Recency Types for Scripting Languages – Technical Report

Phillip Heidegger and Peter Thiemann

Universität Freiburg, Germany,
heidegger,thiemann@informatik.uni-freiburg.de

1 Extensions

Before presenting the proves we present an extended version of the core language for $\mathcal{RAC}$. Therefore we extend the syntax by an conditional and the type syntax by a second effect. §1.1 and §1.2 presents two extensions for $\mathcal{RAC}$ that are formalized. They are part of the proves. Their main goal is to allow more programs to pass the type checker, and to support the conditionals.

The type rules in the technical report make use of judgments defined in Fig. 1, as you will notice in Function Call and If in Fig. 11.

1.1 Access Effects

Up to now, functions are fairly sensitive to their calling context. Consider the following example, where the function h is called twice:

$$\begin{array}{l}\texttt{let } h = \lambda x.\begin{pmatrix}\texttt{let } x_1 = \texttt{new}^{\ell_2} \texttt{ in}\\ x_1.b := x;\ x_1\end{pmatrix} \texttt{ in}\\ \natural^{\ell}\texttt{let } x = \texttt{new}^{\ell} \texttt{ in}\\ x.a := 5;\\ \natural^{\ell_2}\texttt{let } \_ = h(x) \texttt{ in}\\ x.a := \text{"}\texttt{Hello}\text{"};\\ \natural^{\ell_2}\texttt{let } y = h(x) \texttt{ in}\\ \text{"}\texttt{Kitty}\text{"}\end{array} \tag{1}$$

As the type of h includes the referenced singleton environment of the call site, only one of the calls in line 4 and line 6 can be compatible with h's type $(\Sigma, \texttt{int}) \xrightarrow{\{\ell_2\},\emptyset} (\Sigma, \texttt{int})$: Line 4 requires $\Sigma = [\ell \mapsto \{a \mapsto \texttt{int}\}]$ whereas line 6 requires $\Sigma = [\ell \mapsto \{a \mapsto \texttt{string}\}]$.

Access effects address this problem. The access effect A of a function indicates the locations of singleton objects that the function may read or write, but which are not allocated inside the function. It imposes a lower bound on the domain of the singleton environment that must be passed to the function. The rule for function application splits the singleton environment at the call site and only matches the relevant part indicated by the access effect with the environment in the function type. The environment returned from the function is joined with

the rest of the call-site environment to rebuild the singleton environment after the call.

In the example, the type of h is $([], \texttt{int}) \xrightarrow{\{\ell_2\},\emptyset} ([], \texttt{int})$ indicating that h creates objects at location ℓ_2 but does not access other singleton objects. Thus, at both call sites in lines 4 and 6 an empty environment is passed to and returned from the function so that the singleton environment at each call site is undisturbed. The formal definition of the spitting of the most recent environment is presented in Fig. 1.

1.2 Conditionals – Joining Environments

Finally, conditionals pose a few problems in connection with singleton types. As the branches of a conditional may create different objects, the types and environments returned from the branches may differ in the locations mentioned in them as well as in whether they contain singleton or summary object types. As the typing rule for conditionals asks them to be identical, they have to be joined to type check the continuation of the conditional. The treatment of such objects depends on whether they are allocated with the same abstract location. If all else fails, demotions have to be inserted. But often it is possible to remain more precise than that.

Let's first consider an example where the same abstract location is allocated in both branches.

$$\begin{array}{l}\texttt{if } x \ (\texttt{let } x_1 = \texttt{new}^{\ell} \texttt{ in } x_1.a := 22;\ x_1) \\ \qquad (\texttt{let } x_2 = \texttt{new}^{\ell} \texttt{ in } x_2.a := \texttt{"xx"};\ x_2)\end{array} \tag{2}$$

In this case, the type of the conditional is $\texttt{obj}(@\ell)$ with a singleton heap type where ℓ maps to a type subsuming both `int` and `string`. Hence, most recent object types for identical locations are joined.

For a case with different locations, consider the next example.

$$\begin{array}{l}\texttt{let } f = \lambda z. \\ \quad (\texttt{let } o = \texttt{new}^{\ell_1} \texttt{ in } o.b := \texttt{"Hi"};\ o.c := z;\ o) \texttt{ in} \\ \texttt{let } g = \lambda z. \\ \quad (\texttt{let } o = \texttt{new}^{\ell_2} \texttt{ in } o.b := \texttt{false};\ o.c := z;\ o) \texttt{ in} \\ \natural^{\ell_1,\ell_2}\texttt{let } x = \texttt{if } e\ f(31)\ g(32) \texttt{ in} \\ \quad x.b := 5;\ x.b + x.c\end{array} \tag{3}$$

The types of the functions are as follows (ignoring effects):

$$\begin{array}{l} f : ([], \texttt{int}) \to ([\ell_1 \mapsto \{a{:}\texttt{string}; c{:}\texttt{int}\}], \texttt{obj}(@\ell_1)) \\ g : ([], \texttt{int}) \to ([\ell_2 \mapsto \{a{:}\texttt{bool}; c{:}\texttt{int}\}], \texttt{obj}(@\ell_2)) \end{array}$$

The type of the variable x is now $\texttt{obj}(@\{\ell_1, \ell_2\})$, which is a departure from our previous restriction which allowed only one location in a singleton object type. This relaxed singleton object type is not a problem for typing a read access

to the object (just take the join or the referenced locations), but performing a strong update on the object is tricky.

The type of x indicates that x holds either a recent ℓ_1-object or a recent ℓ_2-object. As a strong update overwrites the descriptions of both locations, there must be no further references to an ℓ_1- or ℓ_2-object because one of them would be out of sync with the description.

Our approach is to enhance the structure of the singleton heap environment to keep alternatives indicated by the operator $\|$. As an example, the heap environment at the write operation in (3) would look like this:

$$[\ell_1 \mapsto \{a : \texttt{string}; c : \texttt{int}\}] \parallel [\ell_2 \mapsto \{a : \texttt{bool}; c : \texttt{int}\}]$$

In this case, the strong update of x is possible because each heap alternative defines only one of ℓ_1 and ℓ_2. The heap environment after the update would be

$$[\ell_1 \mapsto \{a : \texttt{int}; c : \texttt{int}\}] \parallel [\ell_2 \mapsto \{a : \texttt{int}; c : \texttt{int}\}]$$

However, it is vital for the example that the function calls happen in the branches of the conditional. Suppose the programmer replaces the conditional in (3) with this functionally equivalent code:

$$\begin{array}{l} \texttt{let } x_1 = f(31) \texttt{ in} \\ \texttt{let } x_2 = g(32) \texttt{ in} \\ \texttt{let } x = \texttt{if } e\ x_1\ x_2\ \texttt{ in} \end{array} \tag{4}$$

With this modification, x receives the same type $\texttt{obj}(@\{\ell_1, \ell_2\})$ as in (3). However, in (4) the typing does not produce alternative environments so that both objects (ℓ_1 and ℓ_2) exist at the same time. In particular, if e evaluates to `true`, then $x = x_1$ and an update to $x.b$ would change the ℓ_1 object, but leave the ℓ_2 object unchanged. A strong update of both ℓ_1 and ℓ_2, however, would also change the type of ℓ_2 and hence the type of $x_2.b$ to `int` whereas $x_2.b$ still contains `false`. Hence, a strong update is unsound and the type system requires demotion beform performing the write operation.

Fig. 1 contains the rules for splitting and merging singleton environments at function calls and for joining them in the conditional rule. The rules for splitting are straightforward, but the rules for joining require some explanation.

The first two join rules capture the trival cases where one environment is empty. The second rule treats non-empty singleton environments with different domains. It first builds a common environment Σ_M which contains bindings for the locations in set L, which are defined in both environments, and then appends an alternative binding consisting of the original environments with the L-bindings removed. This removal operation keeps the structure of the environments thus creating potentially nested alternatives.

The fourth join rule recursively deals with environments with identical domain. It constructs a new environment which pointwise subsumes the entries in the original environments.

$$\boxed{\Sigma, A \vdash_S \Sigma, \Sigma} \qquad \emptyset, A \vdash_S \emptyset, \emptyset \qquad \frac{l \in A \qquad \Sigma, A \vdash_S \Sigma_1, \Sigma_2}{\Sigma(l : r), A \vdash_S \Sigma_1(l : r), \Sigma_2}$$

$$\frac{l \notin A \qquad \Sigma, A \vdash_S \Sigma_1, \Sigma_2}{\Sigma(l : r), A \vdash_S \Sigma_1, \Sigma_2(l : r)} \qquad \frac{\Sigma_1, A \vdash_S \Sigma_1', \Sigma_1'' \qquad \Sigma_2, A \vdash_S \Sigma_2', \Sigma_2''}{\Sigma_1 \parallel \Sigma_2, A \vdash_S \Sigma_1' \parallel \Sigma_2', \Sigma_1'' \parallel \Sigma_2''}$$

$$\boxed{\Sigma, \Sigma \vdash_J \Sigma} \qquad \Sigma, \emptyset \vdash_J \Sigma \qquad \emptyset, \Sigma \vdash_J \Sigma$$

$$\frac{\begin{array}{c}\mathrm{dom}(\Sigma_1) \neq \mathrm{dom}(\Sigma_2) \\ \mathrm{dom}(\Sigma_1) \neq \emptyset \qquad \mathrm{dom}(\Sigma_2) \neq \emptyset \qquad L = \mathrm{dom}(\Sigma_1) \cap \mathrm{dom}(\Sigma_2) \\ \Sigma_2' = \Sigma_2 \downarrow L \qquad \Sigma_1' = \Sigma_1 \downarrow L \qquad \Sigma_1'' = \Sigma_1 \uparrow L \qquad \Sigma_2'' = \Sigma_2 \uparrow L \qquad \Sigma_1', \Sigma_2' \vdash_J \Sigma_M\end{array}}{\Sigma_1, \Sigma_2 \vdash_J \Sigma_M(\Sigma_1'' \parallel \Sigma_2'')}$$

$$\frac{\begin{array}{c}\mathrm{dom}(\Sigma_1) = \mathrm{dom}(\Sigma_2) \neq \emptyset \qquad l \in \mathrm{dom}(\Sigma_1) \\ \Sigma_1 \uparrow \{l\}, \Sigma_2 \uparrow \{l\} \vdash_J \Sigma \qquad (\forall a \in \mathsf{Prop})\ (\Sigma_2(l)(a) <: r(a)) \wedge (\Sigma_1(l)(a) <: r(a))\end{array}}{\Sigma_1, \Sigma_2 \vdash_J \Sigma(l : r)}$$

Fig. 1. Rules for splitting and joining the singleton heap.

$$\begin{array}{rcl}
\mathsf{Value} \ni v & ::= & x \mid \mathtt{rec}\, f(y).e \mid \mathtt{udf} \mid \boxed{(q\ell, i)} \\
\mathsf{TopExpr} \ni e & ::= & v \mid \mathtt{let}^L\ x = s\ \mathtt{in}\ e \mid \boxed{\natural^L e} \\
\mathsf{Expr} \ni s & ::= & v \mid v(v) \mid \mathtt{new}^\ell \mid v.a \mid v.a := v \mid \mathtt{if}\ v\ e\ e \\
\mathsf{Qualifier} \ni q & ::= & \tilde{} \mid @ \\
\mathsf{Loc} \ni \ell & ::= & l_1 \mid l_2 \mid \ldots \\
\mathsf{Loc} \supseteq & & L, A \\
\mathsf{Prop} \ni & & a
\end{array}$$

Fig. 2. Expression syntax. Phrases marked in gray are not written by the programmer. They arise as intermediate steps in the semantics or are inserted automatically by elaboration.

$\mathbf{Z}$	set of integers
$\mathrm{fv}(e)$	free term variables in expression e
$e[x \mapsto v]$	capture-avoiding substitution of value v for x in e
$\mathrm{dom}(m)$	domain of map m
$m \downarrow X$	restrict map m to domain $\mathrm{dom}(m) \cap X$
$m \uparrow X$	restrict map m to domain $\mathrm{dom}(m) \setminus X$
$A \xrightarrow{fin} B$	set of partial finite functions from A to B
$m\{x \mapsto y\}$	map update: if $m' = m\{x \mapsto y\}$, then $m'(x) = y$ and $m'(x') = m(x')$, for all $x' \neq x$
$m\$a$	property access for $m \in \mathsf{Prop} \xrightarrow{fin} \mathsf{Value}$

$$\{\}\$a = \texttt{udf}$$
$$m\{a \mapsto v\}\$a = v$$
$$m\{b \mapsto v\}\$a = m\$a \qquad a \neq b$$

Fig. 3. Notation.

$$H \in \mathsf{Heap} = \mathsf{Loc} \times \mathbf{Z} \xrightarrow{fin} \mathsf{PropMap}$$
$$h \in \mathsf{PropMap} = \mathsf{Prop} \xrightarrow{fin} \mathsf{Value}$$
$$\mathcal{L} ::= \Box \mid \texttt{let}^L\ x = s\ \texttt{in}\ \Box$$

$$\textsc{S0App}\quad H, (\texttt{rec}\, f(x).e)(v) \to_0 H, e[f, x \mapsto \texttt{rec}\, f(x).e, v]$$
$$\textsc{S0Let}\quad H, \texttt{let}\ x = v\ \texttt{in}\ e \to_0 H, e[x \mapsto v]$$
$$\textsc{S0New}\quad H, \texttt{new}^\ell \to_0 H\{(\ell, i) \mapsto \{\}\}, (\tilde{}\ell, i) \quad \text{if } (\ell, i) \notin \mathrm{dom}(H)$$
$$\textsc{S0Rd}\quad H, (q\ell, i).a \to_0 H, H(\ell, i)\$a$$
$$\textsc{S0Wrt}\quad H, (q\ell, i).a := v \to_0 H\{(\ell, i)(a) \mapsto v\}, \texttt{udf}$$
$$\textsc{S0IfT}\quad H, \texttt{if}\ v\ e_1\ e_2 \to_0 H, e_1 \text{ if } v \neq \texttt{udf}$$
$$\textsc{S0IfF}\quad H, \texttt{if udf}\ e_1\ e_2 \to_0 H, e_2$$
$$\textsc{S0Let}'\quad \frac{H, s \to_0 H', \mathcal{L}[v]}{H, \texttt{let}^L\ x = s\ \texttt{in}\ e'' \to_0 H', \mathcal{L}[\texttt{let}^L\ x = v\ \texttt{in}\ e'']}$$

Fig. 4. Small-step operational semantics.

2 Proves

The figures 2-12 restate some relations from the paper, except Fig. 4. They are all extended by access effects, a conditional expression and most recent heap alternatives. Fig. 13 presents the typing for configurations.

In §2.2 a list summaries the differences between the paper and the technical report.

2.1 Dynamic Semantics

Fig. 4 defines a straightforward dynamic semantics for $\mathcal{RAC}$. It is a small-step semantics defined on configurations H, e where H is a heap and e is an expression. A heap maps a pair (ℓ, i) of a location and an integer to some property map h,

$$
\begin{array}{lll}
H & \in & \mathsf{Heap} = \mathsf{Loc} \times \mathbf{Z} \xrightarrow{fin} \mathsf{PropMap} \\
\mathcal{H} & \in & \mathsf{Heap} \times \mathsf{Heap} \\
h & \in & \mathsf{PropMap} = \mathsf{Prop} \xrightarrow{fin} \mathsf{Value} \\
\mathcal{L} & ::= & \square \mid \mathtt{let}^L\ x = s\ \mathtt{in}\ \square \mid \natural^L \square
\end{array}
$$

$$
\begin{array}{lll}
\textsc{SDem} & \mathcal{H}, \natural^L e & \longrightarrow \mathcal{H}^{\natural L}, e \\
\textsc{SApp} & \mathcal{H}, (\mathtt{rec}\, f(x).e)(v) & \longrightarrow \mathcal{H}, e\{f \Rightarrow \mathtt{rec}\, f(x).e\}\{x \Rightarrow v\} \\
\textsc{SLet} & \mathcal{H}, \mathtt{let}^L\ x = v\ \mathtt{in}\ e & \longrightarrow \mathcal{H}, e\{x \Rightarrow v\} \\
\textsc{SNew} & H, H_0, \mathtt{new}^\ell & \longrightarrow H, H_0\{(\ell, i) \mapsto \{\}\}, (@\ell, i) \\
 & & \quad \text{if } \mathrm{dom}(H_0) \cap (\{\ell\} \times \mathbf{Z}) = \emptyset \\
 & & \quad \text{and } (\ell, i) \notin \mathrm{dom}(H) \\
\textsc{SRd} & \mathcal{H}, (q\ell, i).a & \longrightarrow \mathcal{H}, \mathcal{H}(q\ell, i)\$a \\
\textsc{SWrt} & \mathcal{H}, (q\ell, i).a := v & \longrightarrow \mathcal{H}\{(q\ell, i)(a) \mapsto v\}, \mathtt{udf} \\
\textsc{SIfT} & \mathcal{H}, \mathtt{if}\ v\ e_1\ e_2 & \longrightarrow \mathcal{H}, e_1 \text{ if } v \neq \mathtt{udf} \\
\textsc{SIfF} & \mathcal{H}, \mathtt{if}\ \mathtt{udf}\ e_1\ e_2 & \longrightarrow \mathcal{H}, e_2 \\
\textsc{SLet}' & \dfrac{\mathcal{H}, s \longrightarrow \mathcal{H}', \mathcal{L}[v]}{\mathcal{H}, \mathtt{let}^L\ x = s\ \mathtt{in}\ e'' \longrightarrow \mathcal{H}', \mathcal{L}[\mathtt{let}^L\ x = v\ \mathtt{in}\ e'']} &
\end{array}
$$

Fig. 5. Instrumented small-step operational semantics.

a finite map from property names to values. To update property `a` of the object at address (l, i), we write concisely:

$$H\{(l,i)(a) \mapsto v\} := H\{(l,i) \mapsto H(l,i)\{a \mapsto v\}\}$$

Reduction of function application and of the `let` expression are standard. The `new` expression selects a new, unused memory location for ℓ and stores an empty record in the heap. The read expression extracts the property map for the argument location and performs a property access on it. Thus, the result of a read expression may be `udf`. The write expression updates the property map at address (ℓ, i).

The final context rule for `let` expressions forces the header of the let to be evaluated first. If beta reduction yields a value that is wrapped in an $\mathcal{L}$ context, then the rule reestablishes A-normal form by swapping the `let` with the context.

This semantics is close to existing semantics for object-based languages with premethods. However, it is not suitable for proving the soundness of the recency-aware type system, because the operational model does not distinguish between the most recently allocated object of a creation site and the objects previously allocated at the same site.

It is trivial that we can write down for each program, which is evaluated with the standard small step operational semantics, a program that behaves the same, but is evaluated with the instrumented version of the semantics. You must only wrap bodies of let expressions which mask expressions, when the right hand side of the let expression is a new expression.

Fig. 6 defines the non standard substitution with demotion. Consider the paper for explanation why the modification is necessary.

$$
\begin{array}{lcl}
(\mathtt{let}^L\ y = s\ \mathtt{in}\ e) & = & \mathtt{let}^L\ y = s\{x \Mapsto v\}\ \mathtt{in} \\
\quad \{x \Mapsto v\} & & \quad (e\{x \Mapsto v^{\natural L}\}) \\
y\{x \Mapsto v\} & = & \begin{cases} y & \text{if } x \neq y \\ v & \text{if } x = y \end{cases} \\
(\mathtt{rec}\, f(z).e)\{x \Mapsto v\} & = & \begin{cases} \mathtt{rec}\, f(z).e & \text{if } x \in \{z, f\} \\ \mathtt{rec}\, f(z).(e\{x \Mapsto v^{\natural}\}) & \text{if } x \notin \{z, f\} \end{cases} \\
\mathtt{udf}\{x \Mapsto v\} & = & \mathtt{udf} \\
(q\ell, i)\{x \Mapsto v\} & = & (q\ell, i) \\
(v_1(v_2))\{x \Mapsto v\} & = & v_1\{x \Mapsto v\}(v_2\{x \Mapsto v\}) \\
\mathtt{new}^\ell\{x \Mapsto v\} & = & \mathtt{new}^\ell \\
(v_1.a)\{x \Mapsto v\} & = & v_1\{x \Mapsto v\}.a \\
(v_1.a := v_2)\{x \Mapsto v\} & = & v_1\{x \Mapsto v\}.a := (v_2\{x \Mapsto v\}) \\
(\natural^L e)\{x \Mapsto v\} & = & \natural^L(e\{x \Mapsto v^{\natural L}\})
\end{array}
$$

Fig. 6. Substitution with demotion.

$$
\begin{array}{rcll}
\mathsf{Type} \ni t & ::= & \mathtt{obj}(p) \mid (\Sigma, t) \overset{L,A}{\to} (\Sigma, t) \mid \top \mid \mathtt{udf} & \\
p & ::= & \tilde{}L \mid @L & \text{with } |L| \geq 1 \\
r & ::= & \{\} \mid r\{a \mapsto v\} & \\
\mathsf{SummEnv} \ni \Omega & ::= & \emptyset \mid \Omega(\ell : r) & \\
\mathsf{SingEnv} \ni \Sigma & ::= & \emptyset \mid (\ell : r)\Sigma \mid (\Sigma \parallel \Sigma) & \\
\mathsf{TypeEnv} \ni \Gamma & ::= & \emptyset \mid \Gamma(x : t) &
\end{array}
$$

Fig. 7. Type syntax.

Lemma 1. *Let K be a configuration and $i \in \{1, 2, 3, 4, 5\}$. If $P_i(K)$ and $K \longrightarrow K'$, then $P_i(K')$.*

1. *$P_1(H, H_0, e) \equiv fv(e) = \emptyset$. The expression e is closed.*
2. *$P_2(H, H_0, e) \equiv (\forall \ell)|dom(H_0) \cap (\{\ell\} \times \boldsymbol{Z})| \leq 1$. For each abstract location there exists at most one object in the singleton heap.*
3. *$P_3(H, H_0, e)$: For all expressions of the form $\mathtt{rec} f(x).e_0$ that occur in the configuration, the body e_0 does not contain a singleton pointer $(@\ell, i)$.*
4. *$P_4(H, H_0, e)$: if $(@\ell, i)$ occurs in the configuration, then $(\ell, i) \in dom(H_0)$. A singleton pointer references an object in the singleton heap.*
5. *$P_5(H, H_0, e) \equiv dom(H) \cap dom(H_0) = \emptyset$. The domains of the summary heap and the singleton heap are disjoint.*

Proof. Assume $K = (H, H_0, e)$ and $K' = (H', H_0', e')$, $K \longrightarrow K'$ and for $i \in \{1, 2, 3, 5\} P_i(K)$. We show that $P_i(K')$ holds. For P_4 we need a more technical formulation. Please consider **INV-CLS** and lemma 9.
Case distinction over $\longrightarrow$.

– *Case* SDEM: If $e^\natural$ is closed, then e, too. Because $\mathrm{dom}(H_0') \subseteq \mathrm{dom}(H_0)$ $P_2(K')$ is trivial. The definition of $\natural^L H, H_0$ yields $P_3(K')$. P_5 is trivial, look at definition of $\natural^L$.

$$t <: t \qquad t <: \top \qquad \frac{L \subseteq L'}{\texttt{obj}(qL) <: \texttt{obj}(qL')}$$

$$\frac{t_1 <: t_1' \qquad t_2' <: t_2 \qquad L \subseteq L' \qquad A \subseteq A'}{(\Sigma_2, t_2) \overset{L,A}{\to} (\Sigma_1, t_1) <: (\Sigma_2, t_2') \overset{L',A'}{\to} (\Sigma_1, t_1')}$$

Fig. 8. Subtyping.

$$\textsc{Undefined}\quad \Omega, \Gamma \vdash_{\mathrm{v}} \texttt{udf} : \texttt{udf} \qquad \textsc{Object}\quad \Omega, \Gamma \vdash_{\mathrm{v}} (q\ell, i) : \texttt{obj}(q\ell) \qquad \textsc{Variable}\quad \frac{\Gamma(x) = t}{\Omega, \Gamma \vdash_{\mathrm{v}} x : t}$$

$$\textsc{Subsumption}\quad \frac{\Omega, \Gamma \vdash_{\mathrm{v}} v : t \qquad t <: t'}{\Omega, \Gamma \vdash_{\mathrm{v}} v : t'}$$

$$\textsc{Function}\quad \frac{\begin{array}{c}\mathrm{dom}(\Sigma) \cap L = \emptyset \\ L' \cup L'' \subseteq L \qquad \Gamma' = \Gamma \downarrow \mathrm{fv}(\texttt{rec}\, f(x).e) \qquad L' = \mathrm{Locs}(\Gamma') \qquad \Gamma'' = (\Gamma')^{\natural L'} \\ t_f = (\Sigma, t) \overset{L,A}{\to} (\Sigma', t') \qquad \Omega, \Sigma, \Gamma''(f : t_f)(x : t) \vdash_{\mathrm{e}} e : t' \Rightarrow L'', A, \Sigma', \Gamma'''\end{array}}{\Omega, \Gamma \vdash_{\mathrm{v}} \texttt{rec}\, f(x).e : t_f}$$

Fig. 9. Typing rules for values.

$$\begin{array}{ll}\mathrm{Locs}(\_ \overset{L,A}{\to} \_) & = \mathrm{Locs}(\top) = \mathrm{Locs}(\texttt{udf}) = \mathrm{Locs}(\emptyset) = \emptyset \\ \mathrm{Locs}(\texttt{obj}(@L)) & = L \\ \mathrm{Locs}(\texttt{obj}(\^{}L)) & = L \\ \mathrm{Locs}(\Gamma(x : t)) & = \mathrm{Locs}(\Gamma) \cup \mathrm{Locs}(t)\end{array}$$

Fig. 10. Locations in types and environments.

VALUE

$$\frac{\Omega, \Gamma \vdash_{\mathrm{v}} v : t}{\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} v : t \Rightarrow \emptyset, \emptyset, \Sigma, \Gamma}$$

LET

$$\frac{\begin{array}{c}\Sigma, \Gamma \vdash_c L_1 \qquad \Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} s_1 : t_1 \Rightarrow L_1, A_1, \Sigma_1, \Gamma_1 \\ \Omega, \Sigma_1, \Gamma_1(x : t_1) \vdash_{\mathrm{e}} e_2 : t_2 \Rightarrow L_2, A_2, \Sigma_2, \Gamma_2(x : t_1') \\ L = L_1 \cup L_2 \qquad A = A_1 \cup (A_2 - L_1)\end{array}}{\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} \mathtt{let}^{L_1}\ x = s_1\ \mathtt{in}\ e_2 : t_2 \Rightarrow L, A, \Sigma_2, \Gamma_2}$$

DEMOTE

$$\frac{\begin{array}{c}\Omega, \Sigma \vdash_{\mathrm{t}} \mathtt{obj}(@(L \cap \mathrm{dom}(\Sigma))) \lhd \mathtt{obj}(\tilde{}L) \\ \Sigma' = \Sigma^{\natural L} \upharpoonright L \qquad \Omega, \Sigma', \Gamma^{\natural L} \vdash_{\mathrm{e}} e : t \Rightarrow L', A, \Sigma'', \Gamma'' \qquad L \subseteq L' \qquad \Omega = \Omega^{\natural L}\end{array}}{\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} \natural^L e : t \Rightarrow L', A, \Sigma'', \Gamma''}$$

FUNCTION CALL

$$\frac{\begin{array}{c}\Sigma, \Gamma \vdash_c L \qquad \Sigma, A \vdash_S \Sigma_1, \Sigma_2 \\ \Sigma', A \vdash_S \Sigma_1', \Sigma_2 \qquad \Omega, \Gamma \vdash_{\mathrm{v}} v_2 : t_2 \qquad \Omega, \Gamma \vdash_{\mathrm{v}} v_1 : (\Sigma_1, t_2) \xrightarrow{L,A} (\Sigma_1', t_1)\end{array}}{\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} v_1(v_2) : t_1 \Rightarrow L, A, \Sigma', \Gamma}$$

NEW

$$\frac{\Sigma, \Gamma \vdash_c \{\ell\} \qquad \ell \in \mathrm{dom}(\Omega)}{\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} \mathtt{new}^{\ell} : \mathtt{obj}(@\ell) \Rightarrow \{\ell\}, \emptyset, \Sigma(\ell \mapsto \{\}), \Gamma}$$

READ

$$\frac{\Omega, \Gamma \vdash_{\mathrm{v}} v : \mathtt{obj}(p) \qquad \Omega, \Sigma \vdash_{\mathrm{r}} p.a : t \qquad A \vdash_a p}{\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} v.a : t \Rightarrow \emptyset, A, \Sigma, \Gamma}$$

WRITE

$$\frac{\Omega, \Gamma \vdash_{\mathrm{v}} v : \mathtt{obj}(p) \qquad \Omega, \Gamma \vdash_{\mathrm{v}} v' : t' \qquad \Omega, \Sigma \vdash_{\mathrm{w}} p.a := t' \Rightarrow \Sigma' \qquad A \vdash_a p}{\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} v.a := v' : \mathtt{udf} \Rightarrow \emptyset, A, \Sigma', \Gamma}$$

IF

$$\frac{\begin{array}{c}\Omega, \Gamma \vdash_{\mathrm{v}} v : t_v \\ \Sigma_1, \Sigma_2 \vdash_J \Sigma' \qquad \forall i(\Gamma_i <: \Gamma' \wedge t_i <: t \wedge \Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} e_i : t_i \Rightarrow L_i, A_i, \Sigma_i, \Gamma_i)\end{array}}{\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} \mathtt{if}\ v\ e_1\ e_2\ : t \Rightarrow L_1 \cup L_2, A_1 \cup A_2, \Sigma', \Gamma'}$$

Fig. 11. Typing rules for expressions.

$$\boxed{\Omega, \Sigma \vdash t \lhd t'} \qquad \frac{(\forall a \in \mathsf{Prop})\ \Omega, \Sigma \vdash_{\mathrm{t}} r(a) \lhd r'(a)}{\Omega, \Sigma \vdash_h r \lhd r'}$$

$$\frac{(\forall \ell \in L)\ \Omega, \Sigma \vdash_h \Sigma(\ell) \lhd \Omega(\ell)}{\Omega, \Sigma \vdash_{\mathrm{t}} \mathtt{obj}(@L) \lhd \mathtt{obj}(\tilde{}L')} \qquad \frac{t <: t'}{\Omega, \Sigma \vdash_{\mathrm{t}} t \lhd t'}$$

$$\boxed{\Omega, \Sigma \vdash_{\mathrm{r}} p.a : t} \qquad \frac{\Sigma(\ell)(a) = t}{\Omega, \Sigma \vdash_{\mathrm{r}} @\ell.a : t} \qquad \frac{\forall \ell \in L(\Omega, \Sigma \vdash_{\mathrm{r}} q\ell.a : t'_\ell \wedge t'_\ell <: t)}{\Omega, \Sigma \vdash_{\mathrm{r}} qL.a : t}$$

$$\frac{\Omega(\ell)(a) = t}{\Omega, \Sigma \vdash_{\mathrm{r}} \tilde{}\ell.a : t}$$

$$\boxed{\Omega, \Sigma \vdash_{\mathrm{w}} p.a := t \Rightarrow \Sigma'} \qquad \frac{(\forall \ell \in L)\ t <: \Omega(\ell)(a)}{\Omega, \Sigma \vdash_{\mathrm{w}} \tilde{}L.a := t \Rightarrow \Sigma}$$

$$\frac{\Omega, \Sigma \vdash_{\mathrm{w}}^{@} L.a := t \Rightarrow \Sigma', n \qquad n = 1}{\Omega, \Sigma \vdash_{\mathrm{w}} @L.a := t \Rightarrow \Sigma'} \qquad \boxed{\Sigma \vdash_{\mathrm{w}}^{@} p.a := t \Rightarrow \Sigma', n}$$

$$\emptyset \vdash_{\mathrm{w}}^{@} L.a := t \Rightarrow \emptyset, 0 \qquad \frac{\Sigma \vdash_{\mathrm{w}}^{@} L.a := t \Rightarrow \Sigma'', n \qquad \ell \in L}{\Sigma(\ell : r) \vdash_{\mathrm{w}}^{@} L.a := t \Rightarrow (\ell : r[a \mapsto t])\Sigma'', n + 1}$$

$$\frac{\Sigma \vdash_{\mathrm{w}}^{@} L.a := t \Rightarrow \Sigma, n \qquad \ell \notin L}{\Sigma(\ell : r) \vdash_{\mathrm{w}}^{@} L.a := t \Rightarrow \Sigma, n}$$

$$\frac{\Sigma_1 \vdash_{\mathrm{w}}^{@} L.a := t \Rightarrow \Sigma'_1, n_1 \qquad \Sigma_2 \vdash_{\mathrm{w}}^{@} L.a := t \Rightarrow \Sigma'_2, n_2}{(\Sigma_1 \parallel \Sigma_2) \vdash_{\mathrm{w}}^{@} L.a := t \Rightarrow (\Sigma'_1 \parallel \Sigma'_2), \max(n_1, n_2)}$$

$$\boxed{\Sigma, \Gamma \vdash_c L} \qquad \frac{\Gamma = \Gamma^{\natural L} \qquad \Sigma = \Sigma^{\natural L} \qquad \mathrm{dom}(\Sigma) \cap L = \emptyset}{\Sigma, \Gamma \vdash_c L}$$

$$\boxed{A \vdash_a p} \qquad A \cup L \vdash_a @L \qquad A \vdash_a \tilde{}L$$

Fig. 12. Rules for flow, reading, and writing to the heap.

$$\frac{\Omega, \Sigma \Vdash H, H_0 \qquad \Omega, \Sigma, \emptyset \vdash_{\mathrm{e}} e : t \Rightarrow L, A, \Sigma', \emptyset}{\Omega, \Sigma \Vdash_e H, H_0, e : t \Rightarrow L, A, \Sigma'} \qquad \emptyset \Vdash_\Sigma H_0$$

$$\frac{\exists! i\ (\ell, i) \in \mathrm{dom}(H_0) \wedge \Omega, \Sigma \Vdash_o H_0(\ell, i) : r \qquad \Sigma \Vdash_\Sigma H_0}{(l : r)\Sigma \Vdash_\Sigma H_0}$$

$$\frac{j, k \in \{1, 2\} \qquad j \neq k \qquad \Sigma_j \Vdash_\Sigma H \qquad \forall \ell \in \mathrm{dom}(\Sigma_k)\ \forall i\ (l, i) \notin \mathrm{dom}(H)}{(\Sigma_1 \parallel \Sigma_2) \Vdash_\Sigma H}$$

$$\frac{\Sigma \Vdash_\Sigma H_0 \qquad \forall (\ell, i) \in \mathrm{dom}(H)\ \ell \in \mathrm{dom}(\Omega) \wedge \Omega, \Sigma \Vdash_o H(\ell, i) : \Omega(\ell)}{\Omega, \Sigma \Vdash H, H_0}$$

$$\frac{(\forall a \in \mathrm{dom}(h))\ a \in \mathrm{dom}(r) \wedge \Omega, \emptyset \vdash_{\mathrm{v}} h(a) : r(a)}{\Omega, \Sigma \Vdash_o h : r}$$

Fig. 13. Typing of heaps and configurations.

- *Case* SApp: Since e is closed, the substitution yields a closed expression. All other cases are trivial because the heaps do not change.
- *Case* SLet: Analogous to SApp.
- *Case* SNew: The condition of SNew ensures P_2 and P_5, while the closeness is not affected.
- *Case* SRd,SWrt,SIfT,SIfF: trivial.
- *Case* SLet′: by induction

End case distinction over $\longrightarrow$.

2.2 Static Semantics

The following list summaries the differences between the paper and the technical report:

- Object types may have the from $\texttt{obj}(qL)$, even for $q = @$ (§1.1).
- The grammar defining the singleton environment supports with $\Sigma \parallel \Sigma$ heap alternatives (§1.1).
- The function type is enriched with the access effect A. (§1.1).
- The subtyping relation allows subtyping over $\texttt{obj}(\tilde{L}) <: \texttt{obj}(\tilde{L})$.
- In Fig. 11 every rule has the form $\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} e : t \Rightarrow L, A, \Sigma, \Gamma$.
- In Fig. 11 the rule If is added. The judgment from Fig. 1 is used to introduce heap alternative, if that is possible.
- The rule Function Call supports splitting of the most recent environment. Hence in this we make use of the auxiliary judgment from Fig. 1. The relation splits the most recent heap Σ into two parts, Σ_1 and Σ_2, such that Σ_1 is passed to the function while Σ_2 stays outside. After the function returns, the modified most recent heap Σ_1' is joined with Σ_2 and yields a modified

most recent heap Σ', which is the modified most recent heap of the function call expression.
- The rule READ and WRITE ensures with the additional condition $A \vdash_a p$, that the access effects are computed in the sound way.
- The judgment $\vdash_{\mathrm{w}}$ is modified to support environment alternatives. It ensures that for each possible alternative at most one object is affected. More explanation how it works are available in the soundness proof.

For the proves we assume that we have a universe $\mathcal{U}$ and a monotone function $F : \mathcal{P}(\mathcal{U} \times \mathcal{U}) \to \mathcal{P}(\mathcal{U} \times \mathcal{U})$.

Definition 1. *νF denotes the greatest fixed point of F.*

Definition 2. *For a monotone function F the set $X \subseteq \mathcal{U}$ is F-consistent if $X \subseteq F(X)$.*

Lemma 2. *The union of all F-consistent sets is the greatest fixed point of F.*

Definition 3. *For a universe $\mathcal{U}$ a relation $R \subseteq \mathcal{U} \times \mathcal{U}$ is transitive if R is closed under the monotone function $TR(R) = \{(x, y) \mid \exists z \in \mathcal{U} : (x, z) \in R \wedge (z, y) \in R\}$ – i.e., if $TR(R) \subseteq R$.*

Lemma 3. *The subtyping relation $<:$ is reflexive and transitive.*

Proof. Notice that types are defined coinductive. Hence an induction prove is not valid.

Reflexivity per definition

Transitivity Let S be the monotone function presented in Fig. 8. Since νS is a fixed point, $\nu S = S(\nu S)$. Hence $TR(\nu S) = TR(S(\nu S))$. Under the assumption

$$TR(S(\nu S)) \subseteq S(TR(\nu S)) \quad , \tag{5}$$

we can conclude $TR(\nu S) \subseteq S(TR(\nu S))$. Hence $TR(\nu S)$ is S-consistent and Lemma 2 implies $TR(\nu S) \subseteq \nu S$. Hence νS is transitive.

It remains to prove equation (5). We show the stronger fact that for all $R \subseteq \mathcal{U} \times \mathcal{U}$ it holds $TR(S(R)) \subseteq S(TR(R))$.

Let $(t_1, t_2) \in TR(S(R))$. The definition of transitivity yields, that there exists a type t' such that $(t_1, t') \in S(R)$ and $(t', t_2) \in S(R)$.

Now we prove $(t_1, t_2) \in S(TR(R))$.

Case distinction over shape of t'.

- *Case* $t' = \top$: Since $(t', t_2) \in S(R)$, the definition of S implies $t_2 = \top$. Hence $(t_1, t_2) = (t_1, \top) \in S(Q)$ for all Q and therefore $(t_1, t_2) \in S(TR(R))$.
- *Case* $t' = \mathtt{obj}(p)$: Since $(t', t_2) \in S(R)$ the definition of S implies $t_2 = \top$, $t_2 = \mathtt{obj}(p)$ or $t_2 = \mathtt{obj}(p')$.

  Case distinction over shape of t_2.
  - *Case* $t_2 = \top$: This is analogous to the case $t' = \top$.

- *Case* $t_2 = t'$: Then $(t_1, t_2) \in S(R)$. The definition of TR yields $(t_1, t_2) \in TR(S(R))$ by setting $z = y = t_2$ and $x = t_1$.
- *Case* $t_2 = \mathtt{obj}(p')$: The definition of S yields $p = qL$ and $p' = qL'$ with $L \subseteq L'$. Further, it holds $t_1 = \mathtt{obj}(p'')$ and we get $p'' = qL''$ with $L'' \subseteq L$. Hence $L'' \subseteq L'$. By the definition of S $(t_1, t_2) \in S(Q)$ for all Q, hence $(t_1, t_2) \in S(TR(R))$.

End case distinction over shape of t_2.

– *Case* $t' = \mathtt{udf}$: Analog to the object case.

– *Case* $t' = (\Sigma_p, t'_p) \overset{L', A'}{\to} (\Sigma_r, t'_r)$: The case $t_2 = \top$ is analogous to the case $t' = \top$.
Otherwise $(t', t_2) \in S(R)$ implies $t_2 = (\Sigma_p, t^2_p) \overset{L_2, A_2}{\to} (\Sigma_r, t^2_r)$ with $(t^2_p, t'_p) \in R$, $(t'_r, t^2_r) \in R$, $L' \subseteq L_2$ and $A' \subseteq A_2$.
Similarly, $t_1 = (\Sigma_p, t^1_p) \overset{A_1, L_1}{\to} (\Sigma_r, t^1_r)$ with $(t'_p, t^1_p) \in R$, $(t^1_r, t'_r) \in R$, $L_1 \subseteq L'$ and $L_2 \subseteq A'$.
Hence $L_1 \subseteq L_2$ and $A_1 \subseteq A_2$. The definition of TR implies $(t^1_r, t^2_r) \in TR(R)$ and $(t^2_p, t^1_p) \in TR(R)$. The definition of S implies $(t_1, t_2) \in S(TR(R))$.

End case distinction over shape of t'.

Lemma 4. *If* $t <: t'$ *holds, then* $\forall \Omega, \Sigma : \Omega, \Sigma \vdash_{\mathrm{t}} t \lhd t'$

Proof. Trivial, look at the defintion of $\Omega, \Sigma \vdash_{\mathrm{t}} t \lhd t$.

Lemma 5. *If* $\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} e : t \Rightarrow L, A, \Sigma', \Gamma'$ *then* $\Gamma' = \Gamma^{\natural L}$.

Proof. Induction on the typing derivation.
Case distinction over the typing judgment.

– *Case* Demote: Since $(\Gamma^{\natural L})^{\natural L} = \Gamma^{\natural L}$, $\Gamma' = \Gamma^{\natural L}$ follows by induction after inversion of Demote.
– *Case* Value: immediate
– *Case* Let: Induction yields that $\Gamma_1 = \Gamma^{\natural L_1}$ and $\Gamma_2(x : t'_1) = (\Gamma_1(x : t_1))^{\natural L_2}$. Hence, $\Gamma_2 = \Gamma^{\natural L_1 \cup L_2}$.
– *Case* Function Call, New, Read, Write: immediate

End case distinction over the typing judgment.

Lemma 6. *If* $\Omega, \Gamma \vdash_{\mathrm{v}} v : t$ *and* $\Gamma' = \Gamma^{\natural L}$, *then* $\Omega, \Gamma' \vdash_{\mathrm{v}} v^{\natural L} : t^{\natural L}$.

Proof. Case distinction over the typing judgment $\vdash_{\mathrm{v}}$.

– *Case* Undefined, Variable: immediate
– *Case* Object: If $q = \tilde{}$ or $\ell \notin L$, then the result is immediate because $(q\ell, i)^{\natural L} = (q\ell, i)$ and $\mathtt{obj}(q\ell)^{\natural L} = \mathtt{obj}(q\ell)$.
If $q = @$ and $\ell \in L$, then the result holds, too, because $(@\ell, i)^{\natural L} = (\tilde{}\ell, i)$ and $\mathtt{obj}(@\ell)^{\natural L} = \mathtt{obj}(\tilde{}\ell)$.
– *Case* Subsumption: This case requires that $t <: t'$ implies $t^{\natural L} <: t'^{\natural L}$. But the latter is an easy consequence of the definition of $<:$.

– *Case* FUNCTION: The typing remains the same

End case distinction over the typing judgment $\vdash_{\mathrm{v}}$.

Lemma 7. *If* $\Omega, \Gamma \vdash_{\mathrm{v}} \lambda(y, x).e : t$ *then* $Locs(t) = \emptyset$.

Proof. Trivial, look at definition of Locs.

Subsitution

Lemma 8 (Substitution). *For a closed* v *and*

$$\begin{aligned} \Omega, \Sigma, \Gamma(x : t_x) \vdash_{\mathrm{e}} e : t_0 \Rightarrow L, A, \Sigma', \Gamma', \\ \Omega, \Gamma \vdash_{\mathrm{v}} v : t_x, \end{aligned} \tag{6}$$

it holds that

$$\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} e\{x \Mapsto v\} : t_0 \Rightarrow L, A, \Sigma', \Gamma' \quad .$$

Closeness of v is needed. Otherwise substitution may change the set of free variables of a lambda abstraction, which in turn may enlarge the set of exact references passed into the body of the lambda abstraction, and thus enlarge the set L' in the typing rule for abstraction (FUNCTION). This set L' can change the type by demoting references to be inexact.

Proof. by induction on expressions.
Case distinction over the structure of e.

– *Case* $e = v_0$: It must be that $L = \emptyset$, $A = \emptyset$, $\Sigma = \Sigma'$, and $\Gamma(x : t_x) = \Gamma'$. Inversion of VALUE yields

$$\Omega, \Gamma(x : t_x) \vdash_{\mathrm{v}} v_0 : t_0 \tag{7}$$

There are the following cases for values: variable, abstraction, undefined, or a pointer combined by the SUBSUMPTION rule.[1]
Case distinction over the structure of values.

- *Case* $v_0 = y$: Inversion of (7) using VARIABLE yields

$$\Gamma(x : t_x)(y) = t_0' \text{ with } t_0' <: t_0 \tag{8}$$

If $x = y$ then $t_0' = t_x$, and $t_x <: t_0$. VALUE, SUBSUMPTION and (6) yields

$$\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} e\{x \Mapsto v\} : t_0 \Rightarrow \emptyset, \emptyset, \Sigma, \Gamma \tag{9}$$

If $x \neq y$ then nothing happens. The type stays the same: t_0.

[1] Please notice that for the proves we use the notation $\lambda y.e$ instead of $\texttt{rec}\, f(x).e$. It doesn't matter, the difference is only hat with the $\texttt{rec}\, f(x).e$ the scoping is a little bit modified, and sometimes later on, we have to apply the substitution lemma twice instead of once.

- *Case* $v_0 = \lambda y.e_l$: If $x \notin \mathrm{fv}(\lambda y.e_l)$, then the desired result is immediate because v_0 is then not affected due to the substitution.
  Now assume that $x \in \mathrm{fv}(\lambda y.e_l)$. Hence $x \in \mathrm{fv}(e_l)$. First we define two shortcuts: Let x range over all variables, t_x over all types and Γ over all type environments. Then Γ^x is a shortcut for $\Gamma(x : t_x)$ and Γ_x for $\Gamma \downarrow \{y \mid y \in \mathrm{dom}(\Gamma), y \neq x\}$.
  Then inversion of (7) using FUNCTION yields

$$t_0 = (\Sigma_2, t_y) \overset{L,A}{\to} (\Sigma_1, t_r) \tag{10}$$
$$\mathrm{dom}(\Sigma_2) \cap L = \emptyset \tag{11}$$
$$L' \cup L'' \subseteq L \tag{12}$$
$$\Gamma' = \Gamma(x : t_x) \downarrow \mathrm{fv}(\lambda(y, z).e_l)$$
$$L' = \mathrm{Locs}(\Gamma') \tag{13}$$
$$\Gamma'' = (\Gamma')^{\natural L'}$$
$$\Omega, \Sigma_2, \Gamma''^y \vdash_{\mathrm{e}} e_l : t_r \Rightarrow L'', \Sigma_1, \Gamma''' \quad . \tag{14}$$

  $x \in \mathrm{dom}(\Gamma''^y)$ holds because $x \in \mathrm{fv}(e_l)$. That's why Γ'' from equation (14) contains an assignment for x, and we can write $\Gamma'' = \Gamma''_x(x : t_x)$. Rewriting judgment (14) to highlight on the important variable x yields:

$$\Omega, \Sigma_2, \Gamma''^y_x(x : t_x) \vdash_{\mathrm{e}} e_l : t_r \Rightarrow L'', \Sigma_1, \Gamma'''_x(x : t'_x)$$

  Next we show

$$\Omega, \Gamma''^y_x \vdash_{\mathrm{v}} v^\natural : t^\natural_x \quad . \tag{15}$$

  To figure out (15) we make use of (6) and Lemma 6. We are able to change the Γ because v is closed. Notice that if v is closed, then $v^\natural$ is closed, too. So we can apply induction on e_l and $v^\natural$, which yields

$$\Omega, \Sigma_2, \Gamma''^y_x \vdash_{\mathrm{e}} e_l\{x \Mapsto v^\natural\} : t_r \Rightarrow L', A, \Sigma_1, \Gamma'''_x$$

  Thus preconditions for FUNCTION are fulfilled and yields,

$$\Omega, \Gamma \vdash_{\mathrm{v}} (\lambda y.e_l)\{x \Mapsto v\} : (\Sigma_2, t_y) \overset{L,A}{\to} (\Sigma_1, t_r) \tag{16}$$

  The rule VALUE yields the desired

$$\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} (\lambda y.e_l)\{x \Mapsto v\} : (\Sigma_2, t_y) \overset{L,A}{\to} (\Sigma_1, t_r) \Rightarrow \emptyset, \emptyset, \Sigma, \Gamma \quad .$$

- *Case* $v_0 = (q\ell, i)$.: The expression is not affected by the substitution.
- *Case* $v_0 =$ `udf`.: It isn't affected by the substitution.

End case distinction over the structure of values.

– *Case* $e = \texttt{let}^{L_1}\ y = e_1\ \texttt{in}\ e_2$: Suppose that

$$\begin{aligned}\Omega, \Sigma, \Gamma(x : t_x) \vdash_{\mathrm{e}} \texttt{let}^{L_1}\ y = s\ \texttt{in}\ e : t_0 \\ \Rightarrow L, A, \Sigma_0, \Gamma_0(x : t_{x0})\end{aligned} \tag{17}$$

and $\Omega, \Gamma \vdash_{\mathrm{v}} v : t_x$.
Inversion of (17) yields

$$\Sigma, \Gamma \vdash_c L$$
$$L = L_1 \cup L_2$$
$$A = A_1 \cup (A_2 - L_1)$$
$$\Omega, \Sigma, \Gamma(x : t_x) \vdash_{\mathrm{e}} s : t_1 \Rightarrow L_1, A_1, \Sigma_1, \Gamma_1(x : t_{x1}) \tag{18}$$
$$\begin{aligned}\Omega, \Sigma_1, \Gamma_1(x : t_{x1})(y : t_1) \vdash_{\mathrm{e}} e : t_2 \\ \Rightarrow L_2, A_2, \Sigma_2, \Gamma_2(x : t_{x0})(y : t_1')\end{aligned} \tag{19}$$

Induction is applicable to (18) and yields

$$\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} s\{x \Mapsto v\} : t_1 \Rightarrow L_1, A_1, \Sigma_1, \Gamma_1 \tag{20}$$

By Lemma 5 applied to (18), it holds that

$$\Gamma_1(x : t_{x1}) = (\Gamma(x : t_x))^{\natural L_1}$$

such that $t_{x1} = t_x^{\natural L_1}$. By Lemma 6, it holds that

$$\Omega, \Gamma_1 \vdash_{\mathrm{v}} v^{\natural L_1} : t_{x1} \tag{21}$$

Induction is applicable to (19) and (21) and yields that

$$\begin{aligned}\Omega, \Sigma_1, \Gamma_1(y : t_1) \vdash_{\mathrm{e}} e\{x \Mapsto v^{\natural L_1}\} : t_2 \\ \Rightarrow L_2, A_2, \Sigma_2, \Gamma_2(y : t_1')\end{aligned} \tag{22}$$

Finally, the let rule is applicable to (20) and (22) to yield the desired

$$\begin{aligned}\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} \texttt{let}^{L_1}\ y = s\{x \Mapsto v\}\ \texttt{in}\ e\{x \Mapsto v^{\natural L_1}\} : t_2 \\ \Rightarrow L_1 \cup L_2, A_1 \cup (A_2 - L_1), \Sigma_2, \Gamma_2\end{aligned}$$

that is

$$\begin{aligned}\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} (\texttt{let}^{L_1}\ y = s\ \texttt{in}\ e)\{x \Mapsto v\} : t_2 \\ \Rightarrow L, A, \Sigma_2, \Gamma_2\end{aligned}$$

– *Case* $e = v_1(v_2)$: Immediate by induction.
– *Case* $e = \texttt{new}^{\ell}$: Immediate.
– *Case* $e = v_0.a$: Immediate by induction.
– *Case* $e = v_{01}.a := v_{02}$: Immediate by induction.
– *Case* $e = \natural^L e_l$: Immediate by induction.
– *Case* $\texttt{if}\ v\ e_1\ e_2$: Immediate by induction.

End case distinction over the structure of e.

Invariants The invariant **INV-CLS** states closedness of a configuration with respect to the heap: Any reference contained in one of the heaps or in the expression is defined in one of the heaps. An auxiliary definition simplifies the statement of the invariant. Define $(q\ell, i) \in H, H_0$ by

- $(\tilde{}\ell, i) \in H, H_0$ iff $(\ell, i) \in \mathrm{dom}(H) \cup \mathrm{dom}(H_0)$ and
- $(@\ell, i) \in H, H_0$ iff $(\ell, i) \in \mathrm{dom}(H_0)$.

While leading to a provable invariant, this relation does not yield sufficiently precise information about imprecise references: A statement is needed that states when an imprecise reference can definitely be found in the summary heap! Such a statement can be formulated using the following notion of *unblocked contexts*. The Invariant only holds for type correct expressions. Additional annotations on the lambda expressions simplifies the definition of unblocked contexts. We annotate a lambda expression in the same way as it is already done by let expressions.

An unblocked context $\mathcal{U}^\ell$ for an imprecise reference $(\tilde{}\ell, i)$ is defined for all L such that $\ell \notin L$ and arbitrary L' as

$$\begin{aligned}\mathcal{U}^\ell ::= {} & \Box.a \mid \Box.a := v \\ & \mid \natural^L \mathcal{U}^\ell \mid \mathtt{let}^{L'}\ x = \mathcal{U}^\ell\ \mathtt{in}\ e \mid \mathtt{let}^{L}\ x = e\ \mathtt{in}\ \mathcal{U}^\ell\end{aligned}$$

Please notice that for the invariant we are only interested in programs that was correctly preprocessed. This implies that each let expression with a new on the right hand side and each function call is surrounded by a demotation.

INV-CLS is fulfilled by a configuration H, H_0, e if:

1. e is closed and preprocessed
2. If $(q\ell, i)$ occurs in e, then $(q\ell, i) \in H, H_0$.
3. If $e = \mathcal{U}^\ell[(\tilde{}\ell, i)]$, then $(\ell, i) \in \mathrm{dom}(H)$.
4. For all $(\ell, i) \in \mathrm{dom}(H) \cup \mathrm{dom}(H_0)$, if $h = (H \cup H_0)(\ell, i)$ then, for all $a \in \mathrm{dom}(h)$,
   (a) if $(q\ell', i')$ occurs in $h(a)$, then $(q\ell', i') \in H, H_0$ and
   (b) if $h(a) = (\tilde{}\ell', i')$, then $(\ell', i') \in \mathrm{dom}(H)$.

Lemma 9. *If H, H_0, e fulfills **INV-CLS** and*

$$\begin{aligned}& H, H_0, e \longrightarrow^{1,2} H', H_0', e' \\ & \quad \Omega, \Sigma \Vdash_e H, H_0, e : t \Rightarrow L, A, \Sigma'\end{aligned}$$

*then H', H_0', e' fulfills **INV-CLS**.*

The relation $\longrightarrow^{1,2}$ is similar to $\longrightarrow$ but evaluates the body of a mask expression together with the mask expression itself in one step and do not execute a let or a function call without a demotation expression around it. The relation is defined in figure 14.

Proof. Case distinction over the reduction $\longrightarrow^{1,2}$.

$$\frac{H, H_0, \natural^L(\lambda x.e)(v) \longrightarrow H', H_0', (\lambda x.e)(v') \qquad H', H_0', (\lambda x.e)(v') \longrightarrow H'', H_0'', e''}{H, H_0, \natural^L(\lambda x.e)(v) \longrightarrow^{1,2} H'', H_0'', e''}$$

$$\frac{\begin{array}{c} H, H_0, \natural^L(\texttt{let}^L\ x = s\ \texttt{in}\ e) \longrightarrow H', H_0', \texttt{let}^L\ x = s'\ \texttt{in}\ e' \\ H', H_0', \texttt{let}^L\ x = s'\ \texttt{in}\ e' \longrightarrow H'', H_0'', e'' \end{array}}{H, H_0, \natural^L(\texttt{let}\ x = s\ \texttt{in}\ e) \longrightarrow^{1,2} H'', H_0'', e''}$$

$$\frac{H, H_0, e \longrightarrow H', H_0', e' \qquad e \neq (\lambda(x).e_0)(v) \qquad e \neq \texttt{let}\ x = s\ \texttt{in}\ e_0}{H, H_0, e \longrightarrow^{1,2} H', H_0', e'}$$

Fig. 14. Modified Semantics

– *Case* SDem::

$$H, H_0, \natural^L e \longrightarrow (H, H_0)^{\natural L}, e$$

Thus with $H_L = H_0^{\natural L} \downarrow \{(\ell, i) \mid \ell \in L, i \in \mathbf{Z}\}$, $H' = H^{\natural L} \cup H_L$ and $H_0' = H_0^{\natural L} \backslash H_L$.
Item 2 holds because e has not changed and $\mathrm{dom}(H') \cup \mathrm{dom}(H_0') = \mathrm{dom}(H) \cup \mathrm{dom}(H_0)$.
Item 3 holds as follows. Suppose that $e = \mathcal{U}^\ell[(\tilde{}\ell, i)]$. If $\ell \notin L$, then let $\mathcal{U}_1^\ell = \natural^L \mathcal{U}^\ell$ is an unblocking context that works for $\natural^L e$. If $\ell \in L$, then item 2 yields that $(\ell, i) \in \mathrm{dom}(H') \cup \mathrm{dom}(H_0')$ and because H_L has transferred all L-references from H_0 to H' it must be that $(\ell, i) \in \mathrm{dom}(H')$.
For item 4, take some $(\ell, i) \in \mathrm{dom}(H) \cup \mathrm{dom}(H_0) = \mathrm{dom}(H') \cup \mathrm{dom}(H_0')$, let $h = (H \cup H_0)(\ell, i)$ and $h' = (H' \cup H_0')(\ell, i)$. As $\mathrm{dom}(h) = \mathrm{dom}(h')$, pick some $a \in \mathrm{dom}(h)$.
If $(@\ell', i')$ occurs in $h'(a)$, then $\ell' \notin L$ and $(\ell', i') \in \mathrm{dom}(H_0) \cap \mathrm{dom}(H_0')$ proving one part of the claim. If $(\tilde{}\ell', i')$ occurs in $h'(a)$, then $(\ell', i') \in \mathrm{dom}(H') \cup \mathrm{dom}(H_0') = \mathrm{dom}(H) \cup \mathrm{dom}(H_0)$. Thus, item 4a holds.
If $h'(a) = (\tilde{}\ell', i')$, then there are two cases. Either $\ell' \notin L$, in which case $(\ell', i') \in \mathrm{dom}(H) \subseteq \mathrm{dom}(H')$. Or $\ell' \in L$ in which case $(\ell', i') \in \mathrm{dom}(H_L) \subseteq \mathrm{dom}(H_0)$ so that $(\ell', i') \in \mathrm{dom}(H_L) \subseteq \mathrm{dom}(H')$. Thus, item 4b also holds.
Now the relation $\longrightarrow^{1,2}$ executes the body of the demotation expression. There are two cases:

- *Case* SApp::

$$H, H_0, (\lambda x.e)(v) \longrightarrow H, H_0, e\{x \Mapsto v\}$$

Item 2 holds because substitution only changes precise to imprecise references, which is covered by the definition of $(q\ell, i) \in H, H_0$.
To prove item 3 we make use of our assumption and the information that a mask expression was executed. Since that we find a contradiction for the case that $(l, i) \in \mathrm{dom}(H_0)$ and conclude from $(l, i) \in H, H_0$ the desired $(l, i) \in H_0$.
Item 4 holds trivially because H, H_0 does not change.

- *Case* SLet::

$$H, H_0, \texttt{let}^L\ x = v\ \texttt{in}\ e \longrightarrow H, H_0, e\{x \mapsto v\}$$

Item 2 holds because substitution only changes precise references to imprecise ones.
To prove item 3 , assume $(\tilde{l}, i)$ is one reference that is not protected after reduction and that was protected before. The definition of protected references gives us two cases.
If $\ell \notin L$, then $\mathcal{U}_1^\ell = \texttt{let}^L\ x = v\ \texttt{in}\ \mathcal{U}^\ell$ so that $(\ell, i) \in \mathrm{dom}(H)$.
The other case is $l \in L$. Inversion of the typing judgment yields that the expression v is typeable. Because values does not have effects, the set L is empty. So $l \in L$ is a contradiction.
Item 4 holds trivially.

- *Case* SLet′: This case is easy by induction.

– *Case* SNew:: trivial
– *Case* SRd: Item 2 and item 3 holds because of item 4. The other ones holds, because nothing happens with the heap.
– *Case* SWrt: The change in the heap and the expression `udf` fulfills **INV-CLS**.
– *Case* SIfT, SIfF: Nothing happens

End case distinction over the reduction $\longrightarrow^{1,2}$.

Please notice that for all programs the programmer can write down there is no important difference between $\longrightarrow$ and $\longrightarrow^{1,2}$, since the programmer is not allowed to write down references directly into the program.

Progress

Lemma 10 (Progress). *For a configuration H, H_0, e with a closed e that fulfills **INV-CLS** together with*

$$\Omega, \Sigma \Vdash_e H, H_0, e : t \Rightarrow L, A, \Sigma' \tag{23}$$

either e is a value or there exists H', H_0', e' such that

$$H, H_0, e \longrightarrow H', H_0', e'$$

Proof. Inversion of (23) yields:

$$\begin{aligned} &\Omega, \Sigma, \emptyset \vdash_{\mathrm{e}} e : t \Rightarrow L, A, \Sigma', \emptyset \\ &\Omega, \Sigma \Vdash H, H_0 \end{aligned} \tag{24}$$

By induction over the structure of e:
Case distinction over structure of e.

– *Case* $e = v_1(v_2)$: Inversion of (24) yields amongst others

$$\begin{aligned} &\Omega, \Gamma \vdash_{\mathrm{v}} v_1 : (\Sigma, t_0 \times t_2) \overset{L,A}{\to} (\Sigma_1, t_1) \\ &\Omega, \Gamma \vdash_{\mathrm{v}} v_2 : t_2 \\ &\quad \Gamma = \emptyset \end{aligned}$$

The first line implies that v_1 is a lambda expression (notice that e is closed). That's why SApp is applicable.

– *Case* $e = \mathtt{new}^l$: trivial

– *Case* $\natural^L e$: trivial

– *Case* $v.a$: Inversion of (24) yields

$$\begin{aligned} &\Omega, \Gamma \vdash_{\mathrm{v}} v : \mathtt{obj}(ql) \\ &\Omega, \Sigma \vdash_{\mathrm{r}} ql.a : t \\ &\quad A \vdash_a p \end{aligned} \tag{25}$$

with $v = (l, i)$.

Case distinction over q.

- *Case* $q = \tilde{}$: Because of **INV-CLS** it follows $(l, i) \in H, H_0$. Because the pointer is unblocked $(l, i) \in \mathrm{dom}(H)$ holds. Yet SRd is applicable.
- *Case* $q = @$: (25) implies $l \in \mathrm{dom}(\Sigma)$. Heap consistency implies $\exists! j : (l, j) \in \mathrm{dom}(H_0)$. Since **INV-CLS** holds for e, $(ql, i) \in H, H_0$. This implies $(l, i) \in \mathrm{dom}(H_0)$, such that $j = i$. Hence SRd is applicable.

End case distinction over q.

– *Case* $v_1.a := v_2$: As in the read rule we get from **INV-CLS** that v_1 is a pointer to an object present in the corresponding heap. That's why we can apply SWrt.

– *Case* $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$: Depending on the structure of e_1 we can apply SLet or SLet′. The first is applicable if e_1 is a value, the second if e_1 is not a value we can use induction.

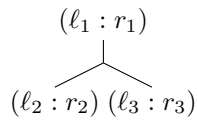
– *Case* $\mathtt{if}\ v\ e_1\ e_2$: Either SIfT or SIfF is applicable.

End case distinction over structure of e.

Most Recent Environment We present here a way to look at local environments as trees. The mapping is easy.

$$\frac{\text{as text } \emptyset\ (\ell : r)\Sigma\ \Sigma_1 \parallel \Sigma_2}{\text{as tree } \square\ \ (\ell : r)\ \ \overbrace{\Sigma_1^T\ \Sigma_2^T}\ \ \Sigma^T}$$

Now we can identify one tree as a set of mappings by starting at the root, go down the tree and end at a leaf, collecting all labels on the way down. In the tree



one map is $(\ell_1 : r_1)(\ell_2 : r_2)$ and an other one is $(\ell_1 : r_1)(\ell_3 : r_3)$. We write $\mathcal{M} \in \Sigma$ for one mapping from Σ. The heap consistency relation $\Sigma \Vdash_\Sigma H$ states, that there exists exactly one mapping $\mathcal{M}$ in Σ, such that for each $\ell \in \mathrm{dom}(\mathcal{M})$: $\exists! i : \mathcal{M}(\ell) \Vdash_o H(\ell, i)$.

Lemma 11. *It holds for all* $\Sigma, \Sigma', t, L, a, n$*:*

$$\begin{aligned}&\big(\Sigma \vdash^{@}_{\mathrm{w}} L.a := t \Rightarrow \Sigma', n\big)\\ \longrightarrow&\big(\forall \mathcal{M} \in \Sigma, \mathcal{M}' \in \Sigma' : |\{\ell \mid \mathcal{M}(\ell) \neq \mathcal{M}'(\ell)\}| \leq n\big)\end{aligned}$$

The interesting case is here when $n = 1$ holds. This ensure that for each $\mathcal{M} \in \Sigma$ at most one type change happens.

Heap consistency ensures, that for each most recent environment Σ there exists exactly one $\mathcal{M} \in \Sigma$, such that for each $\ell \in \mathrm{dom}(\mathcal{M})$ there exists i, such that $(\ell, i) \in \mathrm{dom}(H)$. Additionally, we know for all other $\ell \in \mathrm{dom}(\Sigma)$, $\ell \times \mathbf{Z} \cap \mathrm{dom}(H) = \emptyset$.

Hence the write operation can change for all $\ell \notin \mathrm{dom}(\mathcal{M})$ the most recent heap description Σ without any effect for the heap consistency judgment.

Because if $n = 1$, only for exactly one $\ell \in \mathrm{dom}(\mathcal{M})$ an change of the heap description is allowed, a strong update for the write operation even on union types of precise pointers is sound.

Proof. by induction over the structure of Σ.
Case distinction over Σ.

- *Case* $\Sigma = \emptyset$: In this we can conclude $\Sigma' = \emptyset$ and $n = 0$ by inversion of the definition of $\vdash^{@}_{\mathrm{w}}$. Hence $\mathcal{M}$ and $\mathcal{M}'$ are the empty maps. This yields $|\{\ell \mid \mathcal{M}(\ell) \neq \mathcal{M}'(\ell)\}| = |\emptyset| = 0 \leq n$.
- *Case* $\Sigma = (\ell : r)\Sigma_1$: Let n be an arbitrary natural number. There are two cases, either $\ell \in L$ or $\ell \notin L$.
  Case distinction over $\ell \in L, \ell \notin L$.
  - *Case* $\ell \in L$: Inversion of $\vdash^{@}_{\mathrm{w}}$ yields

$$\begin{aligned}\Sigma_1 \vdash^{@}_{\mathrm{w}} L.a := t \Rightarrow \Sigma_1', n-1\\ \Sigma' = (\ell : r[a \mapsto t])\Sigma_1'\end{aligned} \tag{26}$$

    Using the induction hypothesis, we get

$$\begin{aligned}&\forall \mathcal{M} \in \Sigma_1, \mathcal{M}' \in \Sigma_1' :\\ &|\{\ell \mid \mathcal{M}(\ell) \neq \mathcal{M}'(\ell)\}| \leq n-1 \quad .\end{aligned}$$

    This together with (26) told us that $\forall \mathcal{M} \in \Sigma$ and $\forall \mathcal{M}' \in \Sigma'$ we know

$$\begin{aligned}|\{\ell \mid \mathcal{M}(\ell) \neq \mathcal{M}'(\ell)\}| &\leq n-1+1\\ &\leq n\end{aligned}$$

  - *Case* $\ell \notin L$: This case is easy by induction.

  End case distinction over $\ell \in L, \ell \notin L$.
- *Case* $\Sigma = \Sigma_1 \parallel \Sigma_2$: Induction hypothesis for Σ_1 and Σ_2 yields to the desired result.

End case distinction over Σ.

Subsumption

Lemma 12 (Subsumption). *Suppose that*

$$\Omega, \Gamma \vdash_{\mathrm{v}} v : t \quad .$$

Then there exists a derivation with at most one application of the Subsumption *at the end of the derivation.*

Proof. As usual by induction over $\vdash_{\mathrm{v}}$.

Preservation

Lemma 13 (Preservation). *Suppose that*

$$\begin{aligned} H, H_0, e &\longrightarrow H', H_0', e' \\ \text{and } \Omega, \Sigma &\Vdash_e H, H_0, e : t \Rightarrow L, A, \Sigma' \quad . \end{aligned}$$

Then there exists some Σ_n, A_n and L_n with

$$\Omega, \Sigma_n \Vdash_e H', H_0', e' : t \Rightarrow L_n, A_n, \Sigma'$$

and $L_n \subseteq L$ and $A_n \subseteq A$.

Proof. Case distinction over definition of $\longrightarrow$.

– *Case* SDem:

$$\begin{aligned} H, H_0, \natural^{L_d} e &\longrightarrow (H, H_0)^{\natural L_d}, e \\ \Omega, \Sigma &\Vdash_e \natural^{L_d} e : t \Rightarrow L, A, \Sigma' \end{aligned}$$

Inversion of the typing consistency judgments yields

$$\begin{aligned} \Omega, \Sigma &\Vdash H, H_0 \\ \Omega, \Sigma, \emptyset &\vdash_{\mathrm{e}} \natural^{L_d} e : t \Rightarrow L, A, \Sigma', \emptyset \end{aligned} \tag{27}$$

Now inversion of (27) using Demote yields

$$\begin{aligned} L_d &\subseteq L \\ \Omega &= \Omega^{\natural L_d} \\ (\forall l \in L_d)\ \Omega, \Sigma &\vdash_{\mathrm{t}} \mathtt{obj}(@l) \lhd \mathtt{obj}(\tilde{}l) && (28) \\ \Sigma_n &= \Sigma^{\natural L_d} \upharpoonright L_d && (29) \\ \Omega, \Sigma_n, \emptyset &\vdash_{\mathrm{e}} e : t \Rightarrow L, A, \Sigma', \emptyset \end{aligned}$$

For $H', H_0' = (H, H_0)^{\natural L_d}$ is holds (make use of (28) and (29))

$$\Omega, \Sigma_n \Vdash H', H_0'$$

Notice that $A = A_n$ and $L = L_n$.

– *Case* SApp:

$$H, H_0, (\lambda x.e)(v) \longrightarrow H, H_0, e\{x \Mapsto v\}$$
$$\Omega, \Sigma \Vdash_e H, H_0, (\lambda x.e)(v) : t \Rightarrow L, A, \Sigma'$$

Inversion yields

$$\Omega, \Sigma \Vdash H, H_0 \tag{30}$$
$$\Omega, \Sigma, \emptyset \vdash_{\mathrm{e}} (\lambda x.e)(v) : t \Rightarrow L, \Sigma', \emptyset \tag{31}$$

Further inversion of (31) with the Function Call rule yields

$$\Sigma = \Sigma^{\natural L}$$
$$\Gamma = \Gamma^{\natural L}$$
$$\mathrm{dom}(\Sigma) \cap L = \emptyset$$
$$\Sigma, A \vdash_S \Sigma_1, \Sigma_2 \tag{32}$$
$$\Sigma', A \vdash_S \Sigma_1', \Sigma_2 \tag{33}$$
$$\Omega, \emptyset \vdash_{\mathrm{v}} v : t_2 \tag{34}$$
$$\Omega, \emptyset \vdash_{\mathrm{v}} \lambda x.e : (\Sigma_1, t_2) \overset{L,A}{\to} (\Sigma_1', t) \tag{35}$$

Inversion of (35) with the Function rule yields

$$\mathrm{dom}(\Sigma_1) \cap L = \emptyset$$
$$L' \cup L'' \subseteq L \tag{36}$$
$$\Gamma' = \emptyset = (\Gamma \downarrow \mathrm{fv}(\lambda(y, x).e))$$
$$L' = \emptyset = \mathrm{Locs}(\Gamma')$$
$$\Gamma'' = \emptyset = \Gamma'^{\natural L'}$$
$$\Omega, \Sigma_1, \emptyset(x : t_2) \vdash_{\mathrm{e}} e : t \Rightarrow L'', A'', \Sigma_1', \emptyset(x : t_2') \tag{37}$$

Because of (34) v is closed. We apply Lemma 8 (substitution) on (37), and (34) and get

$$\Omega, \Sigma_1, \emptyset \vdash_{\mathrm{e}} e\{x \Mapsto v\} : t \Rightarrow L'', A'', \Sigma_1', \emptyset \tag{38}$$
$$A'' \subseteq A$$

The claim follows by (30), (38) and (36) and the use of (32), (33) to change from Σ_1, Σ_1' to Σ, Σ'.

– *Case* SLet:

$$H, H_0, \mathtt{let}^{L_1}\ x = v\ \mathtt{in}\ e \longrightarrow H, H_0, e\{x \Mapsto v\}$$

Inversion of the heap typing rule for the left-hand side yields

$$\Omega, \Sigma \Vdash H, H_0 \tag{39}$$
$$\Omega, \Sigma, \emptyset \vdash_{\mathrm{e}} \mathtt{let}^{L_1}\ x = v\ \mathtt{in}\ e : t \Rightarrow L, A, \Sigma', \emptyset \tag{40}$$

Inverting (40) with the LET rule and the VALUE rule yields

$$\Omega, \Sigma, \emptyset \vdash_{\mathrm{e}} v : t_1 \Rightarrow \emptyset, \emptyset, \Sigma, \emptyset$$
$$\Omega, \emptyset \vdash_{\mathrm{v}} v : t_1 \tag{41}$$
$$\Omega, \Sigma, \emptyset(x : t_1) \vdash_{\mathrm{e}} e : t \Rightarrow L_2, A_2, \Sigma_2, \emptyset(x : t_1') \tag{42}$$
$$L = \emptyset \cup L_2$$
$$A = A_2 - L_1 = A_2$$

Applying Lemma 8 to (41) and (42) yields the desired

$$\Omega, \Sigma, \emptyset \vdash_{\mathrm{e}} e\{x \Mapsto v\} : t \Rightarrow L_2, A_2, \Sigma_2, \emptyset$$
$$L_2 \subseteq L$$
$$A_2 \subseteq A$$

Because heaps doesn't change, heap consistency is trivial.

– *Case* SNEW: We know that

$$H, H_0, \mathtt{new}^l \longrightarrow H, H_0[(\ell, i) \mapsto \{\}], (@l, i) \tag{43}$$
$$\mathrm{dom}(H_0) \cap (\{\ell\} \times \mathbf{Z}) = \emptyset$$
$$(\ell, i) \notin \mathrm{dom}(H)$$
$$\Omega, \Sigma \Vdash_e H, H_0, \mathtt{new}^l : \mathtt{obj}(@l) \Rightarrow \{l\}, \Sigma'' \tag{44}$$

where $\Sigma'' = \Sigma(l : \{\})$. Now we have to show that there exists Σ' so that

$$\Omega, \Sigma' \Vdash_e H, H_0', (@l, i) : \mathtt{obj}(@l) \Rightarrow \emptyset, \Sigma'' \tag{45}$$

First inverting (44) yields

$$\Omega, \Sigma \Vdash H, H_0 \tag{46}$$
$$\Omega, \Sigma, \emptyset \vdash_{\mathrm{e}} \mathtt{new}^l : \mathtt{obj}(@l) \Rightarrow \{l\}, \emptyset, \Sigma(\ell \mapsto \{\}), \emptyset \tag{47}$$
$$\Sigma'' = \Sigma(\ell \mapsto \{\})$$

From the defintion of $\vdash_{\mathrm{e}}$ and $\vdash_{\mathrm{v}}$ we get

$$\Omega, \Sigma'', \emptyset \vdash_{\mathrm{e}} (@l, i) : \mathtt{obj}(@l) \Rightarrow \emptyset, \emptyset, \Sigma'', \emptyset$$

For Σ' we choose Σ''. Now it remains to show $\Omega, \Sigma'' \Vdash H', H_0'$ to fullfill (45). The first holds by inverting (47) and using definition of Σ''. Inversion of (47) also gives us $l \notin \mathrm{dom}(\Sigma)$. This tells us that $\forall i : (l, i) \notin \mathrm{dom}(H_0)$. From this we get the information that only one object suitable to l is in H_0'. That's why the second judgment holds. Because H and Ω does not change the last judgment is trivial.

– *Case* SRD: Given is for $q = @$ or $q = \tilde{}$

$$\begin{gathered} H, H_0, (ql, i).a \longrightarrow H, H_0, (H \cup H_0)(l, i)\$a \\ \Omega, \Sigma \Vdash_e H, H_0, (ql, i).a : t \Rightarrow \emptyset, A, \Sigma \end{gathered} \tag{48}$$

We have to show that

$$\begin{gathered} \Omega, \Sigma \Vdash_e H, H_0, (H \cup H_0)(l, i)\$a : t \Rightarrow L_n, A_n, \Sigma \\ L_n \subseteq L \qquad A_n \subseteq A \end{gathered} \tag{49}$$

Inverting (48) gives us consistancy of heaps and their typings and the typing of the expression

$$\begin{gathered} \Omega, \Sigma \Vdash H, H_0 \\ \Omega, \Sigma, \emptyset \vdash_{\mathrm{e}} (ql, i).a : t \Rightarrow \emptyset, A, \Sigma, \emptyset \end{gathered} \tag{50}$$

and after another inversion of the second part

$$\begin{gathered} \Omega, \emptyset \vdash_{\mathrm{v}} (ql, i) : \mathtt{obj}(qL_r) \\ \Omega, \Sigma \vdash_{\mathrm{r}} ql.a : t \\ A \vdash_a qL_r \\ l \in L_r \end{gathered} \tag{51}$$

Case distinction over the precision of the pointer.

- *Case* $ql = @l, \Rightarrow (l, i) \in \mathrm{dom}(H_0)$:
  Since (51) $l \in \mathrm{dom}(\Sigma)$. Because of (50) we know

  $$\Omega, \Sigma \Vdash_o H_0(l, i) : \Sigma(l)$$

  Cause of (51) we get $\Sigma(l)(a) = t'$, $t' <: t$. Inverting the line above yields (make use of SUBSUMPTION)

  $$\Omega, \emptyset \vdash_{\mathrm{v}} H_0(l, i)\$a : t$$

- *Case* $ql = \tilde{}l, \Rightarrow (l, i) \in \mathrm{dom}(H)$:
  The inversion of (51) yields $\Omega(l)(a) = t', t' <: t$. From (50) we get $\Omega, \Sigma \Vdash_o H(l, i) : \Omega(l)$, hence

  $$\Omega, \emptyset \vdash_{\mathrm{v}} H(l, i)\$a : t \quad .$$

  As in the other case we fullfill (49).

End case distinction over the precision of the pointer. In both cases $L_n = \emptyset \subseteq L$ and $A_n = \emptyset \subseteq A$.

– *Case* SWRT: Hence $e = (q\ell, i).a := v$. Inversion of the configuration typing yields:

$$\begin{gathered} L = \emptyset \\ \Omega, \Sigma, \emptyset \vdash_{\mathrm{e}} (q\ell, i).a := v : \mathtt{udf} \Rightarrow \emptyset, A, \Sigma', \emptyset \\ \Omega, \Sigma \Vdash H, H_0 \end{gathered} \tag{52}$$

Further inversion of (52) yields

$$\Omega, \emptyset \vdash_{\mathrm{v}} (q\ell, i) : \mathtt{obj}(qL') \tag{53}$$
$$\Omega, \emptyset \vdash_{\mathrm{v}} v : t$$
$$\Omega, \Sigma \vdash_{\mathrm{w}} qL'.a := t \Rightarrow \Sigma' \tag{54}$$

(53) yields : $\ell \in L'$.

Case distinction over q.

- *Case* $q = @$: Inversion of (54) yields

$$\Sigma \vdash_{\mathrm{w}}^{@} L'.a := t \Rightarrow \Sigma', 1 \tag{55}$$

  From Lemma 11 we can conclude that the update yields in a new local environment description where a strong update happens and in a change to the most recent heap. The new heap is consistent to the new local environment description. The expression `udf` is typed.
- *Case* $q = \tilde{}$: SWRT change the property of the object in the summary heap. The inversion of the judgment $\vdash_{\mathrm{w}}$ ensures that the value that is written to the property has a type that is a subtype of $\Omega(\ell)(a)$. Hence heap consistency is ensured after the update. Typing the expression `udf` trivial.

End case distinction over q.

– *Case* SLET′: Easy be induction.

End case distinction over definition of $\longrightarrow$.

3 The Date Example

A challenging example for object initialization is the `Date` example presented by Anderson, Giannini and Drossopoulou in their work over type inference for JavaScript [1]. We show that our type system can handle this example as well. First we transform the example into our core calculus. Since the calculus does not support a constructor call of the form $\mathtt{new} f(v)$, we make minor adjustments.

First have a look at the function `Date`. It creates a new object at abstract location ℓ, sets the two properties `mSec` and `add` and returns the new object. The effect of the `Date` function is $\{\ell\}$. Hence the two last let assignments, where two `Date` objects are created, are covered by demotions.

$$
\begin{array}{l}
\texttt{let } addFn = \lambda(this, x). \\
\quad \texttt{let } \_ = this.mSec := this.mSec + x.mSec \texttt{ in} \\
\quad\quad this \\
\texttt{in} \\
\texttt{let } Date = \lambda x. \\
\quad \texttt{let } this = \texttt{new}^{\ell} \texttt{ in} \\
\quad \texttt{let } \_ = this.mSec := x \texttt{ in} \\
\quad \texttt{let } \_ = this.add := addFn \texttt{ in} \\
\quad\quad this \\
\texttt{in} \\
\texttt{let } x = Date(1000) \texttt{ in} \\
\texttt{let } y = Date(10000) \texttt{ in} \\
\quad x.add(y)
\end{array}
$$

In the last line the type of `x` is $\texttt{obj}(\tilde{}\ell)$. Define o as a shortcut:

$$
\begin{array}{l}
o := [mSec \mapsto \texttt{int}; \\
\qquad add \mapsto (\emptyset, \texttt{obj}(\tilde{}\ell) \times \texttt{obj}(\tilde{}\ell)) \to (\emptyset, \texttt{obj}(\tilde{}\ell))] \quad .
\end{array}
$$

Because the method parameter is not a precise object, we need a demotion to type the method call. If a demotion moves `y` into the summary heap in front of the method call, the type of y is $\texttt{obj}(\tilde{}\ell)$ in the last line. Thus the global environment and the type environment are there

$$
\begin{array}{l}
\Omega = [\ell \mapsto o] \\
\Gamma = [x \mapsto \texttt{obj}(\tilde{}\ell); y \mapsto \texttt{obj}(\tilde{}\ell)] \quad .
\end{array}
$$

With such types the method call is valid and the complete example works.

Please notice that our implementation supports methods, even if our formal system do. The extension is trivial, add it will add an analogous case to all proofs.

References

1. C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *19th ECOOP*, number 3586 in LNCS, Glasgow, Scotland, July 2005. Springer.