

Type and History Effect

Chris and Scott

July 10, 2003

1 The Language λ_{hist}

1.1 Syntax

$b ::= \text{true} \mid \text{false}$	<i>boolean values</i>
$v ::= x \mid \lambda_z x.e \mid b \mid \neg \mid \vee \mid \wedge \mid ()$	<i>values</i>
$e ::= v \mid ee \mid ev \mid \text{if } e \text{ then } e \text{ else } e \mid \text{check } P$	<i>expressions</i>
$\eta ::= \epsilon \mid \eta; ev$	<i>histories</i>
$E ::= [] \mid vE \mid Ee \mid \text{if } E \text{ then } e \text{ else } e$	<i>evaluation contexts</i>

1.2 Semantics

$\eta, (\lambda_z x.e)v \rightarrow \eta, e[v/x][\lambda_z x.e/z]$	<i>(β)</i>
$\eta, \neg b \rightarrow \eta, \llbracket \neg b \rrbracket_{\text{bool}}$	<i>(not)</i>
$\eta, b_1 \wedge b_2 \rightarrow \eta, \llbracket b_1 \wedge b_2 \rrbracket_{\text{bool}}$	<i>(and)</i>
$\eta, b_1 \vee b_2 \rightarrow \eta, \llbracket b_1 \vee b_2 \rrbracket_{\text{bool}}$	<i>(or)</i>
$\eta, \text{if true then } e_1 \text{ else } e_2 \rightarrow \eta, e_1$	<i>(if1)</i>
$\eta, \text{if false then } e_1 \text{ else } e_2 \rightarrow \eta, e_2$	<i>(if2)</i>
$\eta, ev \rightarrow \eta; ev, ()$	<i>(event)</i>
$\eta, P \rightarrow \eta, ()$	<i>if $P(\eta)$ (check)</i>
$\eta, E[e] \rightarrow \eta', E[e']$	<i>if $\eta, e \rightarrow \eta', e'$ (context)</i>

2 Types for λ_{hist}

$\tau ::= a \mid \tau \xrightarrow{H} \tau \mid \text{bool} \mid \text{unit}$
$H ::= \epsilon \mid h \mid ev \mid H; H \mid H H \mid \mu h.H \mid P$

$$\begin{array}{c}
\text{VAR} \qquad \qquad \qquad \text{BOOL} \qquad \qquad \qquad \text{UNIT} \\
\Gamma, \epsilon \vdash x : \Gamma(x) \qquad \Gamma, \epsilon \vdash b : \text{bool} \qquad \Gamma, \epsilon \vdash () : \text{unit} \\
\\
\text{AND} \qquad \qquad \qquad \qquad \qquad \qquad \text{OR} \\
\Gamma, \epsilon \vdash \wedge : \text{bool} \xrightarrow{\epsilon} \text{bool} \xrightarrow{\epsilon} \text{bool} \qquad \Gamma, \epsilon \vdash \vee : \text{bool} \xrightarrow{\epsilon} \text{bool} \xrightarrow{\epsilon} \text{bool} \\
\\
\text{NOT} \qquad \qquad \qquad \text{EVENT} \qquad \qquad \qquad \text{CHECK} \\
\Gamma, \epsilon \vdash \neg : \text{bool} \xrightarrow{\epsilon} \text{bool} \qquad \Gamma, ev \vdash ev : \text{unit} \qquad \Gamma, P \vdash \text{check } P : \text{unit} \\
\\
\text{ABS} \\
\frac{\Gamma; x : \tau_1; z : \tau_1 \xrightarrow{h} \tau_2, H \vdash e : \tau_2 \quad h \text{ fresh}}{\Gamma, \epsilon \vdash \lambda_z x. e : \tau_1 \xrightarrow{\mu h. H} \tau_2} \\
\\
\text{APP} \\
\frac{\Gamma, H_1 \vdash e_1 : \tau' \xrightarrow{H_3} \tau \quad \Gamma, H_2 \vdash e_2 : \tau'}{\Gamma, H_1; H_2; H_3 \vdash e_1 e_2 : \tau} \\
\\
\text{IF} \\
\frac{\Gamma, H_1 \vdash e_1 : \text{bool} \quad \Gamma, H_2 \vdash e_2 : \tau \quad \Gamma, H_3 \vdash e_3 : \tau}{\Gamma, H_1; H_2 | H_3 \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}
\end{array}$$

2.1 Interpretation

Definition 1 Our interpretation of histories maps the latter to possibly infinite sets S of event traces s , which are variable-free sequences of events:

$$S \ni s ::= \epsilon \mid ev \mid P \mid s; s \quad \text{event traces}$$

Definition 2 A history η is obtained from an event trace s by removing all checks events; we call this squeezing, written \hat{s} and inductively defined as follows:

$$\begin{aligned}
\hat{ev} &= ev \\
\hat{P} &= \epsilon \\
s_1; \hat{s}_2 &= \hat{s}_1; \hat{s}_2
\end{aligned}$$

We say that an event trace s is P -valid iff for all prefixes of s of the form s' ; P it is the case that $P(\hat{s}')$ holds; an event trace s is valid iff it is P -valid for all P . An event trace set S is valid iff every $s \in S$ is.

Definition 3 We inductively define the n th unrolling of a μ -bound history $\mu h. H$, denoted $\circlearrowleft_n \mu h. H$, as follows:

$$\begin{aligned}
\circlearrowleft_0 \mu h. H &= H[\epsilon/h] \\
\circlearrowleft_n \mu h. H &= H[\circlearrowleft_{n-1} \mu h. H/h]
\end{aligned}$$

Definition 4 The history interpretation function $\llbracket \cdot \rrbracket$, mapping closed histories to event trace sets, is inductively defined as follows:

$$\begin{aligned}
\llbracket \epsilon \rrbracket &= \{ \epsilon \} \\
\llbracket ev \rrbracket &= \{ ev \} \\
\llbracket P \rrbracket &= \{ P \} \\
\llbracket H_1; H_2 \rrbracket &= \{ s_1; s_2 \mid s_1 \in \llbracket H_1 \rrbracket \wedge s_2 \in \llbracket H_2 \rrbracket \} \\
\llbracket H_1 | H_2 \rrbracket &= \llbracket H_1 \rrbracket \cup \llbracket H_2 \rrbracket \\
\llbracket \mu h.H \rrbracket &= \bigcup_{i=0}^{\omega} \llbracket \circlearrowleft_i \mu h.H \rrbracket
\end{aligned}$$

Overloading terminology, we say that a closed history H is valid iff $\llbracket H \rrbracket$ is.

Corollary 1 If $\llbracket H \rrbracket \subseteq \llbracket H' \rrbracket$ then $\llbracket \hat{H} \rrbracket \subseteq \llbracket \hat{H}' \rrbracket$.

Corollary 2 For closed $\mu h.H$, we have that $\llbracket H[\mu h.H/h] \rrbracket \subseteq \llbracket \mu h.H \rrbracket$.

2.2 Properties

Corollary 3 If $\Gamma, H \vdash v : \tau$ is derivable, then $H = \epsilon$.

Lemma 1 If both $\Gamma; x : \tau', H \vdash e : \tau$ and $\Gamma, \epsilon \vdash v : \tau'$ are derivable, then so is $\Gamma, H \vdash e[v/x] : \tau$.

Lemma 2 If both $\Gamma; z : \tau_1 \xrightarrow{h} \tau_2, H \vdash e : \tau_2$ and $\Gamma, \epsilon \vdash \lambda_z x.e : \tau_1 \xrightarrow{\mu h.H} \tau_2$ are derivable, then so is $\Gamma, H[\mu h.H/h] \vdash e[v/x] : \tau_2$.

Lemma 3 If $\Gamma, H_1; H_2 \vdash E[e] : \tau$ is derivable with $\Gamma', H_1 \vdash e : \tau'$ following by a subderivation for e in the hole, and $\Gamma', H'_1 \vdash e' : \tau'$ is derivable, then $\Gamma, H'_1; H_2 \vdash E[e'] : \tau$ is derivable.

Lemma 4 If $\Gamma, H \vdash E[e] : \tau$ is derivable with e a redex, then $H = H_1; H_2$ with $\Gamma', H_1 \vdash e : \tau'$ following by a subderivation for e in the hole.

Lemma 5 If $\Gamma, H \vdash e : \tau$ is derivable and $\eta, e \rightarrow \eta', e'$, then $\Gamma, H' \vdash e' : \tau$ is derivable with $\llbracket \widehat{\eta; H} \rrbracket \supseteq \llbracket \widehat{\eta'; H'} \rrbracket$.

Proof. By case analysis on \rightarrow .

Case β . In this case the following assertions hold by assumption:

$$\begin{aligned}
e &= (\lambda_z x.e)v \\
e' &= e[v/x][\lambda_z x.e/z] \\
\eta' &= \eta
\end{aligned}$$

Furthermore, by definition of the typing rules and Corollary 3 we may partially reconstruct the derivation of $\Gamma, H \vdash e : \tau$ as follows, with $H = \mu h.H''$:

$$\frac{\frac{\Gamma; x : \tau'; z : \tau' \xrightarrow{h} \tau, H'' \vdash e : \tau \quad h \text{ fresh}}{\Gamma, \epsilon \vdash \lambda_z x.e : \tau' \xrightarrow{H} \tau} \quad \Gamma, \epsilon \vdash v : \tau}{\Gamma, H \vdash (\lambda_z x.e)v : \tau}$$

But then by Lemma 1 and Lemma 2 we have that $\Gamma, H''[H/h] \vdash e' : \tau$ is derivable, and by Corollary 1 and Corollary 2 we have $\widehat{[H]} \supseteq \widehat{[H''[H/h]]}$, so clearly $\widehat{[\eta; H]} \supseteq \widehat{[\eta'; H''[H/h]]}$.

Case *event*. In this case the following assertions hold by assumption:

$$\begin{aligned} e &= ev \\ e' &= () \\ \eta' &= \eta; ev \end{aligned}$$

Furthermore, by definition of the typing rules the following assertions hold:

$$\begin{aligned} H &= ev \\ \tau &= \text{unit} \\ \Gamma, \epsilon \vdash () &: \text{unit} \end{aligned}$$

and clearly $\widehat{[\eta; ev]} \supseteq \widehat{[\eta'; \epsilon]}$, so this case holds.

Case *check*. In this case the following assertions hold by assumption:

$$\begin{aligned} e &= \text{check } P \\ e' &= () \\ \eta' &= \eta \end{aligned}$$

Furthermore, by definition of the typing rules the following assertions hold:

$$\begin{aligned} H &= P \\ \tau &= \text{unit} \\ \Gamma, \epsilon \vdash () &: \text{unit} \end{aligned}$$

and clearly $\widehat{[\eta; P]} \supseteq \widehat{[\eta'; \epsilon]}$, so this case holds.

Case *context*. In this case the following assertions hold by assumption and Lemma 4:

$$\begin{aligned} e &= E[e_1] \text{ with } e_1 \text{ a redex} \\ e' &= E[e_2] \\ \eta, e_1 &\rightarrow \eta', e_2 \\ H &= H_1; H_2 \\ \Gamma', H_1 &\vdash e_1 : \tau' \end{aligned}$$

Furthermore, by the other cases of this Lemma and Lemma 3 we have:

$$\begin{aligned} \Gamma', H_1' &\vdash e_2 : \tau' \\ \widehat{[\eta; H_1]} &\supseteq \widehat{[\eta'; H_1']} \\ \Gamma', H_1'; H_2 &\vdash E[e_2] : \tau \end{aligned}$$

Therefore clearly $\widehat{[\eta; H_1; H_2]} \supseteq \widehat{[\eta'; H_1'; H_2]}$, so this case holds.

Corollary 4 *If $\Gamma, H \vdash e : \tau$ and $\epsilon, e \rightarrow^* \eta, v$ then $\eta \in \widehat{[H]}$.*

Theorem 1 (Progress) *If $\Gamma, H \vdash e : \tau$ is derivable and η, e is irreducible with $\eta; H$ valid, then e is a value.*

Theorem 2 (Type and History Safety) *If e is well-typed then e does not go wrong.*

Proof. Suppose on the contrary that $e, e \rightarrow^n \eta, e'$ with η, e' stuck. Since e is well-typed, there exists a derivable judgement $\Gamma, H \vdash e : \tau$ with valid H by definition. Furthermore, by Lemma 5 and induction on n we have that $\Gamma, H' \vdash e' : \tau$ such that $\widehat{[H]} \supseteq \widehat{[\eta; H']}$. But since H is valid, therefore $\eta; H'$ must also be valid, since the interpretation of the latter is contained in the former; hence e is a value by Theorem 1, which is a contradiction. \square

3 Parameterized Events

3.1 Syntax

To the term language syntax, we add a denumerable set of atomic constants which are first-class, along with parameterized events; just to anticipate state, I've put a values restriction on let:

$c \in \mathcal{C}$	<i>atomic constants</i>
$b ::= \text{true} \mid \text{false}$	<i>boolean values</i>
$v ::= x \mid \lambda_z x. e \mid c \mid b \mid \neg \mid \vee \mid \wedge \mid ()$	<i>values</i>
$e ::= v \mid e e \mid \text{ev}(e) \mid P(e) \mid$ $\text{if } e \text{ then } e \text{ else } e \mid \text{let } x = v \text{ in } e$	<i>expressions</i>
$\eta ::= \epsilon \mid \eta; \text{ev}(c)$	<i>histories</i>
$E ::= [] \mid v E \mid E e \mid \text{if } E \text{ then } e \text{ else } e \mid \text{ev}(E) \mid P(E)$	<i>evaluation contexts</i>

3.2 Semantics

$\eta, (\lambda_z x. e)v$	\rightarrow	$\eta, e[v/x][\lambda_z x. e/z]$	<i>(β)</i>	
$\eta, \neg b$	\rightarrow	$\eta, \llbracket \neg b \rrbracket_{\text{bool}}$	<i>(not)</i>	
$\eta, b_1 \wedge b_2$	\rightarrow	$\eta, \llbracket b_1 \wedge b_2 \rrbracket_{\text{bool}}$	<i>(and)</i>	
$\eta, b_1 \vee b_2$	\rightarrow	$\eta, \llbracket b_1 \vee b_2 \rrbracket_{\text{bool}}$	<i>(or)</i>	
$\eta, \text{if true then } e_1 \text{ else } e_2$	\rightarrow	η, e_1	<i>(if1)</i>	
$\eta, \text{if false then } e_1 \text{ else } e_2$	\rightarrow	η, e_2	<i>(if2)</i>	
$\eta, \text{let } x = v \text{ in } e$	\rightarrow	$\eta, e[v/x]$	<i>(let)</i>	
$\eta, \text{ev}(c)$	\rightarrow	$\eta; \text{ev}(c), ()$	<i>(event)</i>	
$\eta, P(c)$	\rightarrow	$\eta, ()$	<i>if $P(c)(\eta)$</i>	<i>(check)</i>
$\eta, E[e]$	\rightarrow	$\eta', E[e']$	<i>if $\eta, e \rightarrow \eta', e'$</i>	<i>(context)</i>

3.3 Types

For the type system, we add parameterized events to the language of histories, as well as singleton types $\{s\}$, where s is either a constant or a constant variable. We also add type schemes, quantified over constant variables:

$$\begin{aligned}
s & ::= c \mid \alpha \\
\tau & ::= \{s\} \mid \tau \xrightarrow{H} \tau \mid \text{bool} \mid \text{unit} \\
\sigma & ::= \forall \bar{\alpha}. \tau \\
H & ::= \epsilon \mid h \mid \text{ev}(s) \mid P(s) \mid H; H \mid H|H \mid \mu h. H
\end{aligned}$$

The kinding rules need to be defined and the model updated, but you get the idea. Note that singleton types can be interpreted as singleton set types:

$$\llbracket \{c\} \rrbracket = \{c+, \emptyset\}$$

Now, we need only redefine the EVENT rule, and add rules for constants, let, forall intro, and forall elim, to do the right thing:

$$\begin{array}{c}
\text{CONST} \\
\frac{}{\Gamma, \epsilon \vdash c : \{c\}} \\
\\
\text{EVENT} \\
\frac{\Gamma, H \vdash e : \{c\}}{\Gamma, H; \text{ev}(c) \vdash \text{ev}(e) : \text{unit}} \\
\\
\text{CHECK} \\
\frac{\Gamma, H \vdash e : \{c\}}{\Gamma, H; P(c) \vdash P(e) : \text{unit}} \\
\\
\text{\forall INTRO} \\
\frac{\Gamma, H \vdash e : \tau \quad \bar{\alpha} \cap \text{fv}(H, \Gamma) = \emptyset}{\Gamma, H \vdash e : \forall \bar{\alpha}. \tau} \\
\\
\text{\forall ELIM} \\
\frac{\Gamma, H \vdash e : \forall \bar{\alpha}. \tau}{\Gamma, H \vdash e : \tau[\bar{c}/\bar{\alpha}]} \\
\\
\text{LET} \\
\frac{\Gamma, \epsilon \vdash v : \sigma \quad (\Gamma; x : \sigma), H \vdash e : \tau}{\Gamma, H \vdash \text{let } x = v \text{ in } e : \tau}
\end{array}$$

4 Generative Constants

4.1 Syntax

$$\begin{array}{ll}
c \in \mathcal{C} & \text{atomic constants} \\
b ::= \text{true} \mid \text{false} & \text{boolean values} \\
v ::= x \mid \lambda_z x. e \mid c \mid b \mid \neg \mid \vee \mid \wedge \mid () & \text{values} \\
e ::= v \mid e e \mid \text{ev}(e) \mid P(e) \mid \text{new } x \text{ in } e \mid & \text{expressions} \\
& \text{if } e \text{ then } e \text{ else } e \mid \text{let } x = v \text{ in } e \\
\eta ::= \epsilon \mid \eta; \text{ev}(c) & \text{histories} \\
E ::= [] \mid v E \mid E e \mid \text{if } E \text{ then } e \text{ else } e \mid & \text{evaluation contexts} \\
& \text{ev}(E) \mid P(E) \mid \text{let } x = E \text{ in } e
\end{array}$$

Definition 5 We will be interested in non-recursive functions; thus, we write $\lambda x.e$ to denote $\lambda_z x.e$ where x is not free in e .

4.2 Semantics

$$\begin{array}{llll}
\eta, (\lambda_z x.e)v & \rightarrow & \eta, e[v/x][\lambda_z x.e/z] & (\beta) \\
\eta, \neg b & \rightarrow & \eta, \llbracket \neg b \rrbracket_{\text{bool}} & (\text{not}) \\
\eta, b_1 \wedge b_2 & \rightarrow & \eta, \llbracket b_1 \wedge b_2 \rrbracket_{\text{bool}} & (\text{and}) \\
\eta, b_1 \vee b_2 & \rightarrow & \eta, \llbracket b_1 \vee b_2 \rrbracket_{\text{bool}} & (\text{or}) \\
\eta, \text{if true then } e_1 \text{ else } e_2 & \rightarrow & \eta, e_1 & (\text{if1}) \\
\eta, \text{if false then } e_1 \text{ else } e_2 & \rightarrow & \eta, e_2 & (\text{if2}) \\
\eta, \text{let } x = v \text{ in } e & \rightarrow & \eta, e[v/x] & (\text{let}) \\
\eta, \text{new } x \text{ in } e & \rightarrow & \eta, e[c/x] & c \text{ fresh } (\text{new}) \\
\eta, \text{ev}(c) & \rightarrow & \eta; \text{ev}(c), () & (\text{event}) \\
\eta, P(c) & \rightarrow & \eta, () & \text{if } P(c)(\eta) (\text{check}) \\
\eta, E[e] & \rightarrow & \eta', E[e'] & \text{if } \eta, e \rightarrow \eta', e' (\text{context})
\end{array}$$

4.3 Types

In our type language we distinguish between “normal” variables α , and “distinguished singleton variables” ζ ; the latter are placeholders for substitution with fresh constants. We use ∇ as the binder for ζ variables in function types.

$$\begin{array}{ll}
s & ::= c \mid \alpha \mid \zeta \\
\tau & ::= s \mid \{\tau\} \mid \nabla \bar{\zeta}. \tau \xrightarrow{H} \tau \mid \text{bool} \mid \text{unit} \\
\sigma & ::= \forall \bar{\alpha}. h. \tau \\
H & ::= \epsilon \mid h \mid \text{ev}(s) \mid P(s) \mid H; H \mid H|H \mid \mu h. H
\end{array}$$

Definition 6 To extract free variables from expressions e , we define $\text{fv}(e)$ as usual, and define $\text{fv}_\zeta(e)$ to denote the free ζ variables in e . As a well-formedness conditions on types, we require for any $\nabla \bar{\zeta}. \tau_1 \xrightarrow{H} \tau_2$ that $\text{fv}_\zeta(\tau_1) = \emptyset$. We write $\tau_1 \xrightarrow{H} \tau_2$ for $\nabla \emptyset. \tau_1 \xrightarrow{H} \tau_2$.

Definition 7 ζ variables are placeholders for fresh constants, and at the top level we substitute fresh constants for ζ s to determine satisfiability. Thus, we have that $\Gamma, H \vdash e : \tau$ is valid iff it is derivable and $H[\bar{c}/\text{fv}_\zeta(H)]$ is valid, for \bar{c} fresh.

The type derivation rules are defined as follows. Since generative constants within recursive functions present such a technical problem and have no apparent purpose, we restrict constant generation to non-recursive functions. For this

reason we have two versions of the ABS rule.

$$\begin{array}{c} \text{CONST} \\ \Gamma, \epsilon \vdash c : \{c\} \end{array} \qquad \begin{array}{c} \text{EVENT} \\ \Gamma, H \vdash e : \{s\} \\ \hline \Gamma, H; ev(s) \vdash ev(e) : \text{unit} \end{array} \qquad \begin{array}{c} \text{CHECK} \\ \Gamma, H \vdash e : \{s\} \\ \hline \Gamma, H; P(s) \vdash P(e) : \text{unit} \end{array}$$

$$\begin{array}{c} \text{NEW} \\ \Gamma; x : \zeta, H \vdash e : \tau \quad \zeta \text{ fresh} \\ \hline \Gamma, H \vdash \text{new } x \text{ in } e : \tau \end{array}$$

$$\begin{array}{c} \text{ABS1} \\ \Gamma; x : \tau_1; z : \tau_1 \xrightarrow{h} \tau_2, H \vdash e : \tau_2 \quad \text{fv}(H, \tau_2) - \text{fv}(\Gamma) = \emptyset \quad h \text{ fresh} \\ \hline \Gamma, \epsilon \vdash \lambda_z x. e : \tau_1 \xrightarrow{\mu h. H} \tau_2 \end{array}$$

$$\begin{array}{c} \text{ABS2} \\ \Gamma; x : \tau_1, H \vdash e : \tau_2 \quad \text{fv}(H, \tau_2) - \text{fv}(\Gamma) = \bar{\zeta} \\ \hline \Gamma, \epsilon \vdash \lambda x. e : \nabla \bar{\zeta}. \tau_1 \xrightarrow{H} \tau_2 \end{array}$$

$$\begin{array}{c} \text{APP} \\ H_1, \Gamma \vdash e_1 : \nabla \bar{\zeta}. \tau' \xrightarrow{H_3} \tau \quad H_2, \Gamma \vdash e_2 : \tau' \quad \bar{\zeta}' \text{ fresh} \\ \hline \Gamma, H_1; H_2; H_3[\bar{\zeta}'/\bar{\zeta}] \vdash e_1 e_2 : \tau[\bar{\zeta}'/\bar{\zeta}] \end{array}$$

4.4 Full Let-Polymorphism

Grammatical sorts are starting to proliferate, so I'll use kinding to keep stuff organized. Here's a redefinition of the type language:

$$\begin{array}{l} \tau \quad ::= \quad c \mid \alpha \mid \zeta \mid \{\tau\} \mid H \mid \nabla \bar{\zeta}. \tau \xrightarrow{\tau} \tau \mid \text{bool} \mid \text{unit} \\ H \quad ::= \quad \epsilon \mid \alpha \mid ev(\tau) \mid P(\tau) \mid H; H \mid H|H \mid \mu\alpha. H \\ \sigma \quad ::= \quad \forall \bar{\alpha}. \tau \end{array}$$

Now, we posit the following kinds:

$$k ::= \text{Single} \mid \text{Type} \mid \text{History}$$

and for any k , we specify a disjoint, denumerable set of variables \mathcal{V}_k . We also specify a disjoint denumerable set of ζ -variables \mathcal{V}_ζ . The kinding rules are then

given as follows.

$$\begin{array}{c}
\text{bool} : \textit{Type} \quad \text{unit} : \textit{Type} \quad \zeta : \textit{Single} \quad \frac{\alpha \in \mathcal{V}_k}{\alpha : k} \quad \frac{\tau : \textit{Single}}{\{\tau\} : \textit{Type}} \\
\\
\frac{\tau_1, \tau_2 : \textit{Type} \quad H : \textit{History} \quad \text{fv}_\zeta(\tau_1) = \emptyset}{\nabla \bar{\zeta}. \tau_1 \xrightarrow{H} \tau_2 : \textit{Type}} \quad \epsilon : \textit{History} \\
\\
\frac{\tau : \textit{Single}}{ev(\tau) : \textit{History}} \quad \frac{\tau : \textit{Single}}{P(\tau) : \textit{History}} \quad \frac{\alpha : \textit{History} \quad H : \textit{History}}{\mu \alpha. H : \textit{History}} \\
\\
\frac{H_1 : \textit{History} \quad H_2 : \textit{History}}{H_1; H_2, H_1 | H_2 : \textit{History}}
\end{array}$$

With those in place, we specify the let-polymorphism-sensitive rules as follows. Note that we require substitutions to be kind-consistent, only substituting types of kind k for variables of kind k . Since we've sorted histories as types, the \forall rules allow generalization and instantiation of history variables, as well as normal type variables.

$$\begin{array}{c}
\forall \text{ INTRO} \quad \frac{\Gamma, H \vdash e : \tau \quad \bar{\alpha} \cap \text{fv}(H, \Gamma) = \emptyset}{\Gamma, H \vdash e : \forall \bar{\alpha}. \tau} \quad \forall \text{ ELIM} \quad \frac{\Gamma, H \vdash e : \forall \bar{\alpha}. \tau}{\Gamma, H \vdash e : \tau[\bar{\tau}/\bar{\alpha}]} \\
\\
\text{LET} \quad \frac{\Gamma, \epsilon \vdash v : \sigma \quad (\Gamma; x : \sigma), H \vdash e : \tau}{\Gamma, H \vdash \text{let } x = v \text{ in } e : \tau}
\end{array}$$

4.5 Increasing Flexibility

In the course of thinking about the following function, I've had some insights about aspects of the type system:

$$gcwrap \triangleq \lambda f. \text{new } x \text{ in } fx$$

Now, $gcwrap$ currently is not typable in our system, because we have no way to pick a type for f that anticipates the “freshness” of the type of x in the body. Furthermore, we have been conjecturing that first-order polyvariance is the way to go here, since it will allow us to instantiate the type of f . But I ask, what would be the type of $gcwrap$? In short, I argue that it should be:

$$gcwrap : \nabla \zeta. (\{\zeta\} \xrightarrow{h} \beta) \xrightarrow{h} \alpha$$

and when applied to the identity function id :

$$id \triangleq \lambda x. x : \{\zeta\} \xrightarrow{\epsilon} \{\zeta\}$$

we correctly track the type of the freshly created constant, returned by the identity function:

$$gwrap\ id : \{\zeta'\}$$

And none of this requires polyvariance of any sort. Of course, this narrative presupposes two things ruled out by our current analysis: first, that we can anticipate fresh ζ names, and second, that we can use such variables in function domain types (though you've said the latter should only be deprecated).

I believe this demonstrates we've been doing something wrong, because I believe my narrative should be true. But I think there's an easy way to make it so. First, rather than requiring "freshness" at application and new generation points, we define something call "p-distinctness":

Definition 8 *Any instance of a variable ζ specified as p-distinct in a given type derivation must be distinguished from all other p-distinct variables in the derivation.*

Notice this makes no restrictions about ζ s appearing in Γ , so we can possibly anticipate p-distinct variables in typing assumptions. Now, relax our restrictions on allowing ζ s in domain types, and replace freshness with p-distinctness in the type rules (Note: I use ζ to abbreviate $\{\zeta\}$ in the following):

$$\frac{\text{NEW} \quad \Gamma; x : \zeta, H \vdash e : \tau \quad \zeta \text{ p-distinct}}{\Gamma, H \vdash \text{new } x \text{ in } e : \tau}$$

$$\frac{\text{ABS2} \quad \Gamma; x : \tau_1, H \vdash e : \tau_2 \quad \text{fv}(\tau_1, H, \tau_2) - \text{fv}(\Gamma) = \bar{\zeta}}{\Gamma, \epsilon \vdash \lambda x. e : \nabla \bar{\zeta}. \tau_1 \xrightarrow{H} \tau_2}$$

$$\frac{\text{APP} \quad H_1, \Gamma \vdash e_1 : \nabla \bar{\zeta}. \tau' \xrightarrow{H_3} \tau \quad H_2, \Gamma \vdash e_2 : \tau' \quad \bar{\zeta}' \text{ p-distinct}}{\Gamma, H_1; H_2; H_3[\bar{\zeta}'/\bar{\zeta}] \vdash e_1 e_2 : \tau[\bar{\zeta}'/\bar{\zeta}]}$$

Now, we are free to derive types for id and $gwrap$ as above; let T_1 be the derivation:

$$\frac{\frac{\frac{\text{(instances of VAR)}}{f : \zeta \xrightarrow{\epsilon} \zeta; x : \zeta, \epsilon \vdash fx : \zeta} \text{APP}}{f : \zeta \xrightarrow{\epsilon} \zeta, \epsilon \vdash \text{new } x \text{ in } fx : \zeta} \text{NEW}}{\emptyset, \epsilon \vdash gwrap : \nabla \zeta. (\zeta \xrightarrow{\epsilon} \zeta) \xrightarrow{\epsilon} \zeta} \text{ABS2}}$$

and let T_2 be the derivation:

$$\frac{x : \zeta, \epsilon \vdash x : \zeta}{\emptyset, \epsilon \vdash id : \zeta \xrightarrow{\epsilon} \zeta} \text{ABS2}$$

Then we may achieve the desired result:

$$\frac{T_1 \quad T_2}{\emptyset, \epsilon \vdash gcwrap \ id : \zeta'} \text{APP}$$

I also conjecture that the interaction with let-polymorphism should not be problematic.

4.6 State of the Art

OK, so I think we may be at a point where we can state our state-of-the-art. We use the type language and kinding rules given in the last section, except we allow histories to be ∇ -bound:

$$H ::= \epsilon \mid \alpha \mid ev(\tau) \mid P(\tau) \mid H; H \mid H|H \mid \mu\alpha.H \mid \nabla\bar{\zeta}.H$$

and the kinding rules are extended thusly:

$$\frac{H : \text{History}}{\nabla\bar{\zeta}.H : \text{History}}$$

and the interpretation of histories is extended thusly:

$$\llbracket \nabla\bar{\zeta}.H \rrbracket = \llbracket H[\bar{\zeta}'/\bar{\zeta}] \rrbracket \quad \bar{\zeta}' \text{ fresh}$$

Also, judgements are now of the form:

$$\Gamma, H, \bar{\zeta} \vdash e : \tau$$

Here, the parameter $\bar{\zeta}$ keeps track of the ζ variables necessary for typing freshly generated constants the expression, so at abstraction only the right ones are ∇ -bound. Also, note that well-formedness of vectors $\bar{\zeta}$ imposes disjointness conditions— i.e. if $\bar{\zeta}_1\bar{\zeta}_2$ is well-formed then $\bar{\zeta}_1$ and $\bar{\zeta}_2$ are disjoint. Of course, we require all vectors in valid judgements to be well-formed. These clarifications eliminate any need for vaguely defined “freshness” or “p-distinctness”. The typing rules are then given in *Fig. 3*.

A new feature here is the use of a definition of type instantiation. This is defined modulo type equivalence, which is defined via our interpretation of types. We have as a natural consequence of this definition that:

$$\nabla\bar{\zeta}.\tau_1 \xrightarrow{H} \tau_2 = \tau_1 \xrightarrow{\nabla\bar{\zeta}.H} \tau_2 \quad \bar{\zeta} \cap \text{fv}_\zeta(\tau_1, \tau_2) = \emptyset$$

Thus, the migration rule holds by implication, and there’s so need to define it directly.

Definition 9 *The interpretation of kinds is parameterized by a ground type assignment ρ , and given inductively as follows:*

$$\begin{aligned}
\llbracket \alpha \rrbracket \rho &= \rho(\alpha) \\
\llbracket \zeta \rrbracket \rho &= \rho(\zeta) \\
\llbracket \text{unit} \rrbracket \rho &= \text{unit} \\
\llbracket \text{bool} \rrbracket \rho &= \text{bool} \\
\llbracket c \rrbracket \rho &= c \\
\llbracket \{\tau\} \rrbracket \rho &= \{\llbracket \tau \rrbracket \rho\} \\
\llbracket \nabla_{\bar{c}} \tau_1 \xrightarrow{H} \tau_2 \rrbracket \rho &= \llbracket \tau_1 \rho' \rrbracket \xrightarrow{\llbracket H \rrbracket \rho'} \llbracket \tau_2 \rho' \rrbracket \quad \rho' = \rho[\bar{c}/\bar{\zeta}] \quad \bar{c} \text{ fresh}
\end{aligned}$$

The interpretation of histories is given as previously (actually needs a little sanding, but close). We write $\rho \Vdash \tau = \tau'$ iff $\llbracket \tau \rrbracket \rho = \llbracket \tau' \rrbracket \rho$, and $\tau = \tau'$ iff $\forall \rho. \rho \Vdash \tau = \tau'$.

Definition 10 (Type Instantiation) *We have that $\forall \bar{\alpha}[\tau]. \preceq \tau'$ iff $\exists \bar{\tau}. \tau' = \tau[\bar{\tau}/\bar{\alpha}]$ (Milner's relation is the other way around, but this seems to follow subtyping).*

5 Localizing Multiple Histories

5.1 Syntax

$x \in \mathcal{V}$	<i>identifiers</i>
$\iota \in \mathcal{I}$	<i>history indices</i>
$c \in \mathcal{C}$	<i>atomic constants</i>
$b ::= \text{true} \mid \text{false}$	<i>boolean values</i>
$v ::= x \mid \lambda_z x. e \mid c \mid b \mid \iota \mid \neg \mid \vee \mid \wedge \mid ()$	<i>values</i>
$e ::= v \mid e e \mid \text{ev}_e(e) \mid P_e(e) \mid \text{history } x \text{ in } e$	<i>expressions</i>
$\quad \text{if } e \text{ then } e \text{ else } e \mid \text{let } x = v \text{ in } e$	
$\mathcal{H} \ni \eta ::= \epsilon \mid \eta; \text{ev}(c)$	<i>histories</i>
$\varsigma \in \mathcal{I} \rightarrow \mathcal{H}$	<i>logbook</i>
$\phi \in \mathcal{V} \rightarrow \mathcal{I}$	<i>index assignment</i>
$E ::= [] \mid v E \mid E e \mid \text{if } E \text{ then } e \text{ else } e \mid$	<i>evaluation contexts</i>
$\quad \text{ev}_\iota(E) \mid P_\iota(E)$	

VAR $\Gamma, \epsilon \vdash x : \Gamma(x)$	BOOL $\Gamma, \epsilon \vdash b : \text{bool}$	UNIT $\Gamma, \epsilon \vdash () : \text{unit}$
AND $\Gamma, \epsilon \vdash \wedge : \text{bool} \xrightarrow{\epsilon} \text{bool} \xrightarrow{\epsilon} \text{bool}$		OR $\Gamma, \epsilon \vdash \vee : \text{bool} \xrightarrow{\epsilon} \text{bool} \xrightarrow{\epsilon} \text{bool}$
NOT $\Gamma, \epsilon \vdash \neg : \text{bool} \xrightarrow{\epsilon} \text{bool}$	CONST $\Gamma, \epsilon \vdash c : \{c\}$	EVENT $\frac{\Gamma, H \vdash e : \{\tau\}}{\Gamma, H; \text{ev}(\tau) \vdash \text{ev}(e) : \text{unit}}$
CHECK $\frac{\Gamma, H \vdash e : \{\tau\}}{\Gamma, H; P(\tau) \vdash P(e) : \text{unit}}$	NEW $\frac{\Gamma; x : \zeta, H \vdash e : \tau \quad \zeta \text{ fresh}}{\Gamma, H \vdash \text{new } x \text{ in } e : \tau}$	
IF $\frac{\Gamma, H_1 \vdash e_1 : \text{bool} \quad \Gamma, H_2 \vdash e_2 : \tau \quad \Gamma, H_3 \vdash e_3 : \tau}{\Gamma, H_1; H_2 H_3 \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}$		
ABS1 $\frac{\Gamma; x : \tau_1; z : \tau_1 \xrightarrow{h} \tau_2, H \vdash e : \tau_2 \quad \text{fv}(H, \tau_2) - \text{fv}(\Gamma) = \emptyset \quad h \text{ fresh}}{\Gamma, \emptyset \vdash \lambda z x. e : \tau_1 \xrightarrow{\mu h. H} \tau_2}$		
ABS2 $\frac{\Gamma; x : \tau_1, H \vdash e : \tau_2 \quad \bar{\zeta} \cap \text{fv}_{\zeta}(\Gamma) = \emptyset}{\Gamma, \epsilon \vdash \lambda x. e : \nabla \bar{\zeta}. \tau_1 \xrightarrow{H} \tau_2}$		
APP $\frac{\Gamma, H_1 \vdash e_1 : \nabla \bar{\zeta}. \tau' \xrightarrow{H_3} \tau \quad \Gamma, H_2 \vdash e_2 : \tau' \quad \bar{\zeta}' \text{ fresh}}{\Gamma, H_1; H_2; H_3[\bar{\zeta}'/\bar{\zeta}] \vdash e_1 e_2 : \tau[\bar{\zeta}'/\bar{\zeta}]}$		
\forall INTRO $\frac{\Gamma, H \vdash e : \tau \quad \bar{\alpha} \cap \text{fv}(H, \Gamma) = \emptyset}{\Gamma, H \vdash e : \forall \bar{\alpha}. \tau}$	INST $\frac{\Gamma, H \vdash e : \tau \quad \tau \preceq \tau'}{\Gamma, H \vdash e : \tau'}$	
LET $\frac{\Gamma, \epsilon \vdash v : \sigma \quad (\Gamma; x : \sigma), H \vdash e : \tau}{\Gamma, H \vdash \text{let } x = v \text{ in } e : \tau}$		

Figure 1: Typing rules for λ_{hist} with generative constants

5.2 Semantics

Definition 11 The pre-processing of an expression e , denoted $\llbracket e \rrbracket$, is a function parameterized by an index assignment ϕ , that replaces each declared history variable with a distinct index; so, we have inductively:

$$\begin{aligned} \llbracket ev_x(e) \rrbracket \phi &= ev_{\phi x}(\llbracket e \rrbracket \phi) \\ \llbracket P_x(e) \rrbracket \phi &= P_{\phi x}(\llbracket e \rrbracket \phi) \\ \llbracket \text{history } x \text{ in } e \rrbracket \phi &= \llbracket e \rrbracket \phi[x : \iota] \quad \iota \text{ fresh} \end{aligned}$$

along with a homomorphic extension to the other expression forms:

$$\begin{aligned} \llbracket \lambda_z x. e \rrbracket \phi &= \lambda_z x. \llbracket e \rrbracket \phi \\ \llbracket e_1 e_2 \rrbracket \phi &= \llbracket e_1 \rrbracket \phi \llbracket e_2 \rrbracket \phi \\ &\vdots \end{aligned}$$

$$\begin{aligned} \varsigma, (\lambda_z x. e)v &\rightarrow \varsigma, e[v/x][\lambda_z x. e/z] && (\beta) \\ \varsigma, \neg b &\rightarrow \varsigma, \llbracket \neg b \rrbracket_{\text{bool}} && (\text{not}) \\ \varsigma, b_1 \wedge b_2 &\rightarrow \varsigma, \llbracket b_1 \wedge b_2 \rrbracket_{\text{bool}} && (\text{and}) \\ \varsigma, b_1 \vee b_2 &\rightarrow \varsigma, \llbracket b_1 \vee b_2 \rrbracket_{\text{bool}} && (\text{or}) \\ \varsigma, \text{if true then } e_1 \text{ else } e_2 &\rightarrow \varsigma, e_1 && (\text{if1}) \\ \varsigma, \text{if false then } e_1 \text{ else } e_2 &\rightarrow \varsigma, e_2 && (\text{if2}) \\ \varsigma, \text{let } x = v \text{ in } e &\rightarrow \varsigma, e[v/x] && (\text{let}) \\ \varsigma, ev_\iota(c) &\rightarrow \varsigma[\iota : (\varsigma(\iota); ev(c))], () && (\text{event}) \\ \varsigma, P_\iota(c) &\rightarrow \varsigma, () && \text{if } P(c)(\varsigma(\iota)) \quad (\text{check}) \\ \varsigma, E[e] &\rightarrow \varsigma', E[e'] && \text{if } \varsigma, e \rightarrow \varsigma', e' \quad (\text{context}) \end{aligned}$$

5.3 Types

It is straightforward to update the type language, by adding indexed events and checks:

$$\begin{aligned} s &::= c \mid \alpha \\ \tau &::= \{s\} \mid \tau \xrightarrow{H} \tau \mid \text{bool} \mid \text{unit} \\ \sigma &::= \forall \bar{\alpha}. \tau \\ H &::= \epsilon \mid h \mid ev_\iota(s) \mid P_\iota(s) \mid H; H \mid H|H \mid \mu h. H \end{aligned}$$

The type rules are obtained by adding an index assignment ϕ to judgements,

and keeping the rules mostly intact except for the following modifications:

$$\frac{\text{EVENT}}{\Gamma, H, \phi \vdash e : \{c\}} \quad \frac{\text{CHECK}}{\Gamma, H, \phi \vdash e : \{c\}}$$

$$\frac{}{\Gamma, H; ev_{\phi x}(c), \phi \vdash ev_x(e) : \text{unit}} \quad \frac{}{\Gamma, H; P_{\phi x}(c), \phi \vdash P_x(e) : \text{unit}}$$

$$\frac{\text{HDECL}}{\Gamma, H, \phi[x : \iota] \vdash e : \tau \quad \iota \text{ fresh}}{\Gamma, H, \phi \vdash \text{history } x \text{ in } e : \tau}$$

5.4 Interpretation

Definition 12 Letting \mathcal{S} be the set of all event traces as defined in Sect. 2.1, our revised interpretation of histories is the set of mappings from history indices to possibly infinite sets of event trace:

$$\Sigma \in \mathcal{I} \rightarrow 2^{\mathcal{S}} \quad \text{event trace index}$$

We let Σ_ϵ be the everywhere- $\{\epsilon\}$ event trace index. Also, we define the operations \cup and $;$ on event trace indexes:

$$\begin{aligned} (\Sigma_1; \Sigma_2)(\iota) &= \{s_1; s_2 \mid s_1 \in \Sigma_1(\iota) \wedge s_2 \in \Sigma_2(\iota)\} \\ (\Sigma_1 \cup \Sigma_2)(\iota) &= \Sigma_1(\iota) \cup \Sigma_2(\iota) \end{aligned}$$

Definition 13 Our revised history interpretation function $\llbracket \cdot \rrbracket$, mapping closed histories to event trace indexes, is inductively defined as follows:

$$\begin{aligned} \llbracket \epsilon \rrbracket &= \Sigma_\epsilon \\ \llbracket ev_\iota \rrbracket &= \Sigma_\epsilon[\iota : \{ev\}] \\ \llbracket P_\iota \rrbracket &= \Sigma_\epsilon[\iota : \{P\}] \\ \llbracket H_1; H_2 \rrbracket &= \llbracket H_1 \rrbracket; \llbracket H_2 \rrbracket \\ \llbracket H_1 | H_2 \rrbracket &= \llbracket H_1 \rrbracket \cup \llbracket H_2 \rrbracket \\ \llbracket \mu h. H \rrbracket &= \bigcup_{i=0}^{\omega} \llbracket \circlearrowleft_i \mu h. H \rrbracket \end{aligned}$$

We say that a closed history H is valid iff $\forall \iota. \llbracket H \rrbracket(\iota)$ is.

6 Applications

6.1 Characterizing Predicates

Although we have been abstract with our definition of predicates P to this point, when the rubber hits the road we'll need a sound implementation of predicates on history effects. One problem we immediately note is that history effects may contain histories $\mu h. H$, which represents an infinite number of histories. Obviously, computability requires that we not observe every history explicitly, so we have to find a way around this. A property of predicates that will resolve this is validity invariance for any number of unrollings; stated formally:

Definition 14 A predicate P is μ -invariant on H iff for any subhistory $\mu h.H'$ of H , P -validity of $\llbracket \circlearrowleft_0 \mu h.H' \rrbracket$ implies P -validity of $\cup_{i=0}^{\infty} \llbracket \circlearrowleft_i \mu h.H' \rrbracket$.

If we can demonstrate μ -invariance for any particular predicate, the implementation becomes simple; just unroll all μ -bound histories once and check validity. As we discuss below, stack inspection is μ -invariant, whereas a predicate like “file f has been read 5 times” is not.

6.2 Stack Inspection

6.2.1 Syntactic Transformation

$$\begin{aligned} \text{stackify}(\epsilon) &= \epsilon \\ \text{stackify}(\eta; \text{push}_r) &= \text{stackify}(\eta).r \\ \text{stackify}(\eta; \text{push}_p) &= \text{stackify}(\eta).p \\ \text{stackify}(\eta; \text{pop}) &= \text{let } s.x = \text{stackify}(\eta) \text{ in } s \end{aligned}$$

$$\begin{aligned} \llbracket \text{check } r \text{ then } e \rrbracket &= \text{inspect}_r \circ \text{stackify}; \llbracket e \rrbracket \\ \llbracket \text{enable } r \text{ in } e \rrbracket &= \text{push}_r; (\text{let } x = \llbracket e \rrbracket \text{ in } \text{pop}; x) \quad x \text{ fresh} \\ \llbracket p.e \rrbracket &= \text{push}_p; (\text{let } x = \llbracket e \rrbracket \text{ in } \text{pop}; x) \quad x \text{ fresh} \\ \llbracket \text{fix } z.\lambda x.e \rrbracket &= \lambda_z x. \llbracket e \rrbracket \\ \llbracket e_1 e_2 \rrbracket &= \llbracket e_1 \rrbracket \llbracket e_2 \rrbracket \\ &\vdots \end{aligned}$$

6.2.2 Typing Implementation

Lemma 6 Let e be a λ_{sec} expressions; if $\Gamma, H \vdash \llbracket e \rrbracket : \tau$, then $\text{inspect}_r \circ \text{stackify}$ is μ -invariant on H .

Proof. (Sketch). Pick any subhistory $\mu h.H'$ in H such that $\circlearrowleft_0 \mu h.H'$ is $\text{inspect}_r \circ \text{stackify}$ -valid. Observe that for all $\eta \in \llbracket \circlearrowleft_0 \mu h.H' \rrbracket$, we have that $\text{stackify}(\eta) = \epsilon$, by properties of λ_{sec} and our translation to λ_{hist} . It is then straightforward to obtain the desired result, that $\circlearrowleft_i \mu h.H'$ is $\text{inspect}_r \circ \text{stackify}$ -valid for all i , by induction on i , since any substitution for h in a particular unrolling will stackify to ϵ , meaning that the inductive case reduces to the 0 case, which is true by assumption.

6.3 History-Based Access Control

$$\lambda_z x.p.e$$

$$\llbracket \lambda_z x.p.e \rrbracket = \lambda_z x.ev_p; e$$

$$\text{intersect}(ev_{p_1}; \dots; ev_{p_n}) = p_n \wedge \dots \wedge p_1$$

$$\text{demand}_r(c) = (\lambda x. r(c) \in x) \circ \text{intersect}$$

$$\begin{aligned} & \text{filefact} \\ & = \\ \lambda_{-}. ev_p; \text{new } x \text{ in } \{ \dots; \text{delete} = (\lambda_{-}. ev_p; \text{demand}_{\text{fileDelete}}(x); \text{delete}(x)); \dots \} \\ & ev_{p'}; \dots; f. \end{aligned}$$

7 Type Inference

7.1 Basic

The type inference rules for the basic form of λ_{hist} , without parameters or generative constants, are given in Fig. 2. It is a constraint system, generating a conjunction of equations E . To solve these equations, we first posit a type unification algorithm unify_τ , that returns a pair ϕ, E_H , where ϕ is a unifier for the “type part” of E and E_H is the set of history equations imposed by E (easily generated during an arrow-type case of type unification). We then imagine an algorithm unify_H that returns a unifier ϕ of history equations E_H —at least in some cases. Separating these kinds of equations allows us to consider history unification in isolation.

Given these algorithms, soundness of inference would be stated as follows:

Lemma 7 *Suppose the following hold:*

$$\begin{aligned} \Gamma, H \vdash_W e : \tau / E \\ \text{unify}_\tau(E) = \phi_1, E_H \\ \text{unify}_H(E_H) = \phi_2 \end{aligned}$$

Then $\phi_2 \circ \phi_1(\Gamma), \phi_2(H) \vdash e : \phi_2 \circ \phi_1(\tau)$ is derivable.

Not sure how to state completeness, in the absence of principal types and uncertainty about history unification decidability.

7.2 Inference over Generative Theory

We need to define a partial function $\phi = \text{unify}_\tau(E)$. It is partial because unification may fail. Note I am doing it slightly differently than the previous section, unify_τ unifies the whole thing including the histories; it invokes unify_H

$$\begin{array}{c}
\frac{\Gamma(x) = \tau}{\Gamma, \epsilon \vdash_W x : \tau / \mathbf{true}} \quad \Gamma, ev \vdash_W ev : \mathbf{unit} / \mathbf{true} \quad \Gamma, P \vdash_W P : \mathbf{unit} / \mathbf{true} \\
\\
\frac{\Gamma; x : t'; z : t' \xrightarrow{h} t, H \vdash_W e : \tau / E \quad t', t, h \text{ fresh}}{\Gamma, \epsilon \vdash_W \lambda_z x. e : t' \xrightarrow{\mu h. H} t / E \wedge t = \tau} \\
\\
\frac{\Gamma, H_1 \vdash_W e_1 : \tau_1 / E_1 \quad \Gamma, H_2 \vdash_W e_2 : \tau_2 / E_2 \quad \Gamma, H_3 \vdash_W e_3 : \tau_3 / E_3}{\Gamma, H_1; H_2 | H_3 \vdash_W \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau / E_1 \wedge E_2 \wedge E_3 \wedge \tau_1 = \mathbf{bool} \wedge \tau_2 = \tau_3} \\
\\
\frac{\Gamma, H_1 \vdash_W e_1 : \tau_1 / E_1 \quad \Gamma, H_2 \vdash_W e_2 : \tau_2 / E_2 \quad t, h \text{ fresh}}{\Gamma, H_1; H_2; h \vdash_W e_1 e_2 : t / E \wedge \tau_1 = \tau_2 \xrightarrow{h} t}
\end{array}$$

Figure 2: Typing inference rules for basic λ_{hist}

as a subroutine. Since unification needs to be done in the let rule, this approach is ever so slightly cleaner.

Basically, $unify_\tau$ is purely structural on the standard types, like the normal unify; for unifying ∇ -bound types (function types), the matching of the zetas is computed lazily if there is more than one: in unifying the rest of the function type, the unifier produces a substitution on the zetas as well, and that is applied to the result type's ζ 's after the bodies have been unified. Details forthcoming. One other issue that arises in type unification is the two possible forms of function types; we will need to deal with that issue.

Also for this we will need subroutine $\phi = unify_H(H_1, H_2)$ which is obviously undecidable, so it will just fail more than the ideal function would. This $unify_H$ not only needs to show the histories have the same grammar, these may also contain metavariables h which need unifying. The hi-level spec is perhaps all we need to give for now: $\llbracket \phi(H_1) \rrbracket = \llbracket \phi(H_2) \rrbracket$ for $\phi = unify_H(H_1, H_2)$. Actually, we should also present a simple algorithm of the spirit of μ -invariance: recursive histories only unify with other recursive histories, and their bodies must directly unify as well (no unwinding). I think this simple guy will in fact work well nearly all the time anyway.

Comments

Why can τ be c in the generative theory? It turns out its handy above, but still its a bit odd.

AppLet is useful only for let-defined functions, thus its name.

$\frac{\text{VAR-}\tau \quad \Gamma(x) = \tau}{\Gamma, \epsilon \vdash_W x : \tau / \mathbf{true}}$	$\frac{\text{VAR-}\forall \quad \Gamma(x) = \forall \bar{\alpha}. \tau \quad \bar{\alpha}' \text{ fresh}}{\Gamma, \epsilon \vdash_W x : \tau[\bar{\alpha}'/\bar{\alpha}] / \mathbf{true}}$
$\text{BOOL} \quad \Gamma, \epsilon \vdash_W b : \text{bool} / \mathbf{true}$	$\text{UNIT} \quad \Gamma, \epsilon \vdash_W () : \text{unit} / \mathbf{true}$
$\text{AND} \quad \Gamma, \epsilon \vdash_W \wedge : \text{bool} \xrightarrow{\epsilon} \text{bool} \xrightarrow{\epsilon} \text{bool} / \mathbf{true}$	
$\text{OR} \quad \Gamma, \epsilon \vdash_W \vee : \text{bool} \xrightarrow{\epsilon} \text{bool} \xrightarrow{\epsilon} \text{bool} / \mathbf{true}$	$\text{NOT} \quad \Gamma, \epsilon \vdash_W \neg : \text{bool} \xrightarrow{\epsilon} \text{bool} / \mathbf{true}$
$\text{CONST} \quad \Gamma, \epsilon \vdash_W c : \{c\} / \mathbf{true}$	$\frac{\text{EVENT} \quad \Gamma, H \vdash_W e : \tau / E \quad t \text{ fresh}}{\Gamma, H; \text{ev}(t) \vdash_W \text{ev}(e) : \text{unit} / E \wedge \tau = \{t\}}$
$\text{CHECK} \quad \frac{\Gamma, H \vdash_W e : \tau / E}{\Gamma, H; P(t) \vdash_W P(e) : \text{unit} / E \wedge \tau = \{t\}}$	
$\text{NEW} \quad \frac{\Gamma; x : \zeta, H \vdash_W e : \tau / E \quad \zeta \text{ fresh}}{\Gamma, H \vdash_W \text{new } x \text{ in } e : \tau / E}$	
$\text{IF} \quad \frac{\Gamma, H_1 \vdash_W e_1 : \tau_1 / E_1 \quad \Gamma, H_2 \vdash_W e_2 : \tau_2 / E_2 \quad \Gamma, H_3 \vdash_W e_3 : \tau_3 / E_3}{\Gamma, H_1; H_2 H_3 \vdash_W \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau / E_1 \wedge E_2 \wedge E_3 \wedge \tau_1 = \text{bool} \wedge \tau_2 = \tau_3}$	
$\text{ABS1} \quad \frac{\Gamma; x : t'; z : t' \xrightarrow{h} t, H \vdash_W e : \tau / E \quad t', t, h \text{ fresh}}{\Gamma, \epsilon \vdash_W \lambda_z x. e : t' \xrightarrow{\mu h. H} t / E \wedge t = \tau}$	$\text{ABS2} \quad \text{FORTHCOMING}$
$\text{APP} \quad \frac{\Gamma, H_1 \vdash_W e_1 : \tau_1 / E_1 \quad \Gamma, H_2 \vdash_W e_2 : \tau_2 / E_2 \quad t, h \text{ fresh}}{\Gamma, H_1; H_2; h \vdash_W e_1 e_2 : t / E_1 \wedge E_2 \wedge \tau_1 = \tau_2 \xrightarrow{h} t}$	
$\text{APPLET} \quad \frac{\Gamma, H_1 \vdash_W e_1 : \nabla \bar{\zeta}. \tau' \xrightarrow{H_3} \tau / E_1 \quad \Gamma, H_2 \vdash_W e_2 : \tau'' / E_2 \quad \bar{\zeta}' \text{ fresh}}{\Gamma, H_1; H_2; H_3[\bar{\zeta}'/\bar{\zeta}] \vdash_W e_1 e_2 : \tau[\bar{\zeta}'/\bar{\zeta}] / E_1 \wedge E_2 \wedge \tau' = \tau''}$	
$\text{LET} \quad \frac{\bar{\alpha} \cap \text{fv}(\Gamma) = \emptyset \quad \Gamma, \epsilon \vdash_W v : \tau / E_v \quad \phi = \text{unify}_\tau(E_v) \quad (\Gamma; x : \forall \bar{\alpha}. \phi(\tau), H \vdash_W e : \tau / E)}{\Gamma, H \vdash_W \text{let } x = v \text{ in } e : \tau / E}$	

Figure 3: Typing rules for λ_{hist} with generative constants

8 Related Work Sketch

Igarashi & Kobayashi Our work has similarities. In their type system their usages U are analogous to our H 's. Their acc expressions are analogous to our ev expressions.

One significant difference is the manner in which assertions/specifications are made. I&K have a *global* form of assertion Φ , specifying the set of traces produced for a particular resource (or function). Our assertions P , on the other hand, are *contextual* assertions made from the perspective of particular program points. What does this mean? On the one hand, it is easy to imagine an encoding of our simplest history calculus in the I&K system. For example, for any P in expression e our system, we can define an expression in the I&K system:

$$\text{let } p = \text{new}^{\Phi}() \text{ in } e'$$

where Φ is “every history string that will satisfy P , followed by the constant check_P ”, and e' is e with every occurrence of P replaced with:

$$\text{acc}^{\text{check}_P}(p)$$

and with any ev replaced with acc^{ev} . However, consider parameterized events and predicates; predicates in our system with this extension can have multiple calling contexts, at any program point where actual parameters are supplied, and it is not clear how one could write a Φ as above which would properly anticipate all parameterized contexts. Thus, it is not clear if an I&K encoding is possible without some extension, and even if it is, it is not clean and natural, while it is in our system. (*obviously need to say more here. – c.e.s.*) Our logic is thus a modal logic of programs of the sort like Dynamic Logic [?] and pre/post-conditions [?], whereas I&K is not. (*I don't understand this last sentence. – c.e.s.*)

I&K have one significant feature we do not: first-class histories that may be dynamically generated, and passed as values. Although this is a nice feature, it is the main feature that forces their theory to be so complex. We believe the combination of a static, declarative method for multiple histories, and parametric events with singleton parameters, leads to a theory nearly as expressive in practice, and which is much, much simpler.

Another significant difference is our function types are effect types, in that they record the event effects of the function body. I&K use the diamond and circle-dot operators to do this.

An interesting formal question is if there is any way where the I&K stuff could encode our stuff. The translation would be highly nontrivial since the “check at this program point” aspect would have to be worked into the translation somehow. We may want to beat on this a bit to solidify the case of our theory being more distant from I&K than some readers may

perceive. It is also a problem of independent theoretical interest. (*relevant comments added above. – c.e.s.*)

Also the reverse embedding. I think this is possible by putting a check P before each acc^l in their expressions that asserts a prefix of Φ^l has been seen up to now. for the function Φ , toss out a new special event when the function is entered, and when it exits, check that the Φ for that function body hold back to the point where the special event was plopped on the strace.

DeLine and Fahndrich Need to contrast with this paper.

Ordered Linear Logic Our system is very different, but our history assumptions are treated as are ordered linear logic assumptions.

GradDude Walker Harper, and previous Walker stuff Seems to only check the current state, not the trace(?) So, like I&K its assertions about only the latest event at run-time.