { this.name = name; return this; };
Mammal.prototype.get_name =
  function() { return "Hi, my name is " ^ this.name; };
```

We annotate the `Mammal` constructor to record that the function extends its object argument with its string argument.

$$\begin{aligned}\texttt{Mammal} :: [ctor]\ (this : Ref,\ name : Str)\,/\,(this \mapsto \langle d : T_{emp}, \dot{this}\rangle)\\ \rightarrow \{\nu = this\}\,/\,(this \mapsto \langle d' : \{\nu = upd(d, \text{“}\texttt{name}\text{”}, name)\}, \dot{this}\rangle)\end{aligned}$$

In the pseudoclassical pattern, the prototype object of the constructor is extended with "methods" that derived objects will share, resembling a "class table." The `get_name` function can be assigned *exactly* the same type as before.

Subclassing. Next, we define a "subclass" `Cat` by initializing its "`prototype`" field to a new "instance of `Mammal`"; this object will serve as the prototype of all "instances of `Cat`." Thus, as before, the type system proves that `henrietta` has a `get_name` function which produces a string stored in `s`.

```
var Cat = function(name) { this.name = name; return this; };
Cat.prototype = new Mammal("__dummy__");
var henrietta = new Cat("Henrietta");
var s         = henrietta.get_name();
```

4.3 Pattern: Functional

One shortcoming of the previous two patterns is that they provide no information hiding; all of an object's bindings can be accessed arbitrarily. To provide a measure of privacy, the "functional pattern" uses closures to keep object state private. In the following, a fresh object `x` is allocated internally but its private state (its "instance variables") is stored in a *separate* object `priv`.[4] The method `get_name` has access to `priv` and is installed on the object for public use, but the internals of `priv` cannot be accessed directly outside this initialization function.

```
1: function mammal(priv) {      | let _mammal = ref (fun priv ->
2:   var x = {};                |   let _x = ref (newobj _Object) in
3:   x.get_name = function() { |   objSet (!_x, "get_name",
4:     return priv.name;        |     fun this ->
5:   };                         |       objGet (!priv, "name"));
6:   return x;                  |   !_x
7: };                           | )
```

(In the translated version, we elide the usual prelude `let _priv = ref priv in` since it adds an unnecessary level of indirection in the types that follow.) To type check the above, we must provide annotations for two functions. The `get_name` function definition is different from that in previous examples, and now has the following type, where $T_{priv} \triangleq \{Str(ObjHas(\nu, \text{“}\texttt{name}\text{”}, cur, \dot{priv}))\}$.

$$\begin{aligned}\texttt{get\_name} :: U_{gn} \triangleq\ & Top\,/\,(priv \mapsto \langle d{:}T_{priv}, \dot{priv}\rangle) \\ & \to \{\nu = ObjSel(d, \text{“}\texttt{name}\text{”}, cur, \dot{priv})\}\,/\,same\end{aligned}$$

This arrow does not mention a *this* parameter because the function body does not refer to it. Furthermore, the function accesses the `priv` object from the *enclosing* environment; thus, the locations of `priv` and its prototype are *not* quantified in this type, nor is the heap referred to by *cur*. Instead, these variables are bound by the enclosing environment, namely, by the following type for `mammal`.

$$\begin{aligned}\texttt{mammal} :: \forall L.\ & priv : Ref\,/\,(priv \mapsto \langle d{:}T_{priv}, \dot{priv}\rangle) \\ & \to Ref\,L\,/\,(priv \mapsto same) \oplus (L \mapsto \langle T_{mam}, \hat{\ell}_{obj}\rangle)\end{aligned}$$

The object created by `mammal` has type $T_{mam} \triangleq \{sel(\nu, \text{“}\texttt{get\_name}\text{”}) :: U_{gn}\}$ and has prototype $\hat{\ell}_{obj}$, the location of the primordial object `_Object` that is the prototype of all object literals. The type for `mammal` is quantified by the location L at which to allocate the fresh object, the implicit location parameters *priv* and $\dot{priv}$, and an implicit heap parameter. The arrow type U_{gn} refers to the heap and location parameters quantified by the type of `mammal`.

[4] In [8], the state object is called `spec` and, for reasons beyond the scope of this discussion, the fresh object is called `that`.

Using The Pattern. We now use this initialization function to create an object. In the following, notice how the state for the new object is defined as the `herbPriv` object supplied to `mammal`. Because the return type of the arrow U_{gn} is exactly equal to the "`name`" field of its private state parameter, the type system proves that `oldName` stores a value of type $\{\nu = \text{“}\texttt{Herb}\text{”}\}$.

```
var herbPriv = {name: "Herb"};
var herb     = mammal(herbPriv);
var oldName  = herb.get_name();
```

In the following, the state object *is* mutated after it was used to construct `herb`. The type system precisely tracks the update, so it proves that `newName` stores the new value of type $\{\nu = \text{“}\texttt{Herbert}\text{”}\}$.

```
herbPriv.name = "Herbert";  var newName = herb.get_name();
```

Importantly, the private state of `herb` cannot be tampered with as long as the `herbPriv` object is not explicitly shared with other components.

Code Reuse. A second drawback of the pseudoclassical approach is that the "child" constructor duplicated some work of the "parent" constructor. Using closures, we can define a second function that re-uses the work done by the first.

```
var cat = function(priv) {
  var obj  = mammal(priv);
  obj.purr = function() { return "p-r-r-r-r"; };
  return obj; };
```

The type system verifies that the type of `cat` is identical to that of `mammal` except that T_{mam} is replaced with the following.

$$T_{cat} \triangleq \{(sel(\nu, \text{“}\texttt{purr}\text{”}) :: Top \to Str) \wedge (sel(\nu, \text{“}\texttt{get\_name}\text{”}) :: U_{gn})\}$$

4.4 Pattern: Parts

The final inheritance pattern described in [8] creates an object and installs methods from different sets of "parts." As discussed above, functions stored in objects are in no way tied to either the object or other functions stored within the object; function types in System !D allow functions to be arbitrarily read from and stored in objects. It is, therefore, straightforward to use several initialization functions on the same object.

In the following example, the functions `bark` and `purr` can both be assigned the type $this : Ref\,/\,(this \mapsto \langle Dict, \dot{this}\rangle) \to Str\,/\,same$, and the function `make_dog` (resp. `make_cat`) attaches `bark` (resp. `purr`) to any object passed in and bound to `x`. Since System !D reasons about strong updates through functional calls, it proves that the object `hybrid` is extended with both functions, that the subsequent calls are safe, and that the result stored in `noise` is a string.

```
var make_dog = function(x) { x.bark = function() { ... }; };
var make_cat = function(x) { x.purr = function() { ... }; };
var hybrid   = {};  make_dog(hybrid);  make_cat(hybrid);
var noise    = hybrid.bark() ^ hybrid.purr();
```

5 Syntax and Semantics

We now introduce the formal syntax of values, expressions, and types of System !D, defined in Figure 2.

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 like integers, booleans, and strings, the empty dictionary $\{\}$, and `null`. Logical values w are all values and applications of primitive function symbols F, such as addition $+$ and dictionary selection *sel*, to logical values. The System D constants `mem`, `get`, `set`, and `del` encode the semantics of key membership, key lookup, extension, and deletion on functional dictionaries. The constant `tag` introspects on the type tag of a value at run-time. For example,

$$\texttt{tag}(3) \triangleq \text{“}\texttt{Int}\text{”} \quad \texttt{tag}(\lambda x.\ e) \triangleq \text{“}\texttt{Fun}\text{”} \quad \texttt{tag}(\{\}) \triangleq \text{“}\texttt{Dict}\text{”} \quad \texttt{tag}(r) \triangleq \text{“}\texttt{Ref}\text{”}$$

Expressions. Expressions e include values, function application, if-expressions, and let-bindings. 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` $\ell\ v$ allocates create a fresh object at a 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, as long as v is a location. The constants `objHas` and `objGet` perform key membership and key lookup on objects, recursing over the prototype chain as needed. The constants `objHasOwn`, `objSet`, and `objDel` perform “has-own” key membership, object extension, and key deletion, operations that do not involve prototype chains. Label and break expressions are included to translate the control flow operations of DJS. The 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. Exceptions are arbitrary objects that can be raised and handled. The formal definition of the operational semantics, which can be found in § A, is standard [15].

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.

$$
\begin{array}{rrcl}
\textbf{Values} & v & ::= & x \mid c \mid v_1 \texttt{ ++ } v_2 \mapsto v_3 \mid \lambda x.\, e \mid r \\
\textbf{Expressions} & e & ::= & v \mid [\overline{T}; \overline{\ell}; \overline{\Sigma}]\ v_1\ v_2 \\
 & & \mid & \texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2 \mid \texttt{let } x = e_1 \texttt{ in } e_2 \\
 & & \mid & \texttt{ref } \ell\ v \mid\ !v \mid v_1 := v_2 \mid \texttt{newobj } \ell\ v \\
 & & \mid & @x : e \mid \texttt{break } @x\ e \\
 & & \mid & \texttt{throw } e \mid \texttt{try } e_1 \texttt{ catch } (x)\ e_2 \mid \texttt{try } e_1 \texttt{ finally } e_2 \\
\textbf{Types} & S, T & ::= & \{x \,|\, p\} \\
\textbf{Formulas} & p, q & ::= & P(\overline{w}) \mid w :: U \mid \mathit{HeapHas}(H, \ell, w) \mid p \wedge q \mid p \vee q \mid \neg p \\
\textbf{Logical Values} & w & ::= & v \mid F(\overline{w}) \mid \mathit{HeapSel}(H, \ell, w) \\
\textbf{Type Terms} & U & ::= & \forall[\overline{A}; \overline{L}; \overline{h}]\ x{:}T_1/\Sigma_1 \to T_2/\Sigma_2 \mid A \mid \mathit{Ref}\,\ell \\
\textbf{Heap Types} & \Sigma & ::= & H \mid C \mid \Sigma_1 \oplus \Sigma_2 \\
\textbf{Constraints} & C & ::= & \emptyset \mid (\ell \mapsto x{:}S) \mid (\ell \mapsto \langle x{:}S, \ell' \rangle) \mid C_1 \oplus C_2 \\
\textbf{Locations} & \ell & ::= & \hat{\ell} \mid L \\
\textbf{Heap Variables} & H & ::= & \emptyset \mid h \mid H_1 \oplus H_2
\end{array}
$$

Fig. 2. Syntax of System !D

Types and Formulas. All values in System !D are described by *refinement types* of the form $\{x \,|\, p\}$, where x may appear free in the formula p.[5] The language of *refinement formulas* includes predicates P, such as the equality predicate and dictionary predicate *has*, and the usual logical connectives. For example, the type of integers is $\{\mathit{tag}(\nu) = \text{“}\texttt{Int}\text{”}\}$, which we abbreviate to *Int*, and we write $\mathit{Int}(\nu)$ to abbreviate the refinement formula from the previous type. The type of dictionaries with a positive integer field “`f`” is $\{\mathit{tag}(\nu) = \text{“}\texttt{Dict}\text{”} \wedge \mathit{Int}(\mathit{sel}(\nu, \text{“}\texttt{f}\text{”})) \wedge \mathit{sel}(\nu, \text{“}\texttt{f}\text{”}) > 0\}$. 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.

Type Terms and Type Predicates. As in System D, we use a binary uninterpreted *type predicate* $w :: U$ in refinement formulas to describe values that have complex types, represented by *type terms* U. Type terms include arrows and type variables and reference types. Function types are parametrized by sequences of type, location, and heap variables. A reference type names a particular location in the heap, either a location variable L or a location constant $\hat{\ell}$.

Heap Types. *Heap types* comprise unordered sets of *heap variables* h and *heap constraints* C, combined with the $\oplus$ operator. We use the metavariable H for sets of heap variables, C for sets of heap constraints, and Σ for sets of either.

[5] The presentation in [6] 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; polymorphism is expressed with the type parameters of function types.

A heap can be thought of as a "deep" part for which we have no information (represented by heap variables) and a "shallow" part for which have precise information (represented by heap constraints). The heap constraint $(\ell \mapsto x\!:\!T)$ 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 constraints. The constraint $(\ell \mapsto \langle x\!:\!T, \ell' \rangle)$ additionally records a prototype link ℓ'.

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

6 Type Checking

In this section, we present the well-formedness, typing, and subtyping relations of System !D. The type system reuses the System D 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 and the use of uninterpreted heap symbols to regain precision in the presence of imperative, prototype-based objects.

Environments. Type and label environments are of the form

$$\Gamma ::= \emptyset \mid \Gamma, x\!:\!S \mid \Gamma, p \mid \Gamma, A \mid \Gamma, L \mid \Gamma, h \qquad \Omega ::= \emptyset \mid \Omega, @x\!:\!(T \,/\, \Sigma)$$

where a type environment binding records either: the derived type S for a variable x; a formula p to track the control flow along branches of an if-expression; or type, location, and heap variables introduced in the scope of a function; and a label environment binding records the world that the expression labeled $@x$ is expected to satisfy.

6.1 Well-formedness

The well-formedness relations are defined in Figure 6 of § B.

Types, Formulas, and Logical Values. We require that types be *well-formed* within the current type environment, which means that formulas used in types are boolean propositions and mention only variables that are currently in scope. Checking that values are well-formed is straightforward; the important point is that a variable x may be used only if it is bound in Γ.

Heap Types. To check the well-formedness of a heap Σ, we first use the heap equivalence operator $\equiv$ to rearrange it into the form $H \oplus C$. Each of the variables in H must be bound by the type environment. Locations bound by heap constraints must either be location variables bound by the type environment or constants, and there cannot be multiple constraints for a location. The types bound in heap constraints may refer to binders of other heap constraints. The values bound in a heap can, therefore, be regarded as a tuple of values with types that may depend on each other.

Arrows. Checking that the arrow $\forall[\overline{A};\overline{L};\overline{h}]\ x{:}T_1/\Sigma_1 \to T_2/\Sigma_2$ is well-formed proceeds in several steps. First, the polymorphic variables are added to the environment. Second, the input type T_1 is checked in the environment and the input heap Σ_1, so that it can refer to its locations and values. Next, the input heap is checked in the environment extended with the argument binder x. Finally, the procedure $\mathsf{Snapshot}(\Sigma_1)$ collects all binders from the input heap to be added to the environment, so that the output world $T_2\,/\,\Sigma_2$ can refer to them and, thus, express precise relationships with the input heap (*e.g.* `getCount` from § 2).

6.2 Value Typing

The value typing judgment $\Gamma;\ \Sigma;\ \Omega \vdash v :: T$, defined in Figure 7 of § B, 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. Each primitive constant c has a type, denoted by $ty(c)$, that is used by T-Const. Several examples from the library of constants are provided in § B. In addition to the constants from System D, System !D includes `objHas`, `objHasOwn`, `objGet`, `objSet`, and `objDel` for operations on objects. The rules T-Var and T-Extend are standard. The rule T-Fun uses an empty label environment to type check function bodies, so that break expressions cannot cross function boundaries.

6.3 Expression Typing

The expression typing judgment $\Gamma;\ \Sigma;\ \Omega \vdash v :: T\,/\,\Sigma'$ verifies that the evaluation of expression e produces a value of type T *and* a new heap type 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. A salient feature of our presentation is that several rules generate fresh variables. For each such rule, we require that the expression being checked be paired with an *explicit* let-binding, so that we can add the freshly generated variables to the typing environment for subsequent checking. An alternative approach is to use existential types in the style of [18] to refer to these fresh variables; we forgo this option so that our type language is more similar to System D.

Imperative Operations. The rules for heap-manipulating expressions are shown in Figure 3. To check the reference allocation `ref` $\ell\ v$, the rule T-Ref ensures that ℓ is not already bound in the heap, and then adds a heap constraint that

Expression Typing $\boxed{\Gamma;\ \Sigma;\ \Omega \vdash e :: S / \Sigma'}$

$$\frac{\begin{array}{c}\ell \notin dom(\Sigma) \quad \Gamma;\ \Sigma;\ \Omega \vdash v :: S_0 \\ \Sigma_1 = \Sigma \oplus (\ell \mapsto y{:}S_0) \quad y \text{ fresh} \quad S_1 = \{z \mid z :: Ref\,\ell\} \\ \Gamma, y{:}S_0, x{:}S_1;\ \Sigma_1;\ \Omega \vdash e :: S_2 / \Sigma_2 \quad \Gamma \vdash S_2 / \Sigma_2\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{let}\ x = \texttt{ref}\ \ell\ v\ \texttt{in}\ e :: S_2 / \Sigma_2}\ \text{[T-Ref]}$$

$$\frac{\Gamma;\ \Sigma;\ \Omega \vdash v :: \{z \mid z :: Ref\,\ell\} \quad \Sigma \equiv \Sigma_0 \oplus (\ell \mapsto x{:}S)}{\Gamma;\ \Sigma;\ \Omega \vdash\ !v :: \{y \mid y = x\} / \Sigma}\ \text{[T-Deref]}$$

$$\frac{\begin{array}{c}\Gamma;\ \Sigma;\ \Omega \vdash v_1 :: \{z \mid z :: Ref\,\ell\} \quad \Gamma;\ \Sigma;\ \Omega \vdash v_2 :: S' \\ \Sigma \equiv \Sigma_0 \oplus (\ell \mapsto y{:}S) \quad \Sigma_1 = \Sigma_0 \oplus (\ell \mapsto y'{:}S') \quad y' \text{ fresh} \\ \Gamma, y'{:}S', x{:}S';\ \Sigma_1;\ \Omega \vdash e :: S_2 / \Sigma_2 \quad \Gamma \vdash S_2 / \Sigma_2\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{let}\ x = v_1 := v_2\ \texttt{in}\ e :: S_2 / \Sigma_2}\ \text{[T-SetRef]}$$

$$\frac{\begin{array}{c}\ell_1 \notin dom(\Sigma) \quad \Gamma;\ \Sigma;\ \Omega \vdash v :: \{z \mid z :: Ref\,\ell_2\} \quad \Sigma \equiv \Sigma_0 \oplus (\ell_2 \mapsto \langle z{:}S, \ell_3\rangle) \\ y \text{ fresh} \quad T = \{d \mid d = empty\} \quad \Sigma_1 = \Sigma \oplus (\ell_1 \mapsto \langle y{:}T, \ell_2\rangle) \\ \Gamma, y{:}T, x{:}\{z \mid z :: Ref\,\ell_1\};\ \Sigma_1;\ \Omega \vdash e :: S_2 / \Sigma_2 \quad \Gamma \vdash S_2 / \Sigma_2\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{let}\ x = \texttt{newobj}\ \ell_1\ v\ \texttt{in}\ e :: S_2 / \Sigma_2}\ \text{[T-NewObj]}$$

Fig. 3. Reference expression type checking for System !D

stores the type S_0 of the initial value. The fresh heap binder y is added to the typing environment so that it can be mentioned while type checking the rest of the expression. The rule T-Deref checks that the given value is a reference to a location on the heap, and then produces a value whose type is exactly the heap value. The rule T-SetRef enables strong update of a heap location. The rule T-NewObj creates a fresh object on the heap at location ℓ_1 with prototype link set to the location ℓ_2 that v refers to.

Function Application. The T-App rule does quite a bit of heavy lifting to check the application $[\overline{T_a}; \overline{\ell_a}; \overline{\Sigma_a}]\ v_1\ v_2$. 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 $\Gamma \vdash [\overline{\ell_a}/\overline{L}]$ ensures that the list of location parameters contains no duplicates, a requirement to ensure the soundness of strong updates [11]. The substitution of parameters for polymorphic variables proceeds in three steps. First, the type variables $\overline{A}$ are instantiated with the type parameters $\overline{T_a}$ using the procedure Inst, where the only non-trivial case involves type predicates with type variables:

$$\mathsf{Inst}(w :: A, A, \{x \mid p\}) = p[w/x] \qquad \mathsf{Inst}(w :: B, A, T) = w :: B$$

Second, the location variables $\overline{L}$ are replaced with the parameters $\overline{\ell_a}$ by ordinary substitution. Third, the heap variables $\overline{h_a}$ are instantiated with the parameters $\overline{\Sigma}$ using a procedure HInst that substitutes heap constraints for heap variables. The substitution of heaps presents two complications. First, uninterpreted heap symbols *HeapHas* and *HeapSel* may refer to arbitrary heaps rather than just

Expression Typing (continued) $\boxed{\Gamma;\ \Sigma;\ \Omega \vdash e\ ::\ S/\Sigma'}$

$$\frac{\Gamma;\ \Sigma;\ \Omega \vdash v\ ::\ S}{\Gamma;\ \Sigma;\ \Omega \vdash v\ ::\ S/\Sigma}\ [\text{T-Val}]$$

$$\frac{\begin{array}{c}\Gamma;\ \Sigma;\ \Omega \vdash v\ ::\ \mathit{Bool} \\ \Gamma, v = \texttt{true};\ \Sigma;\ \Omega \vdash e_1\ ::\ S/\Sigma' \qquad \Gamma, v = \texttt{false};\ \Sigma;\ \Omega \vdash e_2\ ::\ S/\Sigma'\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ ::\ S/\Sigma'}\ [\text{T-If}]$$

$$\frac{\begin{array}{c}\Gamma;\ \Sigma;\ \Omega \vdash e_1\ ::\ S_1/\Sigma_1 \qquad \Gamma' = \Gamma, x{:}S_1, \mathsf{Snapshot}(\Sigma_1) \\ \Gamma';\ \Sigma_1;\ \Omega \vdash e_2\ ::\ S_2/\Sigma_2 \qquad \Gamma \vdash S_2/\Sigma_2\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ ::\ S_2/\Sigma_2}\ [\text{T-Let}]$$

$$\frac{\Gamma;\ \Sigma;\ \Omega \vdash e\ ::\ S_1/\Sigma_1 \qquad \Gamma \vdash (S_1/\Sigma_1) \sqsubseteq (S_2/\Sigma_2);\ \pi \qquad \Gamma \vdash S_2/\Sigma_2}{\Gamma;\ \Sigma;\ \Omega \vdash e\ ::\ S_2/\Sigma_2}\ [\text{T-Sub}]$$

$$\frac{\begin{array}{c}\Gamma \vdash \overline{T_a} \qquad \Gamma \vdash [\overline{\ell_a}/\overline{L}] \qquad \Gamma \vdash \overline{\Sigma_a} \\ \Gamma;\ \Sigma;\ \Omega \vdash v_1\ ::\ \{z \mid z :: \forall[\overline{A}; \overline{L}; \overline{h}]\ y{:}T_{11}/\Sigma_{11} \to T_{12}/\Sigma_{12}\} \\ (T'_{11}, \Sigma'_{11}, T'_{12}, \Sigma'_{12}) = \mathsf{Unroll}(\mathsf{HInst}(\mathsf{Inst}((T_{11}, \Sigma_{11}, T_{12}, \Sigma_{12}), \overline{A}, \overline{T_a})[\overline{\ell_a}/\overline{L}], \overline{h}, \overline{\Sigma_a})) \\ \Gamma \vdash \Sigma'_{11}[v_2/y] \qquad \Gamma \vdash \Sigma'_{12}[v_2/y] \qquad (T''_{12}/\Sigma''_{12}) = \mathsf{Freshen}(T'_{12}/\Sigma'_{12}) \\ \Gamma \vdash \Sigma \sqsubseteq \Sigma'_{11}[v_2/y];\ \pi \qquad \Gamma;\ \Sigma;\ \Omega \vdash v_2\ ::\ T'_{11} \qquad \pi' = \pi, [v_2/y] \\ \Gamma, \mathsf{Snapshot}(\pi'\Sigma''_{12}), x{:}\pi'T''_{12};\ \pi'\Sigma''_{12};\ \Omega \vdash e\ ::\ S_3/\Sigma_3 \qquad \Gamma \vdash S_3/\Sigma_3\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{let}\ x = [\overline{T_a}; \overline{\ell_a}; \overline{\Sigma_a}]\ v_1\ v_2\ \texttt{in}\ e\ ::\ S_3/\Sigma_3}\ [\text{T-App}]$$

Fig. 4. Expression type checking for System !D

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. As one last concern unique to our setting, the procedure Freshen generates fresh binders for the output heap type so that the output heap bindings at different call sites do not collide. These fresh variables are the reason why function application, like many of the reference operations, are type checked along with a let-binding.

From this point, the rest of the rule is fairly familiar. First, the current heap Σ is checked to be a heap subtype of the input heap of the arrow. If so, the substitution π maps binders from the input heap to the corresponding ones in the current heap Σ. Next, the value argument is checked against the domain type of the arrow. Finally, the substitution for heap and argument binders is applied, the heap bindings from the output heap are collected using Snapshot and added to the typing environment, and the rest of the expression is checked.

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 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 constraints, thus precisely matching the semantics of object key membership and lookup. We use the notation $\psi(p)$ to refer to a *formula context* ψ, a formula with a hole, filled with p. The functions $\mathsf{UnrollHas}$ and $\mathsf{UnrollSel}$ are each defined by two equations, one for matching inputs with a specific structure and one for the default case.

$$
\begin{aligned}
\mathsf{Unroll}(\mathit{HeapHas}(\Sigma, \ell, k)) &= \mathsf{UnrollHas}(H, C, \ell, k) \text{ where } \Sigma \equiv H \oplus C \\
\mathsf{Unroll}(\psi(\mathit{HeapSel}(\Sigma, \ell, k))) &= \mathsf{UnrollSel}(\psi, H, C, \ell, k) \text{ where } \Sigma \equiv H \oplus C \\
\mathsf{UnrollHas}(H, C \oplus (\ell \mapsto \langle d{:}S, \ell' \rangle), \ell, k) &= \mathit{has}(d, k) \vee \mathsf{UnrollHas}(H, C, \ell', k) \\
\mathsf{UnrollHas}(H, C, \ell, k) &= \mathit{HeapHas}(H, \ell, k) \\
\mathsf{UnrollSel}(\psi, H, C \oplus (\ell \mapsto \langle d{:}S, \ell' \rangle), \ell, k) &= \mathbf{ite}\ \mathit{has}(d, k)\ \psi(\mathit{sel}(d, k))\ \psi(\mathit{HeapSel}(H, \ell, k)) \\
\mathsf{UnrollSel}(\psi, H, C, \ell, k) &= \psi(\mathit{HeapSel}(H, \ell, k))
\end{aligned}
$$

Remaining Rules. The rules for if-expressions, (general) let-bindings, and subsumption (in Figure 4), and for control flow operations (in Figure 8 of § B) are straightforward.

6.4 Subtyping

Several relations comprise subtyping; the ones inherited from System D [6] are in Figure 9 of § B, while the new heap subtyping relations are in Figure 5.

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).

Heap Subtyping. The heap subtyping judgment $\Gamma \vdash \Sigma_1 \sqsubseteq \Sigma_2;\ \pi$ has a single rule that proceeds in three steps. First, it checks that the variables of each heap are identical. Next, it creates a substitution π from the binders of constraints in Σ_2 to the corresponding locations in Σ_1; the procedure Subst fails if there is no corresponding constraint for some location. Finally, it adds all of the (mutually-recursive) bindings of Σ_1 to the environment, and uses a helper judgment $\Gamma;\ \pi \vdash C_1 \sqsubseteq C_2$ to check that each heap constraint is discharged.

Constraint subsumption allows locations to be forgotten, but some care is required to ensure that this is sound. In particular, a discarded location ℓ should not be allocated again, since existing values of type $\mathit{Ref}\,\ell$ would inadvertently refer to the new cell. To prevent this, the typing judgments include as input a set $\mathcal{L}$ of "dead" locations that T-Ref may not allocate. Typing judgments also produce an output set $\mathcal{L}'$ of dead locations. We omit these from the formal definitions for clarity.

Heap Subtyping $\boxed{\Gamma \vdash \Sigma_1 \sqsubseteq \Sigma_2;\ \pi}$

$$\frac{\Sigma_1 \equiv H_1 \oplus C_1 \quad \Sigma_2 \equiv H_2 \oplus C_2 \quad H_1 \equiv H_2 \quad \pi = \mathsf{Subst}(C_1, C_2) \quad \Gamma, \mathsf{Snapshot}(C_1);\ \pi \vdash C_1 \sqsubseteq C_2}{\Gamma \vdash \Sigma_1 \sqsubseteq \Sigma_2;\ \pi}$$

Constraint Subtyping $\boxed{\Gamma;\ \pi \vdash C_1 \sqsubseteq C_2}$

$$\frac{}{\Gamma;\ \pi \vdash C \sqsubseteq \emptyset} \qquad \frac{\Gamma \Rightarrow \pi[\![x_2 : S_2]\!] \quad \Gamma;\ \pi \vdash C_1 \sqsubseteq C_2}{\Gamma;\ \pi \vdash (\ell \mapsto x_1 : S_1) \oplus C_1 \sqsubseteq (\ell \mapsto x_2 : S_2) \oplus C_2}$$

$$\frac{\Gamma \Rightarrow \pi[\![x_2 : S_2]\!] \quad \Gamma;\ \pi \vdash C_1 \sqsubseteq C_2}{\Gamma;\ \pi \vdash (\ell \mapsto \langle x_1 : S_1, \ell' \rangle) \oplus C_1 \sqsubseteq (\ell \mapsto \langle x_2 : S_2, \ell' \rangle) \oplus C_2}$$

World Subtyping $\boxed{\Gamma \vdash (S_1 / \Sigma_1) \sqsubseteq (S_2 / \Sigma_2);\ \pi}$

$$\frac{\Gamma \vdash \Sigma_1 \sqsubseteq \Sigma_2;\ \pi \quad \Gamma, \mathsf{Snapshot}(\Sigma_1) \vdash S_1 \sqsubseteq \pi S_2}{\Gamma \vdash (S_1 / \Sigma_1) \sqsubseteq (S_2 / \Sigma_2);\ \pi}$$

Fig. 5. Heap subtyping for System !D

7 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 [6] 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 elegant and powerful approach of *Alias Types* [11], 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* [23] 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 [23] 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 [10,3,9,2,23,24] allow strong updates to weak references under certain circumstances to support temporary invariant violations. We expect it to be straightforward to incorporate the *unfold/fold* approach of [23] to support weak references, and thus recursive object types, in DJS and System !D.

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 [5,13] 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 [25], 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 [27]. The notion of *recency types*, similar to Alias Types, was applied to JavaScript in [16], in which typing environments, in addition to heap types, are flow-sensitive. Prototype support in [16] 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 [21]. ADsafety [22] is a type system for ADsafe, a JavaScript sandbox, that restricts access to some fields, tracking strings precisely [14] 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 invariants about untagged unions.

JavaScript Semantics. We chose to start with the JavaScript "semantics-by-translation" of λ_{JS} [15] since it targets a conventional core language (lambda-calculus with explicit reference) that has been convenient for the current study. An alternate semantics [19] inherits unconventional aspects of the language specification [17] (*e.g.* "scope objects"), and so is more likely to be compatible with corner cases of JavaScript. The JavaScript standards committee [7] is working to mitigate some of the poor features (*e.g.* by introducing an optional "strict" mode) and to add some conventional features (*e.g.* modules). Thus, we believe

that advances in program analysis techniques for JavaScript, including type systems, have the potential to impact the evolution of the language.

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. V. Bono, V. Bono, K. Fisher, and K. Fisher. An imperative, first-order calculus with object extension. In *ECOOP*, 1998.
6. R. Chugh, P. M. Rondon, and R. Jhala. Nested refinements: A logic for duck typing. In *POPL*, 2012.
7. E. T.-. Committee. `http://www.ecmascript.org/community.php`.
8. D. Crockford. *JavaScript: The Good Parts*. Yahoo! Press, 2008.
9. M. Fahndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, pages 13–24. ACM, 2002.
10. J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI*, 2002.
11. F.Smith, D. Walker, and J. Morrisett. Alias types. In *ESOP*, 2000.
12. M. Furr, J. hoon (David) An, J. S. Foster, and M. W. Hicks. Static type inference for ruby. In *SAC*, pages 1859–1866, 2009.
13. P. D. Gianantonio, F. Honsell, and L. Liquori. A lambda calculus of objects with self-inflicted extension. In *OOPSLA*, 1998.
14. A. Guha, J. G. Politz, and S. Krishnamurthi. Fluid object types. 2011.
15. A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of javascript. In *ECOOP*, 2010.
16. P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In *ECOOP*, 2010.
17. E. International. *ECMAScript Language Specification, ECMA-262, 3rd ed.* 1999.
18. K. W. Knowles and C. Flanagan. Compositional reasoning and decidable checking for dependent contract types. In *PLPV*, pages 27–38, 2009.
19. S. Maffeis, J. Mitchell, and A. Taly. An operational semantics for JavaScript. In *APLAS*, 2008.
20. J. McCarthy. Towards a mathematical science of computation. In *IFIP*, 1962.
21. B. C. Pierce and D. N. Turner. Local type inference. In *POPL*, 1998.
22. J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. Adsafety: type-based verification of javascript sandboxing. In *USENIX Security*, 2011.
23. P. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL*, 2010.
24. J. Sunshine, K. Naden, S. Stork, J. Aldrich, and E. Tanter. First-class state change in plaid. In *OOPSLA*, 2011.
25. P. Thiemann. Towards a type system for analyzing javascript programs. In *ESOP*, 2005.
26. S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *ICFP*, 2010.
27. T. Zhao. Polymorphic type inference for scripting languages with object extensions. In *DLS*, 2011.

A Operational Semantics of System !D

The operational semantics for System !D is standard for a lambda-calculus with explicit references, following the presentation of λ_{JS} [15]. In the following, we show the single-step reduction rules, where run-time heaps σ map run-time locations r to values v. We highlight a few small differences compared to λ_{JS}. In System !D, lambdas and application take a single argument, so we use dictionaries to encode tuples of values. Also, function application in involves type, location, and heap parameters that are simply discarded during evaluation. Furthermore, in System !D we use a function δ that deals with all constants and primitive functions, including the object operations defined below. We refer the reader to λ_{JS} for the full definitions of evaluation, exception, and label contexts.

Operational Semantics $\boxed{\sigma;\ e \hookrightarrow e';\ \sigma'}$

$$\frac{r \text{ fresh} \quad r \in \mathsf{Names}(\ell)}{\sigma;\ \texttt{ref}\ \ell\ v \hookrightarrow r;\ \sigma \oplus r \mapsto v} \qquad \frac{r \in dom(\sigma)}{\sigma;\ !r \hookrightarrow \sigma(r);\ \sigma} \qquad \frac{r \in dom(\sigma)}{\sigma;\ r := v \hookrightarrow v;\ \sigma \oplus r \mapsto v}$$

$$\frac{r \text{ fresh} \quad r' \in dom(\sigma)}{\sigma;\ \texttt{newobj}\ r\ r' \hookrightarrow 0;\ \sigma \oplus r \mapsto \langle\{\}, r'\rangle}$$

$$\frac{\text{if } \delta(\sigma, c, v) \text{ is defined} \quad \delta(\sigma, c, v) = (v', \sigma')}{\sigma;\ c\ v \hookrightarrow v';\ \sigma'}$$

$$\sigma;\ \texttt{let}\ x = v\ \texttt{in}\ e \hookrightarrow e[v/x];\ \sigma \qquad \sigma;\ [\_;\_;\_]\ (\lambda x.\ e)\ v \hookrightarrow e[v/x];\ \sigma$$

$$\sigma;\ \texttt{if true then}\ e_1\ \texttt{else}\ e_2 \hookrightarrow e_1;\ \sigma \qquad \sigma;\ \texttt{if false then}\ e_1\ \texttt{else}\ e_2 \hookrightarrow e_2;\ \sigma$$

$$\frac{\sigma;\ e_1 \hookrightarrow e_1';\ \sigma'}{\sigma;\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \hookrightarrow \texttt{let}\ x = e_1'\ \texttt{in}\ e_2;\ \sigma'}$$

Primitive Functions (selected rules) $\boxed{\delta(\sigma, c, v) = (v', \sigma')}$

$$\begin{aligned}
\delta(\sigma, \texttt{get}\ (v\ \texttt{++}\ \text{“}\texttt{x}\text{”} \mapsto v_x), \text{“}\texttt{x}\text{”}) &= (v_x, \sigma) \\
\delta(\sigma, \texttt{get}\ (v\ \texttt{++}\ \text{“}\texttt{y}\text{”} \mapsto v_y), \text{“}\texttt{x}\text{”}) &= \delta(\sigma, \texttt{get}\ v, \text{“}\texttt{x}\text{”}) \\
\delta(\sigma, \texttt{mem}\ (v\ \texttt{++}\ \text{“}\texttt{y}\text{”} \mapsto v_y), \text{“}\texttt{x}\text{”}) &= \delta(\sigma, \texttt{mem}\ v, \text{“}\texttt{x}\text{”}) \\
\delta(\sigma, \texttt{mem}\ (v\ \texttt{++}\ \text{“}\texttt{x}\text{”} \mapsto v_x), \text{“}\texttt{x}\text{”}) &= (\texttt{true}, \sigma) \\
\delta(\sigma, \texttt{mem}\ \{\}, \text{“}\texttt{x}\text{”}) &= (\texttt{false}, \sigma) \\
\delta(\sigma, \texttt{set}\ d\ k, v) &= (d\ \texttt{++}\ k \mapsto v, \sigma)
\end{aligned}$$

$$\delta(\sigma, \texttt{objHasOwn}(r), v_2) =$$

$$\begin{cases} (\texttt{true}, \sigma) & \text{if } \sigma(r) = \langle v_1, r' \rangle \text{ and } v_2 \in \mathit{dom}(v_1) \\ (\texttt{false}, \sigma) & \text{else if } \sigma(r) = \langle v_1, r' \rangle \end{cases}$$

$$\delta(\sigma, \texttt{objHas}(r), v_2) =$$

$$\begin{cases} (\texttt{true}, \sigma) & \text{if } \sigma(r) = \langle v_1, r' \rangle \text{ and } v_2 \in \mathit{dom}(v_1) \\ (\texttt{false}, \sigma) & \text{else if } \sigma(r) = \langle v_1, \circ \rangle \text{ and } v_2 \notin \mathit{dom}(v_1) \\ \delta(\sigma, \texttt{objHas}(r'), v_2) & \text{else if } \sigma(r) = \langle v_1, r' \rangle \end{cases}$$

$$\delta(\sigma, \texttt{objGet}(r), v_2) =$$

$$\begin{cases} \delta(\sigma, \texttt{get}(v_1), v_2) & \text{if } \sigma(r) = \langle v_1, r' \rangle \text{ and } \delta(\sigma, \texttt{get}(v_1), v_2) \text{ is defined} \\ \delta(\sigma, \texttt{objGet}(r'), v_2) & \text{else if } \sigma(r) = \langle v_1, r' \rangle \end{cases}$$

$$\delta(\sigma, \texttt{objSet}(r)(v_2), v_3) =$$

$$\{ (v_3, \sigma \oplus r \mapsto \langle v_1 \texttt{ ++ } v_2 \mapsto v_3, r' \rangle) \text{ if } \sigma(r) = \langle v_1, r' \rangle \text{ and } v_1 \text{ is a dict}$$

B Type Checking (Additional Definitions)

B.1 System D Constants

Base Values (selected).

$$\begin{array}{rlrl} \texttt{1} & :: \{\nu = \texttt{1}\} & \texttt{true} & :: \{\nu = \texttt{true}\} \\ \text{“john”} & :: \{\nu = \text{“john”}\} & \texttt{false} & :: \{\nu = \texttt{false}\} \end{array}$$

Primitive Operations (selected).

$$\begin{array}{rl} \texttt{+} & :: x\!:\!Int \to y\!:\!Int \to \{Int(\nu) \wedge \nu = x + y\} \\ \texttt{not} & :: x\!:\!Bool \to \{Bool(\nu) \wedge x = \texttt{true} \Leftrightarrow \nu = \texttt{false}\} \\ \texttt{=} & :: x\!:\!Top \to y\!:\!Top \to \{Bool(\nu) \wedge \nu = \texttt{true} \Leftrightarrow x = y\} \\ \texttt{fix} & :: \forall A.\ (A \to A) \to A \\ \texttt{tagof} & :: x\!:\!Top \to \{\nu = tag(x)\} \end{array}$$

Dictionary Operations.

$$\begin{array}{rl} \{\} & :: \{\nu = empty\} \\ \texttt{mem} & :: d\!:\!Dict \to k\!:\!Str \to \{Bool(\nu) \wedge \nu = \texttt{true} \Leftrightarrow has(d,k)\} \\ \texttt{get} & :: d\!:\!Dict \to k\!:\!\{Str(\nu) \wedge\ has(d,\nu)\} \to \{\nu = sel(d,k)\} \\ \texttt{set} & :: d\!:\!Dict \to k\!:\!Str \to x\!:\!Top \to \{\nu = upd(d,k,x)\} \\ \texttt{del} & :: d\!:\!Dict \to k\!:\!Str \to \{\nu = upd(d,k,bot)\} \end{array}$$

Macros.

$$\begin{array}{rl} has(d,K) & \triangleq\ \forall_{k \in K}.\ sel(d,k) \neq bot \\ EqMod(d_1,d_2,K) & \triangleq\ \forall k'.\ (\wedge_{k \in K}\ k \neq k') \Rightarrow sel(d_1,k') = sel(d_2,k') \\ dom(d) = K & \triangleq\ (\wedge_{k \in K}\ has(d,k)) \wedge (\forall k'.\ (\wedge_{k \in K}\ k' \neq k) \Rightarrow \neg has(d,k)) \end{array}$$

B.2 New System !D Constants

Object Operations.

$$\begin{array}{rl} \texttt{objHasOwn} & :: \forall L_1, L_2, h.\ (x\!:\!Ref\, L_1,\ k\!:\!Str)\ /\ h \oplus (L_1 \mapsto \langle d\!:\!Top, L_2\rangle) \\ & \to \{Bool(\nu) \wedge (\nu = \texttt{true} \Leftrightarrow has(d,k))\}\ /\ same \\ \texttt{objHas} & :: \forall L_1, L_2, h.\ (x\!:\!Ref\, L_1,\ k\!:\!Str)\ /\ h \oplus (L_1 \mapsto \langle d\!:\!Top, L_2\rangle) \\ & \to \{Bool(\nu) \wedge (\nu = \texttt{true} \Leftrightarrow ObjHas(d,k,h,L_2))\}\ /\ same \\ \texttt{objGet} & :: \forall L_1, L_2, h.\ (x\!:\!Ref\, L_1,\ k\!:\!Str) \\ & \quad /\ h \oplus (L_1 \mapsto \langle d\!:\!\{ObjHas(\nu,k,h,L_2)\}, L_2\rangle) \\ & \to \{\nu = ObjSel(d,k,h,L_2)\}\ /\ same \\ \texttt{objSet} & :: \forall L_1, L_2, h.\ (x\!:\!Ref\, L_1,\ k\!:\!Str,\ y\!:\!Top)\ /\ h \oplus (L_1 \mapsto \langle d\!:\!Top, L_2\rangle) \\ & \to \{\nu = y\}\ /\ h \oplus (L_1 \mapsto \langle d'\!:\!\{\nu = upd(d,k,y)\}, L_2\rangle) \\ \texttt{objDel} & :: \forall L_1, L_2, h.\ (x\!:\!Ref\, L_1,\ k\!:\!Str,\ y\!:\!Top)\ /\ h \oplus (L_1 \mapsto \langle d\!:\!Top, L_2\rangle) \\ & \to\ Top\ /\ h \oplus (L_1 \mapsto \langle d'\!:\!\{\nu = upd(d,k,bot)\}, L_2\rangle) \end{array}$$

Macros.

$$
\begin{aligned}
\mathit{ObjHas}([d_1, \ldots, d_n], k, H, \ell) &\triangleq \mathit{has}(d_1, k) \vee \mathit{ObjHas}([d_2, \ldots, d_n], k, H, \ell) \\
\mathit{ObjHas}([], k, H, \ell) &\triangleq \mathit{HeapHas}(H, \ell, k) \\
\psi(\mathit{ObjSel}([d_1, \ldots, d_n], k, H, \ell)) &\triangleq \mathit{has}(d_1, k) \Rightarrow \psi(\mathit{sel}(d_1, k)) \wedge \\
&\qquad \neg \mathit{has}(d_1, k) \Rightarrow \psi(\mathit{ObjSel}([d_2, \ldots, d_n], k, H, \ell)) \\
\psi(\mathit{ObjSel}([], k, H, \ell)) &\triangleq \psi(\mathit{HeapSel}(H, \ell, k))
\end{aligned}
$$

B.3 Additional Figures

Well-Formed Types $\boxed{\Gamma;\ \Sigma \vdash S}$

$$\frac{\Gamma, x\!:\!\mathit{Top};\ \Sigma \vdash p}{\Gamma;\ \Sigma \vdash \{x \,|\, p\}}$$

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

$$\frac{\Gamma;\ \Sigma \vdash w \qquad \Gamma;\ \Sigma \vdash U}{\Gamma;\ \Sigma \vdash w :: U} \qquad \frac{\forall i.\ \Gamma;\ \Sigma \vdash w_i}{\Gamma;\ \Sigma \vdash P(\overline{w})} \qquad \frac{\Gamma;\ \Sigma \vdash p}{\Gamma;\ \Sigma \vdash \neg p}$$

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

$$\frac{\Gamma' = \Gamma, \overline{A}, \overline{L}, \overline{h} \qquad \Gamma';\ \Sigma_1 \vdash T_1 \qquad \Gamma', x\!:\!T_1 \vdash \Sigma_1 \qquad \Gamma', x\!:\!T_1, \mathsf{Snapshot}(\Sigma_1) \vdash T_2 / \Sigma_2}{\Gamma;\ \Sigma \vdash \forall[\overline{A}; \overline{L}; \overline{h}]\ x\!:\!T_1/\Sigma_1 \to T_2/\Sigma_2}$$

$$\frac{A \in \Gamma}{\Gamma;\ \Sigma \vdash A} \qquad \frac{\ell \in \mathit{dom}(\Sigma) \qquad \Gamma \vdash \ell}{\Gamma;\ \Sigma \vdash \mathit{Ref}\,\ell}$$

Well-Formed Heap Types $\boxed{\Gamma \vdash \Sigma}$

$$\frac{\Sigma \equiv h_1 \oplus \cdots \oplus h_n \oplus C \qquad \forall i.\ h_i \in \mathit{dom}(\Gamma) \qquad \Gamma;\ \Sigma \vdash C}{\Gamma \vdash \Sigma}$$

Well-Formed Heap Constraints $\boxed{\Gamma;\ \Sigma \vdash C}$

$$\frac{}{\Gamma;\ \Sigma \vdash \emptyset} \qquad \frac{\Gamma \vdash \ell \qquad \ell \notin \mathit{dom}(C) \qquad \Gamma;\ \Sigma \vdash S \qquad \Gamma;\ \Sigma \vdash C}{\Gamma;\ \Sigma \vdash (\ell \mapsto x\!:\!S) \oplus C} \qquad \frac{\Gamma \vdash \ell \qquad \Gamma \vdash \ell' \qquad \ell \notin \mathit{dom}(C) \qquad \Gamma;\ \Sigma \vdash S \qquad \Gamma;\ \Sigma \vdash C}{\Gamma;\ \Sigma \vdash (\ell \mapsto \langle x\!:\!S, \ell' \rangle) \oplus C}$$

Well-Formed Worlds $\boxed{\Gamma \vdash S / \Sigma}$

$$\frac{\Gamma \vdash \Sigma \qquad \Gamma;\ \Sigma \vdash S}{\Gamma \vdash S / \Sigma}$$

Fig. 6. Well-formedness for System !D

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

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

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

$$\frac{\begin{array}{c}U = \forall[\overline{A};\overline{L};\overline{h}]\ x{:}T_1/\Sigma_1 \to T_2/\Sigma_2 \qquad \Gamma;\ \emptyset \vdash U \qquad \Omega_1 = \emptyset \\ \Gamma_1 = \Gamma, \overline{A}, \overline{L}, \overline{h}, x{:}T_1, \mathsf{Snapshot}(\Sigma_1) \qquad \Gamma_1;\ \Sigma_1;\ \Omega_1 \vdash e :: T_2/\Sigma_2\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash \lambda x.\ e :: \{y \mid y = \lambda x.\ e \wedge y :: U\}}\ \text{[T-Fun]}$$

Fig. 7. Value type checking for System !D

Expression Typing (continued) $\boxed{\Gamma;\ \Sigma;\ \Omega \vdash e :: S/\Sigma'}$

$$\frac{\Gamma;\ \Sigma;\ \Omega, x{:}(S, \Sigma') \vdash e :: S/\Sigma'}{\Gamma;\ \Sigma;\ \Omega \vdash @x : e :: S/\Sigma'}\ \text{[T-Label]}$$

$$\frac{\Omega(x) = (S, \Sigma') \qquad \Gamma;\ \Sigma;\ \Omega \vdash e :: S/\Sigma'}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{break}\ @x\ e :: \bot/\bot}\ \text{[T-Break]}$$

$$\frac{\Gamma;\ \Sigma;\ \Omega \vdash e :: S/\Sigma'}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{throw}\ e :: \bot/\bot}\ \text{[T-Throw]}$$

$$\frac{\Gamma;\ \Sigma;\ \Omega \vdash e_1 :: S/\Sigma' \qquad \Gamma, x{:}\mathit{Top};\ \Sigma;\ \Omega \vdash e_2 :: S/\Sigma'}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{try}\ e_1\ \texttt{catch}\ (x)\ e_2 :: S/\Sigma'}\ \text{[T-TryCatch]}$$

$$\frac{\Gamma;\ \Sigma;\ \Omega \vdash e_1 :: S/\Sigma' \qquad \Gamma;\ \Sigma;\ \Omega \vdash e_2 :: S/\Sigma'}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{try}\ e_1\ \texttt{finally}\ e_2 :: S/\Sigma'}\ \text{[T-TryFinally]}$$

Fig. 8. Control expression checking for System !D

Subtyping $\boxed{\Gamma \vdash S_1 \sqsubseteq S_2}$

$$\frac{\Gamma, p_1 \Rightarrow p_2[x_1/x_2]}{\Gamma \vdash \{x_1 \,|\, p_1\} \sqsubseteq \{x_2 \,|\, p_2\}}$$

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

$$\text{[I-Valid]}\ \frac{\mathsf{Valid}(\llbracket \Gamma \rrbracket \Rightarrow p)}{\Gamma \Rightarrow p} \qquad \text{[I-Cnf]}\ \frac{\mathsf{CNF}(p) = \wedge_i\ (p_i \Rightarrow Q_i) \quad \forall i.\ \exists q \in Q_i.\ \Gamma, p_i \Rightarrow q}{\Gamma \Rightarrow p} \qquad \text{[I-ImpSyn]}\ \frac{\mathsf{Valid}(\llbracket \Gamma \rrbracket \Rightarrow w :: U') \quad \Gamma \vdash U' <: U}{\Gamma \Rightarrow w :: U}$$

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

$$\text{[U-Arrow]}\ \frac{\begin{array}{c}(T'_{11}, \Sigma'_{11}, T'_{12}, \Sigma'_{12}) = (T_{11}, \Sigma_{11}, T_{12}, \Sigma_{12})[\overline{A_2}/\overline{A_1}][\overline{L_2}/\overline{L_1}][\overline{h_2}/\overline{h_1}] \\ \Gamma \vdash \Sigma_{21} \sqsubseteq \Sigma'_{11};\ \pi \qquad \Gamma' = \Gamma, \mathsf{Snapshot}(\Sigma_{21}) \qquad \Gamma' \vdash T_{21} \sqsubseteq \pi T'_{11} \\ \pi' = \pi, [x_2/x_1] \qquad \Gamma', x_2 : T_{21} \vdash (\pi' T'_{12} / \pi' \Sigma'_{12}) \sqsubseteq (T_{22} / \Sigma_{22});\ \pi''\end{array}}{\Gamma \vdash \forall[\overline{A_1}; \overline{L_1}; \overline{h_1}]\ x_1 : T_{11}/\Sigma_{11} \to T_{12}/\Sigma_{12} <: \forall[\overline{A_2}; \overline{L_2}; \overline{h_2}]\ x_2 : T_{21}/\Sigma_{21} \to T_{22}/\Sigma_{22}}$$

$$\frac{}{\Gamma \vdash A <: A}\ \text{[U-Var]} \qquad \frac{}{\Gamma \vdash \mathit{Ref}\,\ell <: \mathit{Ref}\,\ell}\ \text{[U-Ref]}$$

Fig. 9. Subtyping for System !D

C From System !D to DJS

In this section, we present DJS with semantics defined by translation to System !D. The translation is borrowed from λ_{JS} [15], which presents a translation, or desugaring, of (untyped) JavaScript to a lambda calculus with references. They empirically demonstrate the soundness of desugaring by testing against JavaScript benchmarks. We focus our discussion on the slight differences in our translation as well as the presence of type annotations, unique to our setting.

We present the syntax of DJS along with a set of desugaring rules $[\![\ \mathsf{e}\ ]\!] = e$ that translate DJS expressions e to System !D expressions e. In addition to the type annotations described in the rest of this section, the syntax of DJS allows arbitrary type annotations as follows.

$$[\![\ /{*} : T {*}/\ \mathsf{e}\ ]\!] \ = \ [\![\mathsf{e}]\!]\ \mathtt{as}\ T \ \stackrel{\circ}{=}\ \mathtt{let}\ x :: T = [\![\mathsf{e}]\!]\ \mathtt{in}\ x$$

Variables and Sequencing. All DJS variables are translated to references, so variable uses translate to dereferences and assignments translate to reference updates. The translation uses the (fresh) location constant $\hat{\ell}_x$ for the reference cell corresponding to DJS variable x.

$$\begin{aligned}
[\![\ \mathsf{var\ x = e;\ e'}\ ]\!] &= \mathtt{let}\ _x = \mathtt{ref}\ \hat{\ell}_x\ [\![\mathsf{e}]\!]\ \mathtt{in}\ [\![\mathsf{e'}]\!] \\
[\![\ \mathsf{e;\ e'}\ ]\!] &= \mathtt{let}\ _ = \mathtt{ref}\ \hat{\ell}__\ [\![\mathsf{e}]\!]\ \mathtt{in}\ [\![\mathsf{e'}]\!] \\
[\![\ \mathsf{x}\ ]\!] &= !_x \\
[\![\ \mathsf{e_1 = e_2}\ ]\!] &= [\![\mathsf{e_1}]\!] := [\![\mathsf{e_2}]\!]
\end{aligned}$$

Simple Values and Operators. The following are a few examples of how constants translate to System !D. Although we could model the implicit coercion semantics of JavaScript by changing the semantics and types of System D constants like `and`, `not`, *etc.*, we leave them unaltered because implicit coercions are not central to the current investigation.

$$[\![\ 12\ ]\!] = 12 \qquad [\![\ \mathsf{typeof\ e}\ ]\!] = \mathtt{tag}\ [\![\mathsf{e}]\!] \qquad [\![\ \mathsf{e_1 == e_2}\ ]\!] = [\![\mathsf{e_1}]\!]\ \mathtt{=}\ [\![\mathsf{e_2}]\!]$$

Objects. Objects are translated to System !D as pairs $\langle \ell, \ell' \rangle$ of locations so that prototype links are syntactically explicit. In contrast, in λ_{JS} prototype links are stored in a distinguished "`__proto__`" field. We syntactically track prototype links in the type system to facilitate our unrolling of heap predicates, which would be more complicated if prototypes were buried arbitrarily within refinement types.

$$\begin{aligned}
[\![\ /{*} : \ell\, {*}/\ \{\ \overline{\mathsf{e_1}} : \overline{\mathsf{e_2}}\ \}\ ]\!] = \ & \mathtt{let}\ \mathit{obj} = \mathtt{newobj}\ \ell\ _\mathit{Object}\ \mathtt{in} \\
& \mathtt{let}\ _ = [;\ell, \hat{\ell}_{obj};]\ \mathtt{objSet}\ (\mathit{obj},\ [\![\mathsf{e_{11}}]\!],\ [\![\mathsf{e_{21}}]\!])\ \mathtt{in}\ \cdots \\
& \mathtt{let}\ _ = [;\ell, \hat{\ell}_{obj};]\ \mathtt{objSet}\ (\mathit{obj},\ [\![\mathsf{e_{1n}}]\!],\ [\![\mathsf{e_{2n}}]\!])\ \mathtt{in}\ \mathit{obj}
\end{aligned}$$

Object operations translate to new System !D constants as follows. Notice that the abstract syntax of DJS includes the locations for objects and their prototypes; our implementation infers some of these annotations where possible. DJS

uses a hasOwn operator to test whether an object itself contains the given key, while in full JavaScript the corresponding operation "`hasOwnProperty`" is an ordinary method. Furthermore, the semantics of `objGet` are stricter than key lookup in JavaScript, which returns `undefined` when a given key is not found.

$$[\![\ /*:\ell\ \ell'*/\ \mathsf{e}_1[\mathsf{e}_2]\ ]\!] = [;\ell,\ell';]\ \texttt{objGet}\ ([\![\mathsf{e}_1]\!],\ [\![\mathsf{e}_2]\!])$$
$$[\![\ /*:\ell\ \ell'*/\ \mathsf{e}_1[\mathsf{e}_2] = \mathsf{e}_3\ ]\!] = [;\ell,\ell';]\ \texttt{objSet}\ ([\![\mathsf{e}_1]\!],\ [\![\mathsf{e}_2]\!],\ [\![\mathsf{e}_3]\!])$$
$$[\![\ /*:\ell\ \ell'*/\ \mathsf{delete}\ \mathsf{e}_1[\mathsf{e}_2]\ ]\!] = [;\ell,\ell';]\ \texttt{objDel}\ ([\![\mathsf{e}_1]\!],\ [\![\mathsf{e}_2]\!])$$
$$[\![\ /*:\ell\ \ell'*/\ \mathsf{e}_2\ \mathsf{in}\ \mathsf{e}_1\ ]\!] = [;\ell,\ell';]\ \texttt{objHas}\ ([\![\mathsf{e}_1]\!],\ [\![\mathsf{e}_2]\!])$$
$$[\![\ /*:\ell\ \ell'*/\ \mathsf{e}_2\ \mathsf{hasOwn}\ \mathsf{e}_1\ ]\!] = [;\ell,\ell';]\ \texttt{objHasOwn}\ ([\![\mathsf{e}_1]\!],\ [\![\mathsf{e}_2]\!])$$

Simple Functions. In JavaScript all functions may be used as constructors, so λ_{JS} translates all functions to objects with "`code`" and "`prototype`" fields. In DJS, we distinguish between "simple" and "constructor" functions to emphasize the difference, although there is no technical reason not to treat all functions uniformly. Notice that each simple function has an explicit *this* parameter in the translation, and an *args* argument that stores all of the parameters in a single tuple (dictionary). Also, the translated function body is wrapped with a $@ret$ label to help translate return expressions.

$$[\![\ \mathsf{function}(\overline{\mathsf{x}})\ /*:T*/\ \{\ \mathsf{e}\ \}\ ]\!] =$$
$$(\lambda(\mathit{this},\mathit{args}).\ \texttt{let}\ (\_x_0,\ \ldots) = (\texttt{ref}\ \hat{\ell}_{x_0}\ (\texttt{get}\ \mathit{args}\ \text{“0”}),\ \ldots)\ \texttt{in}\ @\mathit{ret} : [\![\mathsf{e}]\!])\ \texttt{as}\ [\![T]\!]$$
$$[\![\ \mathsf{this}\ ]\!] = \mathit{this} \qquad [\![\ \mathsf{return}\ \mathsf{e}\ ]\!] = \texttt{break}\ @\mathit{ret}\ [\![\mathsf{e}]\!]$$

When a function is invoked with the syntax $\mathsf{e}_1(\mathsf{e}_2)$ — as opposed to $\mathsf{e}_0[\mathsf{e}_1](\mathsf{e}_2)$ — `null` is supplied as the *this* parameter. In contrast, JavaScript supplies the global object. Although we could adopt the same approach in DJS, we do not because it is widely considered one of the worst features of JavaScript [8].

$$[\![\ /*:[\overline{T};\overline{\ell};\overline{\Sigma}]*/\ \mathsf{e}(\mathsf{e}_1,\ \ldots,\ \mathsf{e}_n)\ ]\!] = [\overline{T};\overline{\ell};\overline{\Sigma}]\ [\![\mathsf{e}]\!]\ (\texttt{null},\ ([\![\mathsf{e}_1]\!],\ \ldots,\ [\![\mathsf{e}_n]\!]))$$
$$\begin{aligned}[\![\ /*:[\overline{T};\overline{\ell};\overline{\Sigma}]\ \ell\ \ell'*/\ \mathsf{e}[\mathsf{e}'](\mathsf{e}_1,\ \ldots,\ \mathsf{e}_n)\ ]\!] = {}& \texttt{let}\ \mathit{obj} = [\![\mathsf{e}]\!]\ \texttt{in}\\ & \texttt{let}\ \mathit{func} = [;\ell,\ell';]\ \texttt{objGet}\ (\mathit{obj},\ [\![\mathsf{e}']\!])\ \texttt{in}\\ & [\overline{T};\overline{\ell};\overline{\Sigma}]\ \mathit{func}\ (\mathit{obj},\ ([\![\mathsf{e}_1]\!],\ \ldots,\ [\![\mathsf{e}_n]\!]))\end{aligned}$$

Constructor Functions. The translation of constructor functions and object construction follow that of λ_{JS}. The DJS keyword $[ctor]$ signifies that the function is a constructor, not a simple, function.

$$\begin{aligned}&[\![\ \mathsf{function}\ \mathsf{F}(\overline{\mathsf{x}})\ /*:[ctor]\ T*/\ \{\ \mathsf{e}\ \}\ ]\!] =\\ &\quad \texttt{let}\ \mathit{ctor} = \lambda(\mathit{this},\mathit{args}).\ \texttt{let}\ (\_x_0,\ \ldots) = (\texttt{ref}\ \hat{\ell}_{x_0}\ (\texttt{get}\ \mathit{args}\ \text{“0”}),\ \ldots)\ \texttt{in}\ @\mathit{ret} : [\![\mathsf{e}]\!]\ \texttt{in}\\ &\quad \texttt{let}\ \mathit{proto} = \texttt{newobj}\ \hat{\ell}_{\mathit{Fproto}}\ \mathit{\_Object}\ \texttt{in}\\ &\quad \texttt{ref}\ \hat{\ell}_F\ \{\ \text{“code”} = \mathit{ctor}\ \texttt{as}\ [\![T]\!];\ \text{“prototype”} = \mathit{proto}\ \}\end{aligned}$$

$$\begin{aligned}&[\![\ \mathsf{new}\ /*:[\overline{T};\overline{\ell};\overline{\Sigma}]\ \ell*/\ \mathsf{e}(\mathsf{e}_1,\ \ldots,\ \mathsf{e}_n)\ ]\!] =\\ &\quad \texttt{let}\ \mathit{foo} = \texttt{get}\ !{[\![\mathsf{e}]\!]}\ \texttt{in}\\ &\quad \texttt{let}\ \mathit{obj} = \texttt{newobj}\ \ell\ (\mathit{foo}\ \text{“prototype”})\ \texttt{in}\\ &\quad [\overline{T};\overline{\ell};\overline{\Sigma}]\ (\mathit{foo}\ \text{“code”})\ (\mathit{obj},\ ([\![\mathsf{e}_1]\!],\ \ldots,\ [\![\mathsf{e}_n]\!]))\end{aligned}$$

Control Flow Operators.

$$
\begin{aligned}
[\![\ \mathsf{if}\ (\mathsf{e}_1)\ \{\ \mathsf{e}_2\ \}\ \mathsf{else}\ \{\ \mathsf{e}_3\ \}\ ]\!] &= \texttt{if}\ [\![\mathsf{e}_1]\!]\ \texttt{then}\ [\![\mathsf{e}_2]\!]\ \texttt{else}\ [\![\mathsf{e}_3]\!] \\
[\![\ \mathsf{try}\ \{\ \mathsf{e}_1\ \}\ \mathsf{catch}\ (\mathsf{x})\ \{\ \mathsf{e}_2\ \}\ ]\!] &= \texttt{try}\ [\![\mathsf{e}_1]\!]\ \texttt{catch}\ (x)\ [\![\mathsf{e}_2]\!] \\
[\![\ \mathsf{try}\ \{\ \mathsf{e}_1\ \}\ \mathsf{finally}\ \{\ \mathsf{e}_2\ \}\ ]\!] &= \texttt{try}\ [\![\mathsf{e}_1]\!]\ \texttt{finally}\ [\![\mathsf{e}_2]\!] \\
[\![\ \mathsf{break}\ ]\!] &= \texttt{break}\ @\mathit{break}\ \texttt{undefined} \\
[\![\ \mathsf{continue}\ ]\!] &= \texttt{break}\ @\mathit{continue}\ \texttt{undefined}
\end{aligned}
$$

We define *frame annotations* F to be just like arrows except that the types are both trivial. Frames are useful for annotating iteration constructs, which neither consume nor produce values.

$$F ::= \forall[\overline{A}; \overline{L}; \overline{h}]\ \_ : \mathit{Top}/\Sigma_1 \rightarrow \mathit{Top}/\Sigma_2$$

In the following, frame annotations F' represent loop invariants.

$$
\begin{aligned}
&[\![\ /* : F */\ \mathsf{while}\ (\mathsf{e}_{\mathsf{cond}})\ \{\ /* : F' */\ \mathsf{e}_{\mathsf{body}}\ \}\ ]\!] = \\
&\quad @\mathit{break} : \\
&\qquad \texttt{let rec}\ \mathit{loop} :: F = \lambda\_. \\
&\qquad\quad \texttt{if}\ [\![\mathsf{e}_{\mathsf{cond}}]\!] \\
&\qquad\quad \texttt{then}\ (@\mathit{continue} : [\![\mathsf{e}_{\mathsf{body}}]\!])\ \texttt{as}\ F'; \mathit{loop}\ () \\
&\qquad\quad \texttt{else undefined} \\
&\qquad \texttt{in}\ \mathit{loop}\ ()
\end{aligned}
$$

$$
\begin{aligned}
&[\![\ /* : F */\ \mathsf{do}\ \{\ /* : F' */\ \mathsf{e}_{\mathsf{body}}\ \}\ \mathsf{while}\ (\ \mathsf{e}_{\mathsf{cond}}\ )\ ]\!] = \\
&\quad @\mathit{break} : \\
&\qquad \texttt{let rec}\ \mathit{loop} :: F = \lambda\_. \\
&\qquad\quad (@\mathit{continue} : [\![\mathsf{e}_{\mathsf{body}}]\!])\ \texttt{as}\ F'; \\
&\qquad\quad \texttt{if}\ [\![\mathsf{e}_{\mathsf{cond}}]\!] \\
&\qquad\quad \texttt{then}\ \mathit{loop}\ () \\
&\qquad\quad \texttt{else undefined} \\
&\qquad \texttt{in}\ \mathit{loop}\ ()
\end{aligned}
$$

$$
\begin{aligned}
&[\![\ /* : F */\ \mathsf{for}\ (\mathsf{e}_{\mathsf{init}};\ \mathsf{e}_{\mathsf{cond}};\ \mathsf{e}_{\mathsf{incr}})\ \{\ /* : F' */\ \mathsf{e}_{\mathsf{body}}\ \}\ ]\!] = \\
&\quad \texttt{let}\ \_ = [\![\mathsf{e}_{\mathsf{init}}]\!]\ \texttt{in} \\
&\quad @\mathit{break} : \\
&\qquad \texttt{let rec}\ \mathit{loop} :: F = \lambda\_. \\
&\qquad\quad \texttt{if}\ [\![\mathsf{e}_{\mathsf{cond}}]\!] \\
&\qquad\quad \texttt{then}\ (@\mathit{continue} : [\![\mathsf{e}_{\mathsf{body}}]\!])\ \texttt{as}\ F'; [\![\mathsf{e}_{\mathsf{incr}}]\!]; \mathit{loop}\ () \\
&\qquad\quad \texttt{else undefined} \\
&\qquad \texttt{in}\ \mathit{loop}\ ()
\end{aligned}
$$