

History Effects and Verification

Christian Skalka
The University of Vermont

Joint work with Scott Smith, The Johns Hopkins University

Programming Language Safety

Large class of program safety properties expressible as *regular sequences of events*, e.g.:

- File open before file read
- Memory allocation before memory usage
- Access control: privilege activation before privileged action

Well-formedness of event sequences expressible and automatically enforceable via program monitors or checks in program logics.

Static Enforcement of Program Properties

Dynamic well-formedness of event sequences enforceable statically, by a two-phase process:

1. Static *program abstractions* predict possible event sequences (*traces*)
2. Analysis of abstractions ensure well-formedness of event sequences
 - *Temporal logics* allow specification, enforcement of general class of properties

Synopsis: we propose a *type-and-effect* program abstraction, coupled with *model-checking*, to statically enforce properties expressed in temporal logic.

Related Work

Previous related proposals for static enforcement of trace-based properties expressed in temporal logic:

Atsushi Igarashi and Naoki Kobayashi, Resource Usage Analysis, POPL02

K. Marriott, P. J. Stuckey and M. Sulzmann, Resource Usage Verification, APLAS03

- Type-theoretic program abstractions (no polymorphism, global or unspecified program logic)

F. Besson and T. Jensen and D. Le Métayer and T. Thorn, Model checking security properties of control flow graphs, J. Computer Security

- Control flow graph abstraction (not higher-order analysis)

Dynamic Language Model: Basic Abstractions

Our dynamic language model (called λ_{hist}) is based on two fundamental abstractions:

- *Histories of events*
 - Events are explicit records of program actions, inserted by programmer or compiler
 - Histories are a configuration component that maintain ordered sequence of events at run-time
- *History predicates*
 - Local checks in temporal logic, inserted at specific program points

λ_{hist} Language Model: Syntax

Our language model is a core functional calculus, where the grammar of expressions e is endowed with booleans, unit, and:

$c \in \mathcal{C}$ *declarative, first-class constants* $ev_i(e)$ *parameterized events*
 $\phi(e)$ *history predicates*

$\lambda_z y. \lambda x. \text{if } y \text{ then } ev_1(x) \text{ else } (ev_2(x); (z \text{ true } x))$

λ_{hist} Language Model: Operational Semantics

The semantics of λ_{hist} is defined via a small-step reduction relation on *configurations*— pairs of *histories* and expressions:

$$\eta ::= \varepsilon \mid ev(c) \mid \eta; \eta \quad \text{histories} \qquad \eta, e \quad \text{configurations}$$

Histories maintain a temporally ordered record of program events:

$$\begin{aligned} \eta, (\lambda_z x. e)v &\rightarrow \eta, e[v/x][\lambda_z x. e/z] \\ \eta, ev(c) &\rightarrow \eta; ev(c), () \\ &\vdots \end{aligned}$$

λ_{hist} Language Model: Semantics

For example, let:

$$f \triangleq \lambda_z y. \lambda x. \text{if } y \text{ then } ev_1(x) \text{ else } (ev_2(x); (z \text{ true } x))$$

Then we have the following multi-step reductions:

$$\varepsilon, f \text{ false } c \rightarrow^* ev_2(c), f \text{ true } c \rightarrow^* ev_2(c); ev_1(c), ()$$

λ_{hist} Language Model: History Checks

History checks $\phi(c)$ are *temporal logic* formulae (linear μ -calculus), specifying sets of histories (viewed as *traces*).

- Formulae describe *regular* properties of traces.
- Decidable interpretation $\llbracket \phi(c) \rrbracket$ is set of histories that $\phi(c)$ specifies.

The semantics of history checks requires success for progress (otherwise, configurations are *stuck*):

$$\eta, \phi(c) \rightarrow \eta; ev_{\phi}(c), () \quad \text{if } \eta \in \llbracket \phi(c) \rrbracket$$

Approximating Histories

Our static analysis is adapted to this dynamic language model; core intuitions:

- Static analysis is a standard type system, enhanced with *history effects*
- History effects are program abstractions that predict possible event sequences (including checks in context)
- History effects are amenable to verification

Type safety comprises static verification of run-time checks, by (1) correctness of program abstraction and (2) correctness of verification.

History Effects

History effect program abstractions are an independent component of the λ_{hist} type language.

History effect terms comprise forms for approximating dynamic histories:

ε *empty history* $ev(c)$ *single event* $H;H$ *ordered sequence*
 $H|H$ *nondeterministic choice* $\mu h.H$ *recursive history effect*

- Sequencing reflects order of evaluation
- Non-deterministic choice for “may analysis” of e.g. conditionals
- Recursive effects for recursive functions

History Effects are LTSs

History type effects are interpreted as *labeled transition systems* (LTSs; similar to BPAs).

$$a ::= ev(c) \mid \varepsilon \mid \downarrow \quad \text{labels}$$

Strings of labels generated in transition semantics:

$$\begin{array}{cccc} ev(c) \xrightarrow{ev(c)} \varepsilon & H_1|H_2 \xrightarrow{\varepsilon} H_1 & H_1|H_2 \xrightarrow{\varepsilon} H_2 & \mu h.H \xrightarrow{\varepsilon} H[\mu h.H/h] \\ \varepsilon; H \xrightarrow{\varepsilon} H & & H_1; H_2 \xrightarrow{a} H'_1; H_2 \text{ if } H_1 \xrightarrow{a} H'_1 & \end{array}$$

History Effect Approximations

Computation of LTSs generates history traces; i.e.:

trace of $H \xrightarrow{a_1} \dots \xrightarrow{a_n} H_n$ *is* $a_1; \dots ; a_n$

The *interpretation* of an effect H , denoted $\llbracket H \rrbracket$, is the set of traces H may generate.

- Goal of type analysis: given e , to statically obtain H such that $\varepsilon, e \rightarrow^* \eta, e'$ implies $\eta \in \llbracket H \rrbracket$.

Sound history effects of expressions approximate run-time histories, *including checks in context*.

Validity/Verification of Effects

Validity of history effects requires all predicted checks to succeed in predicted context:

Definition 1 *A history H is valid iff:*

$$\eta \in \llbracket \phi(c) \rrbracket \quad \text{for all } \eta; ev_{\phi}(c) \in \llbracket H \rrbracket$$

- Validity of sound program history effects *guarantees success of run-time checks.*
- Validity is decidable via *verification/model checking*

Type and History Effect

Sound history effects are generated as an artifact of type analysis.

Language of types τ comprises *latent history effects* on function types:

bool *boolean type* $\{c\}$ *singleton constant type* $\tau \xrightarrow{H} \tau$ *function types*
 t, α, h *type, singleton, effect variables*

Typing comprises *polymorphism over each variable sort*. Latent effects approximate histories generated by functions:

$$f \triangleq \lambda_z y. \lambda x. \text{if } y \text{ then } ev_1(x) \text{ else } (ev_2(x); (z \text{ true } x))$$

$$f : \forall \alpha. \text{bool} \xrightarrow{\varepsilon} \{\alpha\} \xrightarrow{\mu h. ev_1(\alpha) | (ev_2(\alpha); h)} \text{unit}$$

History Type Judgements

In general, expressions e are assigned types τ and effects H in typing environments Γ , via *derivability* of *type judgements*:

$$\Gamma, H \vdash e : \tau \quad \textit{type judgements}$$

Derivability of type judgements defined inductively by derivation rules for each expression form.

- Effects assigned to expressions approximate their evaluation histories.
- Conservative extension of Hindley-Milner typings.

History Type Derivation Rules (Highlights)

<p>CONST</p> $\frac{}{\Gamma, \varepsilon \vdash c : \{c\}}$	<p>EVENT</p> $\frac{\Gamma, H \vdash e : \{s\}}{\Gamma, H; ev(s) \vdash ev(e) : unit}$	<p>CHECK</p> $\frac{\Gamma, H \vdash e : \{s\}}{\Gamma, H; ev_\phi(s) \vdash \phi(e) : unit}$
<p>ABS</p> $\frac{\Gamma; x : \tau_1; z : \tau_1 \xrightarrow{H} \tau_2, H \vdash e : \tau_2}{\Gamma, \varepsilon \vdash \lambda_z x. e : \tau_1 \xrightarrow{H} \tau_2}$	<p>APP</p> $\frac{\Gamma, H_1 \vdash e_1 : \tau' \xrightarrow{H_3} \tau \quad \Gamma, H_2 \vdash e_2 : \tau'}{\Gamma, H_1; H_2; H_3 \vdash e_1 e_2 : \tau}$	
<p>IF</p> $\frac{\Gamma, H_1 \vdash e_1 : bool \quad \Gamma, H_2 \vdash e_2 : \tau \quad \Gamma, H_3 \vdash e_3 : \tau}{\Gamma, H_1; H_2 H_3 \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}$		

Type Safety

Type safety is demonstrable via a subject reduction argument. A corollary formalizes the assertion that history effects approximate histories:

Corollary 1 *If $\Gamma, H \vdash e : \tau$ and $\varepsilon, e \rightarrow^* \eta, e'$ then $\eta \in \llbracket H \rrbracket$.*

Since class of stuck configurations includes failing checks, success of run-time checks implied by type safety:

Theorem 1 (Type Safety) *If $\Gamma, H \vdash e : \tau$ with H valid, then it is not the case that $\varepsilon, e \rightarrow^* \eta, e$ with η, e stuck.*

Corollary 2 *If $\Gamma, H \vdash e : \tau$ with H valid and $\varepsilon, e \rightarrow^* \eta, \phi(c)$, then $\eta \in \llbracket \phi(c) \rrbracket$.*

History Effect Reconstruction

λ_{hist} type assignments are reconstructable.

Type inference algorithm modification of known techniques for HM type inference; extended with *type equality* and *history containment constraints*:

$$\frac{\text{ABS} \quad \Gamma; x : t, H \vdash_W e : \tau/C, HC \quad h \text{ fresh}}{\Gamma, \varepsilon \vdash_W \lambda x. e : t \xrightarrow{h} \tau/C, HC \cup \{H \sqsubseteq h\}}$$

Unification solves type constraints.

Solving History Effect Constraints

History effect containment solved independently:

- General problem unsolvable, but...
- Inferred constraints define *system of lower bounds* on variables; all constraints of the form:

$$H \sqsubseteq h$$

- Solution obtained by assigning to h the join of all lower bounds (decidable)

Automated Program Analysis

A complete automated analysis obtained by composition of:

- Type constraint inference
- Unification and effect constraint solution
- Verification (model checking) of history effects

Correctness of each component guarantees algorithmic type safety.

Stack-Based Policies

In stack-based security, events occurring during function execution are “forgotten” when the function returns:

- Activations annotated with events; call-stack pop erases events
- Ubiquitous example: *stack inspection*

General temporal logic assertions also applicable to traces defined by sequence of events on call stack.

Stackification

Post-processing of history effects allows abstraction of *stack contexts*:

- Pushes and pops coincide with function scope
- Regularity of push and pop events allows stack contexts to be retrieved from history effects

Single-pass type analysis composed with *stackification* of effects allows enforcement of both stack- and history-based properties.

Conclusion

In summary, our proposed analysis combines the following elements:

- Abstraction of program event sequences via type-and-effect
- Static enforcement of program properties by model-checking of program abstraction
- Type safety guarantees success of dynamic checks, expressed in general temporal logic.

`http://www.cs.uvm.edu/~skalka`