

A Type and Effect System for Flexible Abstract Interpretation of Java

Christian Skalka
The University of Vermont

Joint work with Scott Smith (Johns Hopkins) and David Van Horn (University of Vermont)

Presentation Outline

Synopsis: we propose a type and effect analysis for statically approximating event traces generated by object oriented programs.

Principal topics:

- Fundamentals of OO language model with event traces
- Summary of type and effect reconstruction
- Transformation of effects to treat variations on core language model

Programming Language Safety

Many program safety properties expressible as properties of *program event traces*, e.g.:

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

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

Event Traces

Our language model comprises a primitive notion of *traces* of *events*:

- Events are explicit records of program actions, inserted by programmer or compiler.
- Event traces are a semantic configuration component that maintain ordered sequence of events at run-time.

The fundamental abstraction of event traces provides a framework for analyzing a wide range of program safety properties.

Object Oriented Language Model: FJ_{sec}

Goal: to provide a realistic program analysis for Java.

To address fundamental OO features, we consider Featherweight Java (FJ), extended with events.

`ev[i]` `i` is an identifier

Events annotate program points in FJ code:

```
String format(String text){ ev[Applet];... }
```

Object Oriented Language Model: FJ_{sec}

The machine model incorporates event traces $ev[1]; \dots; ev[n]$, denoted η , in *configurations* η, e .

The run-time semantics of FJ_{sec} are defined via an evaluation relation on configurations. For example (where \star is a unit value):

$$\eta, ev[i] \rightarrow (\eta; ev[i]), \star$$

Congruence rules allow for contextual evaluation, so configuration traces maintain sequence of events as they are encountered in evaluation:

$$\frac{\eta, e \rightarrow \eta', e'}{\eta, e.m(\bar{e}) \rightarrow \eta', e'.m(\bar{e})}$$

Static Trace Approximations for FJ_{sec}

A type and effect reconstruction algorithm statically infers approximations of program event traces.

$$\Gamma, C, H \vdash_W e : T \quad \textit{type judgement}$$

T *type* Γ *type environment* C *type constraints* H *trace effects*

OO model presents unique challenges vs. e.g. functional model:

- Independence of inheritance and events: override should agree in type, but not effect; interaction with dynamic dispatch
- Downcasts: involves a *soft typing* approach (combined static and dynamic checks)

Trace Effects

Trace effect program abstractions are *label transition systems*, representing sets of possible traces:

ε *empty trace* $\text{ev}[c]$ *single event* $H; H$ *ordered sequence*

$H|H$ *nondeterministic choice* $\mu h.H$ *recursive trace effect (for methods)*

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

Trace Effect Approximations

Trace effects are endowed with a labelled transition semantics:

$$H \triangleq \mu h. \text{ev}[1] \mid (\text{ev}[2]; h)$$

$$H \xrightarrow{\varepsilon} \text{ev}[1] \mid (\text{ev}[2]; H) \xrightarrow{\varepsilon} \text{ev}[2]; H \xrightarrow{\text{ev}[2]} H \xrightarrow{\varepsilon} \text{ev}[1] \mid (\text{ev}[2]; H) \xrightarrow{\varepsilon} \text{ev}[1] \xrightarrow{\text{ev}[1]} \varepsilon$$

The *interpretation* of an effect H , denoted $\llbracket H \rrbracket$, is the set of label traces that can be so generated.

Correctness of analysis: If $\Gamma, C, H \vdash_{\mathcal{W}} e : T$ and $\varepsilon, e \rightarrow^* \eta, e'$, then $\eta \in \llbracket H \rrbracket$.

Effect Transformations for Flexibility

Trace effects generated by type reconstruction can be post-processed to analyze variations on the core language.

- Theoretical interest: core analysis adaptable to non-trivial variations
- Uniform analysis for treating variations
- Possibly greater efficiency than composition with “direct” analysis of variations

Analysis of Stack Traces

In stack trace model, events occurring during function execution are “forgotten” when the function returns:

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

Post-processing of history effects allows approximation of *stack traces*:

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

Stackification (Basic Idea)

Note: all methods are assigned a distinct μ -scoped effect in effect reconstruction.

Stackification exploits this characteristic— μ -scope delineates corresponding pushes and pops:

$$\begin{aligned} \mathit{stackify}(\mathit{ev}[\mathbf{i}]; \mathbf{H}) &= \mathit{ev}[\mathbf{i}]; \mathit{stackify}(\mathbf{H}) \\ \mathit{stackify}(\mathbf{H}_1 | \mathbf{H}_2; \mathbf{H}) &= \mathit{stackify}(\mathbf{H}_1; \mathbf{H}) | \mathit{stackify}(\mathbf{H}_2; \mathbf{H}) \\ \mathit{stackify}(\mathbf{h}; \mathbf{H}) &= \mathbf{h} | \mathit{stackify}(\mathbf{H}) \\ \mathit{stackify}((\mu \mathbf{h}. \mathbf{H}_1); \mathbf{H}_2) &= (\mu \mathbf{h}. \mathit{stackify}(\mathbf{H}_1)) | \mathit{stackify}(\mathbf{H}_2) \end{aligned}$$

Example:

$$\mathit{stackify}((\mu \mathbf{h}. \mathit{ev}[1] | (\mathit{ev}[2]; \mathbf{h})); \mathit{ev}[3]) = (\mu \mathbf{h}. \mathit{ev}[1] | (\mathit{ev}[2]; \mathbf{h})) | \mathit{ev}[3]$$

Analysis of Exceptions

The effects of exceptions can be analyzed with the addition of two new *pre-effect* constructs and subsequent transformation.

throw *anonymous exception (and throw pre-effect)*

try{e₁}catch{e₂} *exception handlers* H₁ ↗ H₂ *pre-effect of handlers*

$$\dots, \text{throw} \vdash_W \text{throw} : \dots \quad \frac{\dots, H_1 \vdash_W e_1 : \dots \quad \dots, H_2 \vdash_W e_2 : \dots}{\dots, H_1 \rightharpoonup H_2 \vdash_W \text{try}\{e_1\}\text{catch}\{e_2\} : \dots}$$

Exception Transformation (Basic Idea)

The transformation separates a given trace effect into two sets of paths: paths that can end “safely”, and those that can end in a throw.

$$\begin{aligned} \textit{exnize}(\textit{ev}[i]) &= \{\textit{ev}[i]\}, \emptyset \\ \textit{exnize}(\textit{throw}) &= \emptyset, \{\epsilon\} \\ \textit{exnize}(H_1; H_2) &= \textit{let } s_1, t_1 = \textit{exnize}(H_1) \textit{ in} \\ &\quad \textit{let } s_2, t_2 = \textit{exnize}(H_2) \textit{ in} \\ &\quad \{H_1; H_2 \mid H_1 \in s_1 \textit{ and } H_2 \in s_2\}, \\ &\quad \{H_1; H_2 \mid H_1 \in s_1 \textit{ and } H_2 \in t_2\} \cup t_1 \\ \textit{exnize}(H_1 \uparrow H_2) &= \textit{let } s_1, t_1 = \textit{exnize}(H_1) \textit{ in} \\ &\quad \textit{let } s_2, t_2 = \textit{exnize}(H_2) \textit{ in} \\ &\quad \{H_1; H_2 \mid H_1 \in t_1 \textit{ and } H_2 \in s_2\} \cup s_1, \\ &\quad \{H_1; H_2 \mid H_1 \in t_1 \textit{ and } H_2 \in t_2\} \end{aligned}$$

Note: The μ case complicates matters.

Exception Transformation Examples

$(\text{ev}[1]; \text{ev}[2]; \text{throw}; \text{ev}[3]) \mid (\text{ev}[1]; \text{ev}[3])$

safe: $\text{ev}[1]; \text{ev}[3]$ *throw:* $\text{ev}[1]; \text{ev}[2]$

$\mu\text{h}.\text{throw} \mid \text{ev}[2] \mid (\text{ev}[1]; \text{h})$

safe: $\mu\text{h}.\text{ev}[2] \mid (\text{ev}[1]; \text{h})$ *throw:* $\mu\text{h}.\text{throw} \mid (\text{ev}[1]; \text{h})$

Conclusion

Summary of main points:

- FJ_{sec} is Featherweight Java extended with *event traces*.
- Type and effect analysis reconstructs approximations of program event traces.
- Approximations represented as *trace effects* H, a form of label transition system.
- Trace effects amenable to transformations for treating language variations, including stack traces and exceptions.

A *formally appealing*, *flexible*, and *efficient* analysis for Java program safety.

<http://www.cs.uvm.edu/~skalka>

Type Reconstruction (Full Detail)

$$\begin{array}{l} \text{T-Var} \\ \Gamma, \text{true}, \varepsilon \vdash_W x : \Gamma(x) \end{array}$$

$$\begin{array}{l} \text{T-Field} \\ \Gamma, C, H \vdash_W e : [TC] \quad D f \in \text{fields}(C) \\ \hline \Gamma, C \wedge T <: (f : [XD]), H \vdash_W e.f : [XD] \end{array}$$

$$\begin{array}{l} \text{T-Invk} \\ \Gamma, C, H \vdash_W e : [TC] \quad \Gamma, D, H' \vdash_W \bar{e} : [\bar{S}B] \quad \text{mtype}(m, C) = \bar{D} \rightarrow D \quad \bar{B} <: \bar{D} \\ \hline \Gamma, C \wedge D \wedge T <: (m : [\bar{S}B] \xrightarrow{h} [XD]), H; H'; h \vdash_W e.m(\bar{e}) : [XD] \end{array}$$

$$\begin{array}{l} \text{T-New} \\ \Gamma(C) = \forall \bar{X}[D].T \quad \Gamma, C, H \vdash_W \bar{e} : \bar{S} \quad T.\bar{f} = \bar{T} \quad \text{fields}(C) = \bar{C} \bar{f} \\ \hline \Gamma, C \wedge D[\bar{X}'/\bar{X}] \wedge \bar{S} <: \bar{T}[\bar{X}'/\bar{X}], H \vdash_W \text{new } C(\bar{e}) : T[\bar{X}'/\bar{X}] \end{array}$$

$$\begin{array}{l} \text{T-Event} \\ \Gamma, \text{true}, \text{ev}[i] \vdash_W \text{ev}[i] : \text{Unit} \end{array}$$

$$\begin{array}{l} \text{T-Check} \\ \Gamma, \text{true}, \text{chk}[i] \vdash_W \text{chk}[i] : \text{Unit} \end{array}$$

$$\begin{array}{l} \text{T-Cast} \\ \Gamma, C, H \vdash_W e : T \\ \hline \Gamma, C \wedge T < [XD], H \vdash_W (D)e : [XD] \end{array}$$

$$\begin{array}{l} \text{T-Meth} \\ \Gamma; \bar{x} : \bar{T}, C, H \vdash_W e : S \quad \Gamma(\text{this}).m = \bar{T} \xrightarrow{h} T \quad \text{mbody}(m, C) = \bar{x}.e \\ \hline \Gamma, C \wedge S <: T \wedge H <: h \vdash_W m, C : \bar{T} \xrightarrow{h} S \end{array}$$

$$\begin{array}{l} \text{T-Class} \\ \Gamma; C : T; \text{this} : T, \bar{C} \vdash_W \bar{m} : \bar{T} \quad T = [\bar{f} : \bar{R} \bar{m} : \bar{S} C] \text{ is abstract} \\ \hline \Gamma \vdash_W C : \forall \bar{X}[\bar{C}].[\bar{f} : \bar{R} \bar{m} : \bar{T} C] \end{array}$$

Exnization (Full Detail)

$$\begin{aligned} \text{exnize}(\varepsilon) &= \{\varepsilon\}, \emptyset, \emptyset \\ \text{exnize}(\text{ev}[i]) &= \{\text{ev}[i]\}, \emptyset, \emptyset \\ \text{exnize}(\text{throw}) &= \emptyset, \{\varepsilon\}, \emptyset \\ \text{exnize}(\mathbf{h}) &= \{\mathbf{h}\}, \emptyset, \{\mathbf{h}\} \\ \text{exnize}(\mathbf{H}_1 | \mathbf{H}_2) &= \text{let } s_1, t_1, r_1 = \text{exnize}(\mathbf{H}_1) \text{ in} \\ &\quad \text{let } s_2, t_2, r_2 = \text{exnize}(\mathbf{H}_2) \text{ in} \\ &\quad s_1 \cup s_2, t_1 \cup t_2, r_1 \cup r_2 \\ \text{exnize}(\mathbf{H}_1; \mathbf{H}_2) &= \text{let } s_1, t_1, r_1 = \text{exnize}(\mathbf{H}_1) \text{ in} \\ &\quad \text{let } s_2, t_2, r_2 = \text{exnize}(\mathbf{H}_2) \text{ in} \\ &\quad s_1; s_2, t_1 \cup (s_1; t_2), r_1 \cup (s_1; r_2) \\ \text{exnize}(\mathbf{H}_1 \uparrow \mathbf{H}_2) &= \text{let } s_1, t_1, r_1 = \text{exnize}(\mathbf{H}_1) \text{ in} \\ &\quad \text{let } s_2, t_2, r_2 = \text{exnize}(\mathbf{H}_2) \text{ in} \\ &\quad s_1 \cup (t_1; s_2), t_1; t_2, r_1 \cup (t_1; r_2) \\ \text{exnize}(\mu \mathbf{h}. \mathbf{H}) &= \text{let } s, t, r = \text{exnize}(\mathbf{H}) \text{ in} \\ &\quad \text{let } \mathbf{H}_s = \mu \mathbf{h}. \text{join}(s) \text{ in} \\ &\quad \text{let } r' = \text{map } (\lambda(\mathbf{H}, \mathbf{h}'). \mathbf{H}[\mathbf{H}_s/\mathbf{h}]; \mathbf{h}') r \text{ in} \\ &\quad \text{let } r_{\mathbf{h}} = \text{filter } (\lambda(\mathbf{H}; \mathbf{h}'). \mathbf{h}' = \mathbf{h}) r' \text{ in} \\ &\quad \text{let } t' = \text{map } (\lambda \mathbf{H}. \mu \mathbf{h}. \text{join}(\{\mathbf{H}[\mathbf{H}_s/\mathbf{h}]\} \cup r_{\mathbf{h}})) t \text{ in} \\ &\quad \{\mathbf{H}_s\}, t', r' - r_{\mathbf{h}} \end{aligned}$$

Stackification (Full Detail)

$$\begin{aligned} \mathit{stackify}(\varepsilon) &= \varepsilon \\ \mathit{stackify}(\varepsilon; H) &= \mathit{stackify}(H) \\ \mathit{stackify}(\mathit{ev}[i]; H) &= \mathit{ev}[i]; \mathit{stackify}(H) \\ \mathit{stackify}(h; H) &= h | \mathit{stackify}(H) \\ \mathit{stackify}((\mu h.H_1); H_2) &= (\mu h. \mathit{stackify}(H_1)) | \mathit{stackify}(H_2) \\ \mathit{stackify}((H_1 | H_2); H) &= \mathit{stackify}(H_1; H) | \mathit{stackify}(H_2; H) \\ \mathit{stackify}((H_1; H_2); H_3) &= \mathit{stackify}(H_1; (H_2; H_3)) \\ \mathit{stackify}(H) &= \mathit{stackify}(H; \varepsilon) \end{aligned}$$