Dependent Types for JavaScript — Appendix *

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

A. Additional Definitions

We presented many parts of System !D and DJS in §3, §4, and §5. In this appendix, we consider some details that did not fit in that presentation, as well as our treatment of *break* and *label* expressions to facilitate the desugaring of control operators in JavaScript. Then, in Appendix B, we outline how to extend System !D to support better *location polymorphism*.

A.1 Syntax

In addition to the expression syntax in Figure 7, System !D includes the following forms:

$$e \quad ::= \quad \cdots \mid @x : e \mid \texttt{break}\, @x\; v$$

An expression $@x : e$ labels the enclosed expression, and a break expression $\texttt{break}\ @x\ v$ terminates execution of the innermost expression labeled $@x$ within the function currently being evaluated and produces the result v. If no such labeled expression is found, evaluation becomes stuck. Label and break expressions are included to translate the control flow operations of DJS.

To analyze label and break expressions, the expression typing relation uses a *label environment* Ω (in addition to type and heap environments), where each binding records the world that the expression labeled $@x$ is expected to satisfy.

$$\Omega \quad ::= \quad \varnothing \mid \Omega, @x{:}(T/\hat{\Sigma})$$

A.2 Well-Formedness

The well-formedness relations, defined in Figure 14, are largely straightforward. We use the procedure $\mathsf{Binders}$ to collect all of the binders in a world or heap.

A.3 Subtyping

Figure 15 presents more of the subtyping relations.

Implication. As in System D, subtyping on refinement types reduces to implication of refinement formulas, which are discharged by a combination of uninterpreted, first-order reasoning and syntactic subtyping. If the SMT solver alone cannot discharge an implication obligation (I-VALID), the formula is rearranged into conjunctive normal form (I-CNF), and goals of the form $w :: U$ are discharged by a combination of uninterpreted reasoning and syntactic subtyping (I-IMPSYN).

We write $[\![T]\!]$ for the *embedding* of a type as a formula, a straightforward definition [3] that lifts to environments $[\![\Gamma]\!]$,

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

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

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

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

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

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

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

$$\frac{\Gamma_1 = \Gamma, \overline{A}, \overline{L}, \overline{H} \quad \Gamma_1 \vdash W_1 \quad \Gamma_1, \mathsf{Binders}(W_1) \vdash W_2}{\Gamma \vdash \forall [\overline{A}; \overline{L}; \overline{H}]\, W_1 \to W_2}$$

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

Well-Formed Worlds $\boxed{\Gamma \vdash W}$

$$\frac{\Gamma \vdash S \quad \Gamma, x{:}\,Top \vdash \hat{\Sigma}}{\Gamma \vdash x{:}S / \hat{\Sigma}}$$

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

$$\frac{H \in \Gamma \quad \Gamma, \mathsf{Binders}(\hat{h}) \vdash \hat{h} \quad \text{no duplicate locations in } \hat{h}}{\Gamma \vdash (H, \hat{h})}$$

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

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

Figure 14. Well-formedness for System !D

* This report supplements our OOPSLA 2012 paper [2].

Subtyping (see Figure 8) $\boxed{\Gamma \vdash T_1 \sqsubseteq T_2}$

Syntactic Subtyping (extends Figure 8) $\boxed{\Gamma \vdash U_1 <: U_2}$

$$\frac{\Gamma \vdash W_{21} \sqsubseteq W_{11};\ \pi \qquad \Gamma, [\![W_{21}]\!] \vdash \pi W_{12} \sqsubseteq W_{22};\ \pi'}{\Gamma \vdash \forall[\overline{A};\overline{L};\overline{H}]\ W_{11} \to W_{12} <: \forall[\overline{A};\overline{L};\overline{H}]\ W_{21} \to W_{22}}\ \text{[U-ARROW]}$$

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

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

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

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

$$\frac{\begin{array}{c}\Gamma \vdash T_1 \sqsubseteq T_2 \qquad \hat{h}_1 \sim \hat{h}_2;\ \pi \\ \pi' = \pi[x_1/x_2] \qquad \Gamma, [\![\hat{h}_1]\!] \Rightarrow \pi[\![\hat{h}_2]\!]\end{array}}{\Gamma \vdash x_1{:}T_1/(H,\hat{h}_1) \sqsubseteq x_2{:}T_2/(H,\hat{h}_2)}$$

Heap Matching $\boxed{\hat{h}_1 \sim \hat{h}_2;\ \pi}$

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

$$\frac{\hat{h}_1 \sim \hat{h}_2;\ \pi}{\hat{h}_1 \oplus (\tilde{\ell} \mapsto \theta) \sim \hat{h}_2 \oplus (\tilde{\ell} \mapsto \theta);\ \pi}$$

$$\frac{\hat{h}_1 \sim \hat{h}_2;\ \pi}{\hat{h}_1 \oplus (\ell \mapsto x{:}T) \sim \hat{h}_2 \oplus (\ell \mapsto y{:}S);\ \pi[x/y]}$$

$$\frac{\hat{h}_1 \sim \hat{h}_2;\ \pi}{\hat{h}_1 \oplus (\ell \mapsto \langle x{:}T, \ell'\rangle) \sim \hat{h}_2 \oplus (\ell \mapsto \langle y{:}S, \ell'\rangle);\ \pi[x/y]}$$

Figure 15. Subtyping for System !D

heap bindings $[\![\hat{h}]\!]$, heaps $[\![\hat{\Sigma}]\!]$, and worlds $[\![W]\!]$. Because heap binders may refer to each other in any order (recall that a heap can be thought of as a dependent tuple, where each component is named with a binder), the embedding of a heap starts by inserting dummy bindings so that all binders in scope for the type of each heap binding. For example:

$$\hat{\Sigma}_0 \doteq (H_0, (\ell_1 \mapsto x{:}T_1) \oplus (\ell_2 \mapsto y{:}T_2))$$
$$[\![\hat{\Sigma}_0]\!] = [\![x{:}\,Top]\!], [\![y{:}\,Top]\!], [\![T_1]\!](x), [\![T_2]\!](y)$$

Syntactic Subtyping. The U-ARROW rule for function types is familiar, treating input worlds contravariantly and output worlds covariantly.

Worlds. In order to check world subtyping, the judgement $\Gamma \vdash x_1{:}T_1/(H,\hat{h}_1) \sqsubseteq x_2{:}T_2/(H,\hat{h}_2)$ checks that T_1 is a subtype of T_2 and that the heaps agree on the "deep" part H. Then, it checks that the structure of the "shallow" parts match — using a heap matching relation that uses a $\equiv$ operator (not shown) that permutes bindings as necessary — and creates a substitution π of binders from $\hat{h}_2$ to $\hat{h}_1$. Finally, the heap bindings, which can be thought of as dependent tuples, are embedded as formulas and checked by implication.

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

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

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

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

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

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

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

Heap Environment Matching $\boxed{\Sigma \sim \hat{\Sigma};\ \pi}$ $\boxed{h \sim \hat{h};\ \pi}$

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

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

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

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

Figure 16. Value type checking for System !D

A.4 Value Typing

We supplement our discussion of value typing, defined in Figure 16, from §4. The T-EXTEND rule for dictionaries is straightforward. The T-LOC rule assigns run-time location r (which appears during evaluation, but not in source programs) a reference type corresponding to its compile-time location, using the mapping StaticLoc. Notice that, unlike the version of the rule from Figure 9, the rule T-FUN uses an empty label environment to type check function bodies, so that break expressions cannot cross function boundaries.

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

Rules from Figure 10, updated with label environments:

$$\text{[T-VAL]}\ \frac{\Gamma;\ \Sigma \vdash v :: T}{\Gamma;\ \Sigma;\ \Omega \vdash v :: T/\Sigma} \qquad \frac{\Gamma;\ \Sigma;\ \Omega \vdash e_1 :: T_1/\Sigma_1 \quad \Gamma, x{:}T_1;\ \Sigma_1;\ \Omega \vdash e_2 :: T_2/\Sigma_2}{\Gamma;\ \Sigma;\ \Omega \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 :: \exists x{:}T_1.\ T_2/\Sigma_2}\ \text{[T-LET]}$$

$$\frac{\Gamma;\ \Sigma \vdash v :: S \quad \Gamma, \mathit{truthy}(v);\ \Sigma;\ \Omega \vdash e_1 :: T_1/\Sigma_1 \quad \Gamma, \mathit{falsy}(v);\ \Sigma;\ \Omega \vdash e_2 :: T_2/\Sigma_2 \quad T/\Sigma' = \mathsf{Join}(v, T_1/\Sigma_1, T_2/\Sigma_2)}{\Gamma;\ \Sigma;\ \Omega \vdash \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 :: T/\Sigma'}\ \text{[T-IF]}$$

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

$$\text{[T-SETREF]}\ \frac{\begin{array}{c}\Gamma;\ \Sigma \vdash (v_1,\ v_2) :: (\mathit{Ref}\,\ell,\ T)\\ \Sigma \equiv \Sigma_0 \oplus (\ell \mapsto v) \quad \Sigma' = \Sigma_0 \oplus (\ell \mapsto v_2)\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash v_1 := v_2 :: \{x \mid x = v_2\}/\Sigma'} \qquad \frac{\begin{array}{c}\ell_1 \notin \mathit{dom}(\Sigma) \quad \Gamma;\ \Sigma \vdash (v_1,\ v_2) :: (\mathit{Dict},\ \mathit{Ref}\,\ell_2)\\ \Sigma \equiv \Sigma_0 \oplus (\ell_2 \mapsto \langle v', \ell_3\rangle) \quad \Sigma' = \Sigma \oplus (\ell_1 \mapsto \langle v_1, \ell_2\rangle)\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash \mathtt{newobj}\ \ell_1\ v_1\ v_2 :: \mathit{Ref}\,\ell_1/\Sigma'}\ \text{[T-NEWOBJ]}$$

$$\frac{\begin{array}{c}\Gamma;\ \Sigma \vdash v_1 :: \forall[\overline{A}; \overline{L}; \overline{H}]\ W_1 \to W_2 \quad \Gamma;\ \Sigma \vdash v_2 :: T_2 \quad \Gamma \vdash [\overline{T}/\overline{A}] \quad \Gamma \vdash [\overline{m}/\overline{M}] \quad \Gamma \vdash [\overline{\hat{\Sigma}}/\overline{H}]\\ W_2' = \mathsf{Freshen}(W_2) \quad (W_1', W_2'') = \mathsf{Unroll}(\mathsf{HInst}(\mathsf{LInst}(\mathsf{TInst}((W_1, W_2'), \overline{A}, \overline{T}), \overline{L}, \overline{\ell}), \overline{H}, \overline{\hat{\Sigma}}))\\ \Gamma \vdash T_2/\Sigma \models W_1';\ \pi \quad W_1' = x{:}T_{11}/\hat{\Sigma}_{11} \quad \pi' = \pi[v_2/x] \quad \pi' W_2'' = x'{:}T_{12}/\hat{\Sigma}_{12} \quad \mathsf{HeapEnv}(\hat{\Sigma}_{12}) = (\overline{y}{:}\overline{S}, \Sigma_{12})\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash [\overline{T}; \overline{\ell}; \overline{\hat{\Sigma}}]\ v_1\ v_2 :: \exists x'{:}T_{12}.\ \exists \overline{y}{:}\overline{S}.\ \{z \mid z = x'\}/\Sigma_{12}}\ \text{[T-APP]}$$

Additional rules:

$$\text{[T-AS]}\ \frac{\Gamma \vdash T \quad \Gamma;\ \Sigma;\ \Omega \vdash e :: T/\Sigma}{\Gamma;\ \Sigma;\ \Omega \vdash e\ \mathtt{as}\ T :: T/\Sigma} \qquad \frac{\Gamma;\ \Sigma;\ \Omega \vdash e :: S/\Sigma' \quad \Gamma \vdash S \sqsubseteq T \quad \Gamma \vdash T}{\Gamma;\ \Sigma;\ \Omega \vdash e :: T/\Sigma'}\ \text{[T-SUB]}$$

[T-FREEZE]

$$\frac{\begin{array}{c}\Gamma;\ \Sigma \vdash v :: \mathit{Ref}\,\ell \quad \Gamma(\tilde{\ell}) = (T, \ell')\\ \Sigma \equiv \Sigma_0 \oplus (\tilde{\ell} \mapsto \theta) \oplus (\ell \mapsto \langle v', \ell'\rangle) \quad \theta = \mathtt{frzn}\ \text{or}\ \theta = \mathtt{thwd}\ \ell\\ \Gamma;\ \Sigma \vdash v' :: T \quad \Sigma' = \Sigma_0 \oplus (\tilde{\ell} \mapsto \mathtt{frzn})\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash \mathtt{freeze}\ \tilde{\ell}\ \theta\ v :: \{\nu :: \mathit{Ref}\,\tilde{\ell} \wedge \nu \neq \mathtt{null}\}/\Sigma'}$$

[T-THAW]

$$\frac{\begin{array}{c}\Gamma;\ \Sigma \vdash v :: \mathit{Ref}\,\tilde{\ell} \quad \Gamma(\tilde{\ell}) = (T, \ell')\\ \Sigma \equiv \Sigma_0 \oplus (\tilde{\ell} \mapsto \mathtt{frzn}) \quad \Sigma' = \Sigma_0 \oplus (\tilde{\ell} \mapsto \mathtt{thwd}\ \ell) \oplus (\ell \mapsto \langle x, \ell'\rangle)\\ S = \{y \mid \mathsf{ite}\ (v = \mathtt{null})\ (y = \mathtt{null})\ (y :: \mathit{Ref}\,\ell)\}\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash \mathtt{thaw}\ \ell\ v :: \exists x{:}T.\ S/\Sigma'}$$

$$\text{[T-LABEL]}\ \frac{\begin{array}{c}\Omega' = \Omega, @x{:}(T/\hat{\Sigma}) \quad \Gamma;\ \Sigma;\ \Omega' \vdash e :: T/\Sigma'\\ \Gamma \vdash \Sigma' \models \hat{\Sigma};\ \pi \quad \mathsf{HeapEnv}(\hat{\Sigma}) = (\overline{x}{:}\overline{S}, \Sigma'')\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash @x : e :: \exists \overline{x}{:}\overline{S}.\ T/\Sigma''} \qquad \frac{\begin{array}{c}\Omega(@x) = T/\hat{\Sigma}\\ \Gamma;\ \Sigma \vdash v :: T \quad \Gamma \vdash \Sigma \models \hat{\Sigma};\ \pi\end{array}}{\Gamma;\ \Sigma;\ \Omega \vdash \mathtt{break}\ @x\ v :: \{\mathit{false}\}/\Sigma}\ \text{[T-BREAK]}$$

Figure 17. Expression type checking for System !D

A.5 Expression Typing

When we presented expression typing in §4.4, we ignored break and label expressions, so the typing judgement referred only to type and heap environments. To account for control operators, the expression typing judgement is of the form $\Gamma;\ \Sigma;\ \Omega \vdash e :: T/\Sigma'$, where a label environment is an additional input. We define the typing rules in Figure 17 and supplement our previous discussion. The T-AS and T-SUB rules are straightforward. Aside from the rules for label and break expressions, label environments Ω play no interesting role. The rules we discussed in §4.4 carry over directly to the formulation with label environments.

Weak Location Bindings. For simplicity, we assume that the initial type environment contains all the weak location bindings $(\tilde{\ell} \mapsto \langle T, \ell\rangle)$ required by the program.

Thaw and Freeze. To safely allow a weak location $\tilde{\ell}$ to be treated *temporarily* as strong, System !D ensures that $\tilde{\ell}$ has at most one corresponding *thawed* location at a time; if there is none, we say $\tilde{\ell}$ is *frozen*. The rule T-THAW thaws $\tilde{\ell}$ to a strong location ℓ (which we syntactically require be distinct from all other thawed locations for $\tilde{\ell}$) and updates the heap environment with thaw state `thwd` ℓ to track the correspondence. Subtyping allows `null` weak references, so the output type is `null` if the original reference is; otherwise, it is a reference to ℓ. Finally, the new heap also binds a value x of type T, the invariant for all values stored at $\tilde{\ell}$, and the output type introduces an *existential* so that x is in scope in the new heap.

The rule T-FREEZE serves two purposes, to merge a strong location ℓ into a weak (frozen) location $\tilde{\ell}$ and to *re-freeze* a thawed (strong) location ℓ that originated from $\tilde{\ell}$, as long as the heap value stored at ℓ satisfies the invariant required by $\tilde{\ell}$. The strong reference is guaranteed to be non-`null`, so the output type remembers that the frozen reference is, too. Compared to the presentation in [1], we have combined freeze and re-freeze into a single `freeze` expression that includes an explicit thaw state θ.

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

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

Type Instantiation. The TInst procedure processes has-type predicates in formulas as follows:

$$\mathsf{TInst}(w :: A, A, \{x \,|\, p\}) = p[w/x]$$
$$\mathsf{TInst}(w :: B, A, T) = w :: B$$

Join. The T-IF rule uses a Join operator, defined in Figure 18, that combines the type and heap environments along each branch (T_1 / Σ_1 and T_2 / Σ_2) such that the type and output heap for the overall if-expression (T / Σ') are in prenex form. The operator starts by using JoinTypes to move existential binders for the types to the top-level. Rearranging variables in this way is sound because we assume that, by convention, all let-bound variables in a program are distinct. Then, we use JoinHeaps to combine the bindings in a heap environment one location at a time. We show a few representative equations in Figure 18, abusing notation in several ways. For example, we write $h \setminus \ell$ to denote that ℓ is not bound in h. When a location ℓ is bound in both heaps to values v_1 and v_2, respectively, JoinHeaps introduces a new binding y whose type is the join of v_1 and v_2. When a location ℓ is bound in only one heap, we use the dummy type Top to describe the (non-existent) value in the other heap. There is no danger that ℓ will be unsoundly dereferenced after the if-expression, since JoinTypes guards the types of references $\mathit{Ref}\,\ell$ with the appropriate guard predicates.

Control Operators. The T-LABEL rule for $@x : e$ binds the label $@x$ to an expected world $T / \hat{\Sigma}$ in the label environment Ω' used to check e, and expects that *all* exit points of e produce a value and heap environment that satisfy the expected world. The exit points are all `break` $@x\ v$ expressions in e, as well as the "fall-through" of expression e for control flow paths that do not end with `break`; the T-BREAK rule handles the former cases, and the second and third premises of T-LABEL handle the latter. If all exit points satisfy the expected world, we use the HeapEnv procedure to convert

Desugaring (extends Figure 12) $\quad \boxed{\langle\!\langle\, \texttt{e} \,\rangle\!\rangle = e}$

$\langle\!\langle\, \texttt{function}\ (\overline{\texttt{x}})\ /\!*: T *\!/\ \{\ \texttt{e}\ \}\,\rangle\!\rangle =$ [DS-FUNC]

$\quad \lambda(\mathit{this}, \mathit{arguments}).$
$\quad\quad \texttt{let}\ (\_\texttt{x}_0, \ldots) = (\texttt{ref}\ a_{x_0}\ (\texttt{get}\ \mathit{arguments}\ \text{"0"}), \ldots)\ \texttt{in}$
$\quad\quad @\mathit{return} : \langle\!\langle \texttt{e} \rangle\!\rangle$

$\langle\!\langle\, \texttt{function F}(\overline{\texttt{x}})\ /\!*: \#\texttt{ctor}\ T *\!/\ \{\ \texttt{e}\ \}\,\rangle\!\rangle =$ [DS-CTOR]

$\quad \texttt{let}\ f = \lambda(\mathit{this}, \mathit{arguments}).$
$\quad\quad \texttt{let}\ (\_\texttt{x}_0, \ldots) = (\texttt{ref}\ a_{x_0}\ (\texttt{get}\ \mathit{arguments}\ \text{"0"}), \ldots)\ \texttt{in}$
$\quad\quad @\mathit{return} : \langle\!\langle \texttt{e} \rangle\!\rangle\ \texttt{in}$
$\quad \texttt{let}\ p = \texttt{newobj}\ a_{F\mathit{proto}}\ \{\}\ (\mathit{pro}(\texttt{Object}))\ \texttt{in}$
$\quad \texttt{let}\ d = \{\ \text{"\_\_code\_\_"} = f\ \texttt{as}\ \langle\!\langle T \rangle\!\rangle;\ \text{"prototype"} = p\ \}\ \texttt{in}$
$\quad \texttt{newobj}\ a_F\ d\ (\mathit{pro}(\texttt{Function}))$

$\langle\!\langle\, \texttt{return e}\,\rangle\!\rangle = \texttt{break}\ @\mathit{return}\ \langle\!\langle \texttt{e} \rangle\!\rangle$ [DS-RETURN]

$\langle\!\langle\, /\!*: T *\!/\ \texttt{while}\ (\texttt{e}_{\texttt{cond}})\ \{\ \texttt{e}_{\texttt{body}}\ \}\,\rangle\!\rangle =$ [DS-WHILE]

$\quad @\mathit{break} : \texttt{letrec}\ \mathit{loop} :: T = \lambda().$
$\quad\quad\quad \texttt{if}\ \langle\!\langle \texttt{e}_{\texttt{cond}} \rangle\!\rangle\ \texttt{then}\ (\langle\!\langle \texttt{e}_{\texttt{body}} \rangle\!\rangle; \mathit{loop}\ ())$
$\quad\quad\quad \texttt{else undefined in}\ \mathit{loop}\ ()$

$\langle\!\langle\, \texttt{break}\,\rangle\!\rangle = \texttt{break}\ @\mathit{break}\ \texttt{undefined}$ [DS-BREAK]

$\langle\!\langle\, /\!*: \#\texttt{thaw}\ \ell\ \texttt{e} *\!/\,\rangle\!\rangle = \texttt{thaw}\ \ell\ \langle\!\langle \texttt{e} \rangle\!\rangle$ [DS-THAW]

$\langle\!\langle\, /\!*: \#\texttt{freeze}\ \tilde{\ell}\ \theta\ \texttt{e} *\!/\,\rangle\!\rangle = \texttt{freeze}\ \tilde{\ell}\ \theta\ \langle\!\langle \texttt{e} \rangle\!\rangle$ [DS-FREEZE]

$\langle\!\langle\, \texttt{assert(e)}\,\rangle\!\rangle = \langle\!\langle \texttt{e} \rangle\!\rangle\ \texttt{as}\ \{\nu = \texttt{true}\}$ [DS-ASSERT]

Figure 19. Desugaring DJS to System !D

the heap type into a heap environment, like in the T-APP rule. Notice that T-BREAK derives the type $\{\mathit{false}\}$ because a `break` immediately completes the evaluation context, thus making the subsequent program point unreachable.

A.6 Desugaring

In Figure 19, we show more of the desugaring rules.

Functions and Constructors. As discussed in §5, we desugar non-constructor functions (DS-FUNC) to scalar function values and constructor functions (DS-CTOR) to objects. Following λ_{JS} [4], we wrap each desugared function body with the label $@\mathit{return}$, which facilitates the desugaring of `return` statements (DS-RETURN). We desugar named, recursive DJS functions via the standard `letrec` encoding using `fix`; we omit this rule from Figure 19.

DS-CTOR first creates a fresh object at location a_F with prototype `Function.prototype`, then stores the desugared constructor function in the "`__code__`" field, and finally creates an empty object at location $a_{F_{proto}}$ that is stored in the "`prototype`" field, to be used when creating an object with this constructor (DS-NEW).

Loops. Following λ_{JS}, the DS-WHILE desugars `while` loops to recursive functions (we write `letrec` as syntactic sugar for the standard encoding using `fix`). As such, a (function type) annotation describes the invariants that hold before and after each iteration. A $@\mathit{break}$ label around the desugared loop body facilitates the desugaring of `break`

$$\mathsf{Join}(b, S_1/\Sigma_1, S_2/\Sigma_2) = \exists \overline{x}{:}\overline{T}.\ \exists \overline{y}{:}\overline{T'}.\ S/\Sigma \quad \text{where } \mathsf{JoinTypes}(b, S_1, S_2) = \exists \overline{x}{:}\overline{T}.\ S \text{ and } \mathsf{JoinHeaps}(b, \Sigma_1, \Sigma_2) = (\exists \overline{y} : \overline{T'}, \Sigma)$$

$$\mathsf{JoinTypes}(b, S_1, S_2) = \{(b = \mathtt{true} \Rightarrow [\![S_1]\!]) \wedge (b = \mathtt{false} \Rightarrow [\![S_2]\!])\}$$

$$\mathsf{JoinTypes}(b, (\exists \overline{x_1}{:}\overline{T_1}.\ S_1), S_2) = \exists \overline{x_1}{:}(\mathsf{JoinTypes}(b, \overline{T_1}, \overline{Top})).\ \mathsf{JoinTypes}(b, S_1, S_2)$$

$$\mathsf{JoinTypes}(b, S_1, (\exists \overline{x_2}{:}\overline{T_2}.\ S_2)) = \exists \overline{x_2}{:}(\mathsf{JoinTypes}(b, \overline{Top}, \overline{T_2})).\ \mathsf{JoinTypes}(b, S_1, S_2)$$

$$\mathsf{JoinTypes}(b, (\exists \overline{x_1}{:}\overline{T_1}.\ S_1), (\exists \overline{x_2}{:}\overline{T_2}.\ S_2)) = \exists \overline{x_1}{:}(\mathsf{JoinTypes}(b, \overline{T_1}, \overline{Top})).\ \exists \overline{x_2}{:}(\mathsf{JoinTypes}(b, \overline{Top}, \overline{T_2})).\ \mathsf{JoinTypes}(b, S_1, S_2)$$

$$\mathsf{JoinHeaps}(b, (\ell \mapsto v_1) \oplus h_1, (\ell \mapsto v_2) \oplus h_2) = (\exists y{:}\mathsf{JoinTypes}(b, \{x \,|\, x = v_1\}, \{x \,|\, x = v_2\}), (\ell \mapsto y) \oplus \mathsf{JoinHeaps}(b, h_1, h_2))$$

$$\mathsf{JoinHeaps}(b, (\ell \mapsto v_1) \oplus h_1, h_2 \backslash \ell) = (\exists y{:}\mathsf{JoinTypes}(b, \{x \,|\, x = v_1\}, Top), (\ell \mapsto y) \oplus \mathsf{JoinHeaps}(b, h_1, h_2))$$

$$\mathsf{JoinHeaps}(b, h_1 \backslash \ell, (\ell \mapsto v_2) \oplus h_2) = (\exists y{:}\mathsf{JoinTypes}(b, Top, \{x \,|\, x = v_2\}), (\ell \mapsto y) \oplus \mathsf{JoinHeaps}(b, h_1, h_2))$$

Figure 18. Environment join

statements (DS-BREAK). We elide similar mechanisms for `do-while` loops, `for` loops, `for-in` loops, and `continue` statements.

B. Extensions

We now outline two ways to increase the expressiveness of location polymorphism in System !D.

B.1 Weak Location Polymorphism

So far, we have universally quantified function types over strong locations. We can make several changes to allow quantification over weak locations as well. First, we extend the syntax of locations.

$$\tilde{\ell} \ ::= \ \cdots \mid \tilde{L} \qquad m \ ::= \ \ell \mid \tilde{\ell} \qquad M \ ::= \ L \mid \tilde{L}$$

We use $\tilde{L}$ to range over weak location variables, and we extend the grammar of weak locations $\tilde{\ell}$ to include them (in addition to weak location constants $\tilde{a}$). We also define m (resp. M) to range over *arbitrary* locations (resp. location variables). Next, we extend the syntax of function types and function application.

$$\begin{array}{lcl} e & ::= & \cdots \ \mid \ [\overline{T}; \overline{m}; \overline{\hat{\Sigma}}]\ v_1\ v_2 \\ U & ::= & \cdots \ \mid \ \forall [\overline{A}; \overline{M}; \overline{H}]\ \Psi / W_1 \to W_2 \\ \Psi & ::= & (\tilde{\ell} \mapsto \langle T, \ell \rangle) \ \mid \ \Psi_1 \oplus \Psi_2 \ \mid \ \varnothing \end{array}$$

A function type is now parametrized over arbitrary location variables $\overline{M}$ and a *weak heap* Ψ of bindings that describe weak location variables. To match, function application now includes location arguments $\overline{m}$ rather than $\overline{\ell}$; typing must ensure that strong location variables L (resp. weak location variables $\tilde{L}$) are instantiated only with strong locations ℓ (resp. weak locations $\tilde{\ell}$).

A function type refers to a weak heap only in the domain of the function, because weak locations are flow-insensitive and do not vary at different program points. Before, we assumed that the initial typing environment contained bindings for all weak locations. The new syntax of function types replaces this convention by *abstracting* over weak locations. Consequently, the function application rule must check that the declared weak heap Ψ of a function type is satisfiable given the current heap environment Σ at a call site (after substitution of all polymorphic variables).

B.2 Existential Locations

Universally quantifying over all locations, including simple locations, clutters function types and applications with additional arguments, and also exposes locations that are "internal" or "local" to the desugared System !D program and *not* accessible in the original DJS program.

Consider the following example; we refer to the original function as f and the desugared version as f'.

```
function(x) {
 var y = x
 return y.f }
```

```
fun x -> let _x = ref x
      in let _y = ref (deref _x)
      in getElem (deref _y, "f")
```

We might annotate the DJS function f with the type

$$\begin{array}{l} \forall L, L'.\ Ref\, L / (L \mapsto \langle Dict, L' \rangle) \\ \to Top / (L \mapsto same) \end{array}$$

and the desugared version would have the type

$$\begin{array}{l} \forall L, L', L_x, L_y.\ Ref\, L / (L \mapsto \langle Dict, L' \rangle) \\ \to Top / (L \mapsto same) \oplus (L_x \mapsto Ref\, L) \oplus (L_y \mapsto Ref\, L) \end{array}$$

that uses additional location variables for the references inserted by the translation. Although it is straightforward to mechanically desugar function types in this manner, the additional location parameters at function calls increase the manual annotation burden or, more likely since we cannot expect DJS programmer to write them, the burden on the type system to infer them.

Instead, we can introduce *existential location types* into the system and write the following type for the desugared function f'.

$$\forall L, L'.\ \mathit{Ref}\,L \,/\, (L \mapsto \langle \mathit{Dict}, L' \rangle)$$
$$\to \exists L_x, L_y.\ \mathit{Top} \,/\, (L \mapsto \mathit{same}) \oplus (L_x \mapsto \mathit{Ref}\,L) \oplus (L_y \mapsto \mathit{Ref}\,L)$$

Notice that the *output* world uses existentials to name the (strong, simple) locations inserted by desugaring. As a result, a call to this function need not instantiate the local locations; instead, the type system can generate *fresh* location constants (*i.e.,* skolemize) for the existential locations.

We provide a sketch of how to extend System !D with existential locations. First, we extend the syntax of types.

$$T \quad ::= \quad \cdots \ \mid \ \exists \ell.\ T$$

We intend that existential locations only appear in *positive* positions of function types, which we can compute in similar fashion to the Poles procedure from System D [3] that tracks *polarity* of types nested within formulas. Effectively, we require that every function type be of the form

$$\forall [\overline{A}; \overline{L}; \overline{H}]\ x\!:\!(\{y \,|\, p\}) \,/\, \hat{\Sigma} \to x'\!:\!(\exists \overline{\ell}.\ \{y' \,|\, p'\}) \,/\, \hat{\Sigma}'$$

where the input type is a refinement type and the output type is in a *prenex* form that requires all existentially-quantified locations to appear at the top-level and which prohibits existentially-quantified values. Intuitively, the locations $\overline{\ell}$ correspond to local reference cells that a function allocates when invoked and are inaccessible to callers.

To introduce existential locations for simple references (which are only used by desugaring), we use a new typing rule. For technical reasons, we use a `let`-expression to type check reference allocation *along with* a subsequent expression e as a way to describe the scope of ℓ.

$$\frac{\Gamma;\ \Sigma \vdash v \ ::\ T \qquad \Gamma, x\!:\!\mathit{Ref}\,\ell;\ \Sigma \oplus (\ell \mapsto v);\ \Omega \vdash e \ ::\ S \,/\, \Sigma'}{\Gamma;\ \Sigma;\ \Omega \vdash \texttt{let}\ x = \texttt{ref}\ \ell\ v\ \texttt{in}\ e \ ::\ \exists \ell.\ S \,/\, \Sigma'}$$

To facilitate algorithmic type checking, we ensure that existential locations are always prenex quantified in types. The Join procedure, used for conditionals, rearranges existential locations allocated on different branches to maintain this invariant.

Finally, we need to handle subtyping of existential location types. The simplest approach is to require that two types have the same quantifier structure.

$$\frac{\Gamma \vdash T_1 \sqsubseteq T_2}{\Gamma \vdash \exists \ell.\ T_1 \sqsubseteq \exists \ell.\ T_2}$$

For first-order functions, we can work around this limitation by playing tricks with dummy locations. For higher-order functions, however, the presence of existential locations limits expressivness by constraining the use of the heap. Abstracting over the mutable state of higher-order functions, however, can can be quite heavyweight (see *e.g.,* Hoare Type Theory [5]); adding more lightweight support in our setting is left for future work.

References

[1] A. Ahmed, M. Fluet, and G. Morrisett. L^3: A Linear Language with Locations. *Fundamenta Informaticae*, 77(4), June 2007.

[2] R. Chugh, D. Herman, and R. Jhala. Dependent Types for JavaScript. In *OOPSLA*, 2012.

[3] R. Chugh, P. M. Rondon, and R. Jhala. Nested Refinements: A Logic for Duck Typing. In *POPL*, 2012.

[4] A. Guha, C. Saftoiu, and S. Krishnamurthi. The Essence of JavaScript. In *ECOOP*, 2010.

[5] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the Awkward Squad. In *ICFP*, 2008.