Status Report: Dependent Types for JavaScript

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

David Herman
Mozilla Research
dherman@mozilla.com

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

We are developing Dependent JavaScript (DJS), an explicitly-typed dialect of a large JavaScript subset that features mutable, prototype-based objects and arrays as well as precise control-flow reasoning. We *desugar* DJS programs to a core lambda-calculus with explicit references following in the style of λ_{JS} [3]. Our new type system operates on desugared programs, building upon techniques from System D [2] and Alias Types [4]. With our preliminary implementation, we demonstrate that DJS is expressive enough to reason about a variety of tricky idioms found in small examples drawn from several sources, including the popular book *JavaScript: The Good Parts* and the SunSpider benchmark suite. In this report, we provide a brief overview of DJS; more details can be found in [1].

Path Sensitivity. Consider the following function where the type annotation says that if the input is a number, then so is the return value, and otherwise it's a boolean; the "if-then-else" macro $\mathbf{ite}\ p\ q_1\ q_2$ abbreviates the formula $(p \Rightarrow q_1) \wedge (\neg p \Rightarrow q_2)$.

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

When checking the true case of the conditional, DJS tracks that `x` is a number. Because the subtraction also produces a number, it concludes that the return value has type $\{\nu \,|\, Num(x) \Rightarrow Num(\nu)\}$. In the false case, `x` is an arbitrary, non-number value, which is safe to use because the JavaScript negate operator inverts the "truthiness" of *any* value, not just booleans. So, the return value has type $\{\nu \,|\, \neg Num(x) \Rightarrow Bool(\nu)\}$. By combining the types of values stored in `x` along both branches, DJS verifies that the return type satisfies its specification.

Refinement Types. Even the simple example above requires sophisticated propositional and equational reasoning that depends on program *values*. In DJS, we employ *refinement types* to encode these relationships and use an SMT solver to discharge logical validity queries that arise during (sub)type checking. Refinement types are quite expressive but, by using formulas drawn from a *decidable* logic, once the programmer has provided annotations on functions, type checking proceeds automatically. In comparison, more expressive *dependent* type systems like Coq rely on the programmer or heuristics to interactively discharge proof obligations.

Primitive Operators. In DJS, we use refinements to assign precise types to primitive operators; we show a few below.

$$\begin{aligned}
\texttt{typeof} &:: x{:}\mathit{Top} \rightarrow \{\nu \,|\, \nu = \mathit{tag}(x)\} \\
\texttt{!} &:: x{:}\mathit{Top} \rightarrow \{\nu \,|\, \mathbf{ite}\ \mathit{falsy}(x)\ (\nu = \texttt{true})\ (\nu = \texttt{false})\} \\
\texttt{\&\&} &:: x{:}\mathit{Top} \rightarrow y{:}\mathit{Top} \rightarrow \{\nu \,|\, \mathbf{ite}\ \mathit{falsy}(x)\ (\nu = x)\ (\nu = y)\} \\
\texttt{||} &:: x{:}\mathit{Top} \rightarrow y{:}\mathit{Top} \rightarrow \{\nu \,|\, \mathbf{ite}\ \mathit{falsy}(x)\ (\nu = y)\ (\nu = x)\}
\end{aligned}$$

The above types allow DJS to reason about `negate` and idioms like `if (x && x.f)` to guard key lookups and `x = x || default` to set default values. Refinement types provide the flexibility to choose more restrictive types for operators, if desired, to statically prevent implicit coercions, which often lead to subtle programming errors. For example, we can restrict the negation operator to boolean values as follows.

$$\texttt{!} :: x{:}\mathit{Bool} \rightarrow \{\nu \,|\, \mathbf{ite}\ (x = \texttt{false})\ (\nu = \texttt{true})\ (\nu = \texttt{false})\}$$

Flow Sensitivity. Consider the following function that is like `negate` but first assigns the eventual result in the variable `x`.

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

To precisely reason about the different types of values stored in the (imperative) variable `x`, DJS maintains a flow-sensitive heap that can be *strongly updated* at each program point. As a result, DJS tracks that the updated value of `x` along the true case is $\{\nu \,|\, Num(x) \Rightarrow Num(\nu)\}$ (where x is the formal parameter initially stored in `x`) and along the false case is $\{\nu \,|\, \neg Num(x) \Rightarrow Bool(\nu)\}$. Thus, as before, DJS verifies that the return value (the new value of `x`) satisfies the specification.

Objects. JavaScript objects make heavy use of property extension and prototype inheritance to transitively resolve lookups.

```
var parent = {last: " Smith"};
var child = Object.create(parent);
child.first = "Bob";
child.first + child.last; // "Bob Smith"
```

For object extension, strong updates allow DJS to track that the "`first`" property is added to `child`. For prototypes, DJS precisely tracks parent links between objects in the heap, and *unrolls* prototype chains to match the semantics of object operations. For example, the type of the value retrieved by `child.last` is

$$\begin{aligned}
\{\ \nu \,|\, &\mathbf{if}\ \mathit{has}(\texttt{child}, \text{“}\texttt{last}\text{”})\ \mathbf{then}\ \nu = \mathit{sel}(\texttt{child}, \text{“}\texttt{last}\text{”}) \\
&\mathbf{elif}\ \mathit{has}(\texttt{parent}, \text{“}\texttt{last}\text{”})\ \mathbf{then}\ \nu = \mathit{sel}(\texttt{parent}, \text{“}\texttt{last}\text{”}) \\
&\mathbf{else}\ \nu = \texttt{undefined}\ \}
\end{aligned}$$

which is a subtype of $\{\nu \,|\, \nu = \mathit{sel}(\texttt{parent}, \text{“}\texttt{last}\text{”})\}$ given what we know about `child` and `parent`. Furthermore, we use *uninterpreted heap symbols* to reason about portions of the heap that are statically unknown. This allows DJS to verify that the property lookup in `if (k in x) x[k]` does *not* return `undefined` (unless the type of `x[k]` includes `undefined`, of course) even when *nothing* is known about the prototype chain of `x`.

Arrays as Arrays. Arrays are (mostly) ordinary prototype-based objects with string keys, but JavaScript programmers (and optimizing JIT compilers) commonly treat arrays as if they are traditional "packed" arrays with integer "indices" zero to "size" minus one. DJS reconciles this discrepancy by maintaining the following invariants about every array `a` :: $Arr(T)$.

1. `a` contains the special "`length`" key.
2. All other "own" keys of `a` are (strings that coerce to) integers.
3. For all integers i, either `a` maps the key i to a value of type T, or it has no binding for i.
4. All inherited keys of `a` are "safe" (*i.e.* non-integer) strings.

Furthermore, we use the uninterpreted predicate $packed(\texttt{a})$ to describe arrays that also satisfy the following property, where $len(\texttt{a})$ is an uninterpreted function symbol.

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

These invariants allow DJS to reason locally (without considering the prototype chain of `a`) that for any integer, `a[i]` produces a value of type $\{\nu \mid \nu :: T \vee \nu = \texttt{undefined}\}$, and that if $0 \leq \texttt{i} < len(\texttt{a})$, then `a[i]` *definitely* has type T. We assign types to array-manipulating operations, including the `Array.prototype.push` and `Array.prototype.pop` functions that all arrays inherit, to maintain these invariants and treat packed arrays precisely when possible.

Tuples. Arrays are used as finite tuples in several idiomatic ways.

```
var a0 = [0, 1, 2];
var a1 = []; a1[0] = 0; a1[1] = 1; a1[2] = 2;
var a2 = []; a2.push(0); a2.push(1); a2.push(2);
```

For `a1` and `a2`, DJS is able to track that the array updates — even when going through the `Array.prototype.push` native function that is inherited by `a` — maintain the invariant that the arrays are packed. Thus, each of the arrays has the following type.

$$\{\nu \mid \nu :: Arr(Int) \wedge packed(\nu) \wedge len(\nu) = 3\}$$

Benchmarks. We are actively working on our implementation (available at `ravichugh.com/nested`). So far, we have tested on 300 lines of unannotated benchmarks from several sources including *JavaScript: The Good Parts* and the SunSpider and V8 microbenchmark suites. Figure 1 summarizes our current results, where for each example: "Un" is the number of (non-whitespace, non-comment) lines of code in the *unannotated* benchmark; "Ann" is the lines of code in the annotated DJS version (including comments because they contain DJS annotations); "Time" is the running time rounded to the nearest second; and "Queries" is the number of validity queries issued to Z3 during type checking.

Taken together, the set of benchmarks rely on the gamut of features in the type system, requiring type invariants that describe relationships between parent and child objects, between the contents of imperative variables and arrays across iterations of a loop, and intersections of function types to encode control-flow invariants.

Annotation Burden and Running Time. As Figure 1 shows, our annotated benchmarks are approximately 1.7 times as large (70% overhead) as their unannotated versions on average. In our experience, a significant portion of the annotation burden is boilerplate — unrelated to the interesting typing invariants — that fall into a small number of patterns, which we have started to optimize.

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

Figure 1. Benchmarks (Un: LOC without annotations; Ann: LOC with annotations; Queries: Number of Z3 queries; Time: Running time in seconds)

The running time of our type checker is acceptable for small examples, but less so as the number of queries to the SMT solver increases. We have not yet spent much effort to improve performance, but we have implemented a few optimizations that have already reduced the number of SMT queries. There is plenty of room for future work to further improve both the annotation overhead as well as performance.

Conclusion. We have found that the full range of features in DJS are indeed required, but that many examples fall into patterns that do not simultaneous exercise all features. Therefore, we believe that future work on desugaring and on type checking can treat common cases specially in order to reduce the annotation burden and running time, and fall back to the full expressiveness of the system when necessary. In addition, we are working to extend DJS with support for additional features, including more general support for recursive types, for the `apply` and `call` forms (often used, for example, to set up inheritance patterns), and variable-arity functions. We believe that Dependent JavaScript is a promising approach for supporting real-world dynamic languages like JavaScript.

References

[1] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent Types for JavaScript. April 2012. `http://arxiv.org/abs/1112.4106`.

[2] Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested Refinements: A Logic for Duck Typing. In *POPL*, 2012.

[3] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *ECOOP*, 2010.

[4] Frederick Smith, David Walker, and Greg Morrisett. Alias Types. In *ESOP*, 2000.