Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 131 (2005) 111–124
www.elsevier.com/locate/entcs

A Type and Effect System for Flexible Abstract Interpretation of Java
(Extended Abstract)

Christian Skalka [1]

Department of Computer Science
University of Vermont

Scott Smith [2]

Department of Computer Science
Johns Hopkins University

David Van Horn [3]

Department of Computer Science
University of Vermont

Abstract

This paper describes a flexible type and effect inference system for Featherweight Java (FJ). The effect terms generated by static type and effect inference embody the abstract interpretation of program event sequences. Flexibility in the analysis is obtained by post-processing of inferred effects, allowing a modular adaptation to extensions of the language. Several example transformations are discussed, including how inferred effects can be transformed to reflect the impact of exceptions on FJ control flow.

Keywords: Type analysis, language security, object oriented languages.

[1] Email: `skalka@cs.uvm.edu`
[2] Email: `scott@cs.jhu.edu`
[3] Email: `dvanhorn@cs.uvm.edu`

doi:10.1016/j.entcs.2005.01.027

1 Introduction

A number of authors [2,12,15,18] have recently shown how abstract interpretations of program control flow can be extracted from higher-order programs, via the use of type effect systems. In these works, the type effects predict atomic events, and the order in which they occur. In all of these systems except [12], the effects form simple labeled transition systems (LTSs) on which standard model-checking algorithms can apply to relate abstracted programs with specifications. Such a program analysis allows for automatic static verification of program properties, including resource usage analysis [15] and access control [18].

The terminology we use in [18] is to speak of *histories*, denoted η, which are traces of atomic *events*. Events are records of some program action, explicitly inserted into program code either manually (by the programmer) or automatically (by the compiler). Events are intended to be sufficiently abstract to represent a variety of program actions—e.g. opening a file, access control privilege activation, or entry to or exit from critical regions. Histories maintain the ordered sequences of events that occur during program execution. We define in [18] a type effect system that produces types with *history effects* that are abstract interpretations of program histories. More specifically, history effects are a form of LTS; if a program is statically assigned a history effect, that effect's interpretation conservatively approximates the event history that can be dynamically generated by the program. A simple example of a history effect in our grammar is:

$$\mu \mathtt{h}.\mathtt{ev}[1] | (\mathtt{ev}[2]; \mathtt{h}; \mathtt{ev}[3])$$

This is the effect of a recursive function with a base case generating atomic event $\mathtt{ev}[1]$, and the recursive case generating $\mathtt{ev}[2]$ before the recursive call and $\mathtt{ev}[3]$ after.

1.1 The FJ$_{sec}$ language model

While the aforecited works were developed for core functional languages, in [16] we extended the results of [18] to Featherweight Java (FJ) [13], showing how the same ideas can be generalized to an object-oriented setting, with special care to handle the case of dynamic dispatch. The language model, called FJ$_{\mathrm{sec}}$, is FJ extended with event histories and a logic of program checks.

The syntax of FJ$_{\mathrm{sec}}$ includes events $\mathtt{ev}[\mathtt{i}]$, indexed by identifiers $\mathtt{i}$. The semantics of FJ$_{\mathrm{sec}}$ is defined via a reduction relation $\rightarrow$ on configurations, which are pairs of histories and expressions $(\eta, \mathtt{e})$. The reduction relation specifies that events encountered during execution are placed at the rightmost, or "most recent", end of the history in the current configuration, i.e. (where () is the FJ$_{\mathrm{sec}}$ unit value):

$$\eta, \mathtt{ev}[\mathtt{i}] \rightarrow (\eta; \mathtt{ev}[\mathtt{i}]), ()$$

So for example, if a method `format` is a component of an applet, the beginning of the method can be labeled with an `Applet` event, and so record on the program history that an applet has affected the program control flow when the method is invoked:

$$\mathtt{String\ format}(\mathtt{String\ text})\{\ \mathtt{ev}[\mathtt{Applet}]; \ldots\ \}$$

A type effect analysis is defined in [16], that statically approximates the histories generated by FJ$_{\mathrm{sec}}$ programs.

1.2 Related Work

The idea of using some form of abstract program interpretation as input to model checking [19] for verification of specified program properties has been explored by many authors, including [7,3,5]. In these particular works, the specifications are temporal logics, regular languages, or finite automata, and the abstract control flow is extracted as an LTS in the form of a finite automaton, grammar, or push-down automata. However, none of these works defines a rigorous process for extracting an LTS from higher-order programs.

Perhaps the most closely related work is [15], which proposes a similar type and effect system and type inference algorithm, but their "resource usage" abstraction is of a markedly different character, based on grammars rather than LTSs. Their system also lacks parametric polymorphism, which restricts expressiveness in practice.

The system of [12] is based on linear types, which are related to, but different from, effect types. Their usages U are similar to our history effects H, but the usages have a much more complex grammar, and do not appear amenable to model-checking. The systems in [8,4,14,5] use LTSs extracted from control-flow graph abstractions to model-check program security properties expressed in temporal logic. Their approach is close in several respects, but we are primarily focused on the programming language as opposed to the model-checking side of the problem. Their analyses assume the pre-existence of a control-flow graph abstraction, which is in the format for a first-order program analysis only. Our type-based approach is defined directly at the language level, and type inference provides an explicit, scalable mechanism for extracting an abstract program interpretation, which is applicable to object oriented features. Also, none of this related work considers the transformations discussed in Sect. 3.

1.3 Outline of the paper

In this extended abstract we focus on transformations of $\mathrm{FJ}_{\mathrm{sec}}$ history effects, showing how they can be useful. First, we define a simple stack transformation called *stackification* to generate an abstract interpretation of the possible stack states at runtime. This transformation is particularly useful for security analyses such as Java stack inspection [20]. Then, we show how exceptions can be interpreted by another transformation called *exnization*. A principal benefit of our approach is that a variety of language features can be treated in a modular fashion, without redefinition of type effect inference.

2 Automatic Program Abstraction via Type and Effect

Our program analysis is a type and effect inference system, where effects are approximations of program histories. This approach allows sophisticated type inference techniques to be applied, e.g. constraint-based polymorphic subtyping. This expressiveness has benefits in higher-order functional [18] and object-oriented settings [16]; the latter has a more complete discussion of the type inference system presented here, and its benefits for Java in particular.

2.1 History Effects

The effects component of our type system, called *history effects*, are essentially label transition systems (LTSs), similar to basic process algebras (BPAs) [6]. Label transition systems generate possibly infinite strings called *traces* via a transition relation, wherein transitions are labeled (possibly by the empty string ϵ); sequences of transitions generate traces of labels. A history effect $\mathtt{H}$ may be an event $\mathtt{ev[i]}$, or a sequence of history effects $\mathtt{H_1;H_2}$, a nondeterministic choice of history effects $\mathtt{H_1|H_2}$, or a recursive history effect $\mu\mathtt{h.H}$, where the variable $\mathtt{h}$ is recursively bound in $\mathtt{H}$. So for example, letting $\mathtt{H} \triangleq \mu\mathtt{h.ev[1]} \mid (\mathtt{ev[2];h})$:

$$\mathtt{H} \xrightarrow{\epsilon} \mathtt{ev[1]|(ev[2];H)} \xrightarrow{\epsilon} \mathtt{ev[2];H} \xrightarrow{\mathtt{ev[2]}} \mathtt{H} \xrightarrow{\epsilon} \mathtt{ev[1]|(ev[2];H)} \xrightarrow{\epsilon} \mathtt{ev[1]} \xrightarrow{\mathtt{ev[1]}} \epsilon$$

which yields the trace $(\mathtt{ev[2]\,ev[1]})$. The interpretation of any $\mathtt{H}$, denoted $[\![\mathtt{H}]\!]$, is the set of traces that can be so generated. The effects reconstructed by type inference serve as an approximation of run-time histories, in the sense that if $\mathtt{H}$ is the effect statically assigned to a program $\mathtt{e}$, and $\epsilon, \mathtt{e} \rightarrow^{\star} \eta, \mathtt{e}'$, then $\eta \in [\![\mathtt{H}]\!]$. This property is formalized in Theorem 2.1.

Sound effect approximations in this sense are obtained from programs via type and effect inference, which has three distinct phases: (1) type and effect constraint inference, (2) constraint *closure*, and (3) effect *extraction*, where top-level program effects are obtained as a solution to effect constraints generated by the previous phases. We discuss each of these phases in turn.

2.1.1 Type constraint inference.

Featherweight Java is equipped with a declarative, nominal type system; the type language is based on class names, which annotate function return and argument types, casts, and object creation points. The system is algorithmically checkable, and enjoys a type safety result [13]. Our system is not intended to redo the type system of FJ, but to "superimpose" a type and effect analysis on it, thereby subsuming type safety for the FJ subset of $\mathrm{FJ}_{\mathrm{sec}}$. This superimposition is conservative and transparent to the programmer, both for ease of use, and for backwards compatibility with Java.

Our language of types includes class types $[\bar{\mathtt{T}}\,\mathtt{C}]$, where $\bar{\mathtt{T}}$ is a vector of field and method types in the class, and $\mathtt{C}$ is the class name. Methods types are of the form $\bar{\mathtt{T}} \xrightarrow{\mathtt{H}} \mathtt{T}$, where $\mathtt{H}$ is an approximation of the event histories that a method can generate. The latter may be abstract, so that polymorphism extends to effects. For example, if we assume that $\mathtt{C}$ and $\mathtt{D}$ are defined classes, the latter containing a single nullary method $\mathtt{n}$, and given the following method definition, where we assume the trivial addition of a sequencing construct to the language:

$$\mathtt{C\ m(D\ f)\{ev[1];\ f.n();\ return\ this.m(f)\}}$$

the following polymorphic type can be assigned to $\mathtt{m}$ (where the details of the types $\mathtt{T}$ and $\mathtt{S}$ are irrelevant to the example):

$$\mathtt{m} : \forall \mathtt{h}.[(\mathtt{n} : () \xrightarrow{\mathtt{h}} \mathtt{S})\,\mathtt{D}] \xrightarrow{\mu \mathtt{h}'.\mathtt{ev[1]};\mathtt{h};\mathtt{h}'} [\mathtt{T}\,\mathtt{C}]$$

Polymorphism on method effects is quite useful, since Java allows objects in *subclasses* of $\mathtt{D}$ to be actual parameters of $\mathtt{m}$; without polymorphism, this discipline would require equivalence of effects throughout the inheritance hierarchy, or some sort of behavioral subtyping with regard to events, which is unrealistic. For example, if events represent privilege authorizations, the least privileged member of a class hierarchy would determine the authorization level of every class in the hierarchy, and any extension of the class hierarchy would require re-computation of superclass types. This issue is discussed more thoroughly in [16].

Our inference system reconstructs type and effect constraints of the form $\mathtt{S} <: \mathtt{T}$; constraint sets are represented as conjunctions of these atomic constraints. The relation is defined with respect to the inheritance hierarchy in the same manner as [13], and with respect to histories as a containment relation; in particular, if $\mathtt{H}_1$ and $\mathtt{H}_2$ are closed, then $\mathtt{H}_1 <: \mathtt{H}_2$ iff $[\![\mathtt{H}_1]\!] \subseteq [\![\mathtt{H}_2]\!]$. Constraints on method and class types are defined in a familiar manner [9,16], modulo constraints on the latters' history effect annotations. Typings are reconstructed via an algorithmic derivation systems on judgements of the form $\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{T}$, where Γ is a type environment, $\mathtt{H}$ is the effect of $\mathtt{e}$, $\mathtt{T}$ is the type of e given Γ, and C imposes constraints on type variables in the judgement. Thus, where $\mathit{mtype}(\mathtt{m}, \mathtt{C})$ returns the annotated type of $\mathtt{m}$ in the definition of $\mathtt{C}$, the rule T-Invk reconstructs the type of method invocations:

$$\textsc{T-Invk}\quad \frac{\begin{array}{c}\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : [\mathtt{T}\,\mathtt{C}] \\ \Gamma, D, \mathtt{H}' \vdash_W \bar{\mathtt{e}} : [\mathtt{S}\,\bar{\mathtt{B}}] \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{D}} \to \mathtt{D} \qquad \bar{\mathtt{B}} <: \bar{\mathtt{D}}\end{array}}{\Gamma, C \wedge D \wedge \mathtt{T} <: (\mathtt{m} : [\mathtt{S}\,\bar{\mathtt{B}}] \xrightarrow{\mathtt{h}} [\mathtt{t}\,\mathtt{D}]), \mathtt{H}; \mathtt{H}'; \mathtt{h} \vdash_W \mathtt{e.m}(\bar{\mathtt{e}}) : [\mathtt{t}\,\mathtt{D}]}$$

Note that a left-to-right, call-by-value evaluation order is reflected in the sequencing of effects of subexpressions in the consequent.

2.1.2 Constraint closure and consistency.

In addition to type inference rules, the type implementation comprises a constraint closure algorithm, called *close*, and a consistency check. A constraint is consistent if it contains no contradictions, i.e. it possesses a solution in a regular tree model [16]. Closure normalizes constraints, essentially by "breaking down" given constraints to discover all implicitly represented constraints of "basic" form. For example:

$$\text{given:}\quad \bar{\mathtt{T}} \xrightarrow{\mathtt{H}} \mathtt{T} <: \bar{\mathtt{S}} \xrightarrow{\mathtt{H}'} \mathtt{S} \qquad \text{closure adds:}\quad \bar{\mathtt{S}} <: \bar{\mathtt{T}} \wedge \mathtt{T} <: \mathtt{S} \wedge \mathtt{H} <: \mathtt{H}'$$

Note that in particular, closure explicitly accrues all effect constraints. Once constraints are broken down in this manner, it is straightforward to check consistency.

2.1.3 History effect extraction.

Another benefit of closure, noted above, is that it generates a constraint that is amenable to history effect extraction, since all effect constraints are explicit in closed constraints. In particular, any expression's top-level history effect can be extracted from the constraint system inferred for the expression via the *hextract* algorithm. It is demonstrable that inferred history effect constraints define a system of lower bounds on history effect variables; the *hextract* algorithm exploits this property to obtain a least solution of each constrained

$$\epsilon;\mathtt{H} \longrightarrow_{simp} \mathtt{H} \qquad \mathtt{H};\epsilon \longrightarrow_{simp} \mathtt{H} \qquad \frac{\mathtt{H}_1 \longrightarrow_{simp} \mathtt{H}_1'}{\mathtt{H}_1;\mathtt{H}_2 \longrightarrow_{simp} \mathtt{H}_1';\mathtt{H}_2} \qquad \frac{\mathtt{H}_2 \longrightarrow_{simp} \mathtt{H}_2'}{\mathtt{H}_1;\mathtt{H}_2 \longrightarrow_{simp} \mathtt{H}_1;\mathtt{H}_2'}$$

$$\mathtt{H}|\mathtt{H} \longrightarrow_{simp} \mathtt{H} \qquad \frac{\mathtt{H}_1 \longrightarrow_{simp} \mathtt{H}_1'}{\mathtt{H}_1|\mathtt{H}_2 \longrightarrow_{simp} \mathtt{H}_1'|\mathtt{H}_2} \qquad \frac{\mathtt{H}_2 \longrightarrow_{simp} \mathtt{H}_2'}{\mathtt{H}_1|\mathtt{H}_2 \longrightarrow_{simp} \mathtt{H}_1|\mathtt{H}_2'} \qquad \frac{\mathtt{h} \notin \mathrm{fv}(\mathtt{H})}{\mu\mathtt{h}.\mathtt{H} \longrightarrow_{simp} \mathtt{H}}$$

Fig. 1. Effect simplification rewrite rules

variable, as the join of its lower bounds.

Soundness of the complete automated analysis is then obtained by correctness of each phase; in the current context, our principal result is as follows:

Theorem 2.1 *If* $\epsilon, \mathtt{e} \longrightarrow^\star \eta, \mathtt{e}'$*, and* $\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{T}$ *is derivable with* $\mathit{close}(C)$ *consistent, then* $\eta \in [\![\mathit{hextract}(\mathtt{H}, \mathit{close}(\mathtt{C}))]\!]$.

3 Effect Transformations for Flexibility

In this section we observe that a benefit of our type and effect inference approach is that it yields abstract program interpretations, in the form of history effects, that are amenable to transformational techniques for flexibility of analysis. These transformations can be used to post-process inferred effects, without requiring any reworking of the inference component of analysis. This means that certain extensions of the language can be treated statically in a *modular* fashion.

One obvious transformation is a *simplification* transformation, based on equivalences induced by the interpretation of effects discussed in Sect. 2.1, defined in Fig. 1 as a set of effect rewrite rules. The benefit of this transformation is reduced size and complexity of effects for model checking. More complicated are the *stackification* and *exnization* transformations considered in detail below. Stackification is useful in a stack-based safety context– that is, where events associated with function activations are "forgotten" when that activation returns, as in e.g. Java stack inspection. Exnization implements the impact of exceptions on effect representations.

3.1 Stack-Based Transformations for Security

Rather than consistently accruing events in a history, a stack-based model can be defined where events generated by method activations are associated

R-Invk

$$\frac{\mathit{mbody}(\mathtt{m},\mathtt{C}) = \bar{\mathtt{x}}.\mathtt{e}}{\varsigma, (\mathtt{new\ C}(\bar{\mathtt{v}})).\mathtt{m}(\bar{\mathtt{u}}) \longrightarrow \varsigma :: \epsilon, \cdot[\bar{\mathtt{u}}/\bar{\mathtt{x}}, \mathtt{new\ C}(\bar{\mathtt{v}})/\mathtt{this}]\mathtt{e}\cdot}$$

R-Event

$$\varsigma :: \eta, \mathtt{ev}[\mathtt{i}] \longrightarrow \varsigma :: \eta; \mathtt{ev}[\mathtt{i}], ()$$

R-Pop

$$\varsigma :: \eta, \mathtt{E}[\cdot\mathtt{v}\cdot] \longrightarrow \varsigma, \mathtt{E}[\mathtt{v}]$$

Fig. 2. The stack-based semantics of $\mathrm{FJ}_{\mathrm{stack}}$

with the activation call-stack frame; when the activation is popped, so are the associated events. The Java stack inspection [11] access control mechanism, for example, is based on sequences of events on the call stack. There are other stack-based properties that are of interest in program analysis, and so in general an abstraction of the possible call stacks is a useful property. A stack-based access control model has additionally been combined with a history-based security mechanism in [1], and a static analysis for enforcing general stack-based properties via temporal logic is presented in [5]. Direct type inference for a stack-based security model has been studied previously, e.g. in [17]; however, since security mechanisms such as that proposed in [1] require both a history- and stack-based perspective, we believe that our uniform approach is simpler, hence more efficient, than e.g. combining direct stack- and history-based inference in such a context.

In this section, we observe that a stack-based security model can be statically enforced, not by redefining the inference system discussed in Sect. 2, but by an effect post-processing technique called *stackification*. Stackification takes as input an effect that predicts the history generated by a program, and returns the stack contexts generated by a program. The stack contexts generated by a program are formalized by refiguring the $\mathrm{FJ}_{\mathrm{sec}}$ operational semantics with regard to stacks, rather than histories, yielding the language model we call $\mathrm{FJ}_{\mathrm{stack}}$. Stack contexts maintain a notion of ordering; hence stacks, which we denote ς, are LIFO sequences of histories, and are either **nil** or constructed with a cons operator (::):

$$\varsigma ::= \mathbf{nil} \mid \varsigma :: \eta \qquad \textit{history stacks}$$

The $\mathrm{FJ}_{\mathrm{stack}}$ source language is identical to $\mathrm{FJ}_{\mathrm{sec}}$, except that "framed" expressions $\cdot\mathtt{e}\cdot$ are included, to delimit regions of code associated with a stack frame. Stack frames are associated with activations in keeping with the standard stack-based model. It is also notationally convenient to define a syntactic

$$
\begin{aligned}
\mathit{stackify}(\epsilon) &= \epsilon \\
\mathit{stackify}(\epsilon;\mathtt{H}) &= \mathit{stackify}(\mathtt{H}) \\
\mathit{stackify}(\mathtt{ev}[\mathtt{i}];\mathtt{H}) &= \mathtt{ev}[\mathtt{i}];\mathit{stackify}(\mathtt{H}) \\
\mathit{stackify}(\mathtt{h};\mathtt{H}) &= \mathtt{h}|\mathit{stackify}(\mathtt{H}) \\
\mathit{stackify}((\mu\mathtt{h}.\mathtt{H}_1);\mathtt{H}_2) &= (\mu\mathtt{h}.\mathit{stackify}(\mathtt{H}_1)) \mid \mathit{stackify}(\mathtt{H}_2) \\
\mathit{stackify}((\mathtt{H}_1|\mathtt{H}_2);\mathtt{H}) &= \mathit{stackify}(\mathtt{H}_1;\mathtt{H}) \mid \mathit{stackify}(\mathtt{H}_2;\mathtt{H}) \\
\mathit{stackify}((\mathtt{H}_1;\mathtt{H}_2);\mathtt{H}_3) &= \mathit{stackify}(\mathtt{H}_1;(\mathtt{H}_2;\mathtt{H}_3)) \\
\mathit{stackify}(\mathtt{H}) &= \mathit{stackify}(\mathtt{H};\epsilon)
\end{aligned}
$$

Fig. 3. The *stackify* algorithm

notion of *evaluation contexts*:

$$\mathtt{E} ::= [\,] \mid \mathtt{E.f} \mid \mathtt{E.m}(\bar{\mathtt{e}}) \mid \mathtt{v.m}(\bar{\mathtt{v}},\mathtt{E},\bar{\mathtt{e}}) \mid \mathtt{new}\ \mathtt{C}(\bar{\mathtt{v}},\mathtt{E},\bar{\mathtt{e}}) \mid (\mathtt{C})\mathtt{E} \mid \cdot\mathtt{E}\cdot$$

Thus, in the operational semantics defined in Fig. 2, the rule governing method invocation, R-Invk, will push a new frame on the stack, and delimit the code region associated with the activation. Frames are popped, as in R-Pop, when activations return the result of evaluation. Events are accrued in order within an activation frame, as in R-Event.

Given this model, a static analysis in $\mathrm{FJ}_{\mathrm{stack}}$ will approximate the stack contexts that can be generated during program execution. As mentioned above, we accomplish this by a *stackify* transformation, which takes as input history effects output by $\mathrm{FJ}_{\mathrm{sec}}$ type inference, which is also applicable to $\mathrm{FJ}_{\mathrm{stack}}$ source programs (framed expressions are only generated at run-time). A phenomenon exploited by our transformation is that the scope of inferred method effects is always delimited by a μ-binding. This is because the *hextract* algorithm will resolve any history effect variable h as a μ-bound effect, and every method is assigned a variable h as its effect during inference. In other words, stack "pushes" and "pops" are implicitly recorded during inference as the beginning and end of μ-scope.

This means that *stackify*, defined in Fig. 3, can use the syntax of effects to recognize corresponding pushes and pops. Note that in the transformation of μ-bound effects, any effects $\mathtt{H}_2$ following a μ-bound effect $\mathtt{H}_1$ will be considered as part of a different stack context, since $\mathtt{H}_1$ is associated with an activation that will be pushed and popped before any events predicted by $\mathtt{H}_2$ can occur. Note also that this technique requires that simplifications in Fig. 1 only be applied post-stackification, lest they eliminate μ-bindings.

R-Throw

$$\eta, \mathtt{E}[\mathtt{throw}] \rightarrow \eta, \mathtt{throw}$$

RC-Try

$$\frac{\eta, \mathtt{e}_1 \rightarrow \eta', \mathtt{e}_1'}{\eta, \mathtt{try}\{\mathtt{e}_1\}\mathtt{catch}\{\mathtt{e}_2\} \rightarrow \eta, \mathtt{try}\{\mathtt{e}_1'\}\mathtt{catch}\{\mathtt{e}_2\}}$$

R-Try

$$\frac{\mathtt{v} \neq \mathtt{throw}}{\eta, \mathtt{try}\{\mathtt{v}\}\mathtt{catch}\{\mathtt{e}_2\} \rightarrow \eta, \mathtt{v}}$$

R-Catch

$$\eta, \mathtt{try}\{\mathtt{throw}\}\mathtt{catch}\{\mathtt{e}_2\} \rightarrow \eta, \mathtt{e}_2$$

Fig. 4. Semantics of exceptions

The *stackify* algorithm generally exploits a normal form representation of effects as a sequence $\mathtt{H}_1;\mathtt{H}_2$. The last three clauses use history effect equalities to massage history effects into this normal form. Observe that the range of *stackify* consists of history effects that are all tail-recursive; stacks are therefore finite-state transition systems and more efficient model-checking algorithms are possible for stacks than for general histories [10].

Example 3.1 With a, b, c, d representing arbitrary events (*NB*: the output of *stackify* in this example is simplified and rearranged for readability):

$$\mathit{stackify}(a; (\mu.\mathtt{h}.b; c); (\mu\mathtt{h}.c; (\epsilon|(d; \mathtt{h}; a)))) = a; ((\mu\mathtt{h}.b; c)|(\mu\mathtt{h}.c; (\epsilon|(d; \mathtt{h})|(d; a))))$$

3.2 A Transformation for Exceptions

Exceptions exist in Java, so a realistic application of our approach must account for them. In this section we consider a first approximation of the full exception feature set, where we assume there exists only one anonymous exception in the language. The $\mathrm{FJ}_{\mathrm{sec}}$ language of expressions is extended to include the form `throw` for throwing this anonymous exception, and to include the form $\mathtt{try}\{\mathtt{e}_1\}\mathtt{catch}\{\mathtt{e}_2\}$ for handling thrown exceptions. The expressions $\mathtt{e}_1$ and $\mathtt{e}_2$ are expected to agree in their type annotations. The semantics of $\mathrm{FJ}_{\mathrm{sec}}$ are extended with the rules in Fig. 4, where evaluation contexts $\mathtt{E}$ are as defined in Sect. 2, less the form $\cdot\mathtt{E}\cdot$.

To treat these new language forms in type inference, we introduce two new "pre-history" forms to the language of effects: `throw` to identify control flow points where an exception is thrown, and $\mathtt{H}_1 \Rsh \mathtt{H}_2$ to represent the effect of handlers, where $\mathtt{H}_1$ represents the effect of the `try` clause, and $\mathtt{H}_2$ represents the effect of the `catch` clause. We call these pre-history forms, because our effect transformation will eliminate them by replacing them with a direct representation of their control flow. They are not endowed with an LTS semantics, but merely serve as placeholders for this transformation. We extend the $\mathrm{FJ}_{\mathrm{sec}}$

T-Throw

$$\Gamma, \mathbf{true}, \mathtt{throw} \vdash_W \mathtt{throw} : \mathtt{t}$$

T-TryCatch

$$\frac{\Gamma, C, \mathtt{H}_1 \vdash_W \mathtt{e}_1 : [\mathtt{T}\,\mathtt{C}] \qquad \Gamma, D, \mathtt{H}_2 \vdash_W \mathtt{e}_2 : [\mathtt{S}\,\mathtt{C}]}{\Gamma, C \wedge D \wedge \mathtt{S} <: \mathtt{t} \wedge \mathtt{T} <: \mathtt{t}, \mathtt{H}_1 \Rsh \mathtt{H}_2 \vdash_W \mathtt{try}\{\mathtt{e}_1\}\mathtt{catch}\{\mathtt{e}_2\} : [\mathtt{t}\,\mathtt{C}]}$$

Fig. 5. Type inference rules for exceptions

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

Fig. 6. The exception transformation function *exnize*

type inference system with the rules specified in Fig. 5, to assign these new effect forms to the language extensions for exceptions. All other components of type inference remain unchanged.

In the post-processing of inferred effects, called *exnization*, the impact of exceptions on control flow needs to be discovered. To accomplish this, the exception transformation classifies effect paths into three categories; *safe paths*, *throw paths*, and *precursors*. Safe paths do not encounter a throw, whereas throw paths are terminated by a throw. Precursors are safe paths that end in a variable (tail recursive paths); they may be "pruned back" from other paths (safe, throw, or recursive), since if a recursive call causes a throw, everything after the call with be short-circuited until the first enclosing handler. The idea is that precursors should be joined with throw paths within a μ-binding, since any number of recursive calls can be made before a throw is encountered; this yields μ-bound throw paths. For example, given:

$$\mathtt{H} \triangleq \mu\mathtt{h}.\mathtt{ev}[1]|(\mathtt{ev}[2];\mathtt{h};\mathtt{ev}[3])|(\mathtt{ev}[4];\mathtt{throw};\mathtt{ev}[5])$$

we yield the following safe and throw paths obtained from $\mathtt{H}$:

$$\textit{safe:}\quad \mu\mathtt{h}.\mathtt{ev}[1]|(\mathtt{ev}[2];\mathtt{h};\mathtt{ev}[3]) \qquad \textit{throw:}\quad \mu\mathtt{h}.\mathtt{ev}[4]|(\mathtt{ev}[2];\mathtt{h})$$

An added complication is that recursive calls $\mathtt{h}$ may precede other tail recursions or throws in recursive or throw paths; the analysis replaces these "inner" recursions with the safe recursive call paths of $\mathtt{h}$, to obtain safe preceding ground paths.

In more detail, the algorithm *exnize*, defined in Fig. 6, returns a triple s, t, r, where s are the safe paths, t are the throw paths, and r are the precursors, each represented as history effect sets. The exception transformation is defined via some auxiliary functions, including *map* and *filter* defined as usual (where the latter accepts only values that match the given predicate), as well as a cartesian product operation for sequencing pairs of sets, and a join operation defined on non-empty sets:

$$s_1; s_2 = \{\mathtt{H}_1;\mathtt{H}_2 \mid \mathtt{H}_1 \in s_1 \text{ and } \mathtt{H}_2 \in s_2\} \qquad \begin{aligned} \textit{join}(\{\mathtt{H}\}) &= \mathtt{H} \\ \textit{join}(\{\mathtt{H}\} \cup s) &= \mathtt{H}|\textit{join}(s) \end{aligned}$$

The idea is that at the top-level, the exception transformation is the join of the deduced safe and throw paths; note also that at the top-level, the set of precursors should be empty. Thus, the exception transformation of an effect $\mathtt{H}$ is implemented as:

$$\text{let } s, t, \varnothing = \textit{exnize}(\mathtt{H}) \text{ in } \textit{join}(s \cup t)$$

For brevity, we have excluded a special subcase of *exnize*$(\mu\mathtt{t}.\mathtt{H})$, where the recursive call *exnize*$(\mathtt{H})$ returns s, t, r such that $s = \varnothing$. However, this is the case where every program control path through a function throws an exception, which we believe will be rare, and can be easily dealt with by modifying the case where s is not empty.

References

[1] Abadi, M. and C. Fournet, *Access control based on execution history*, in: *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS'03)*, 2003.

[2] Amtoft, T., F. Nielson and H. R. Nielson, "Type and Effect Systems," Imperial College Press, 1999.

[3] Ball, T. and S. K. Rajamani, *Bebop: A symbolic model checker for boolean programs*, in: *SPIN*, 2000, pp. 113–130.

[4] Besson, F., T. de Grenier de Latour and T. Jensen, *Secure calling contexts for stack inspection*, in: *Proceedings of the Fourth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'02)* (2002), pp. 76–87.

[5] Besson, F., T. Jensen, D. L. Métayer and T. Thorn, *Model checking security properties of control flow graphs*, J. Computer Security **9** (2001), pp. 217–250.

[6] Burkart, O., D. Caucal, F. Moller, and B. Steffen, *Verification on infinite structures*, in: J. Bergstra, A. Pons and S. Smolka, editors, *Handbook on Process Algebra*, North-Holland, 2001 .

[7] Chen, H. and D. Wagner, *MOPS: an infrastructure for examining security properties of software*, in: *Proceedings of the 9th ACM Conference on Computer and Communications Security*, Washington, DC, 2002, pp. 235–244.

[8] Colcombet, T. and P. Fradet, *Enforcing trace properties by program transformation*, in: *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2000, pp. 54–66, citeseer.ist.psu.edu/article/colcombet00enforcing.html.

[9] Eifrig, J., S. Smith and V. Trifonov, *Type inference for recursively constrained types and its application to OOP*, Electronic Notes in Theoretical Computer Science **1**, 1995.

[10] Esparza, J., A. Kucera and S. Schwoon, *Model-checking LTL with regular valuations for pushdown systems*, in: *TACS: 4th International Conference on Theoretical Aspects of Computer Software*, 2001.

[11] Gong, L., M. Mueller, H. Prafullchandra and R. Schemers, *Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2*, in: *USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, 1997, pp. 103–112.

[12] Igarashi, A. and N. Kobayashi, *Resource usage analysis*, in: *Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, Oregon, 2002, pp. 331–342.

[13] Igarashi, A., B. C. Pierce and P. Wadler, *Featherweight Java: a minimal core calculus for Java and GJ*, ACM Trans. Program. Lang. Syst. **23** (2001), pp. 396–450.

[14] Jensen, T., D. L. Métayer and T. Thorn, *Verification of control flow based security properties*, in: *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.

[15] K. Marriott, P. J. S. and M. Sulzmann, *Resource usage verification*, in: *Proc. of First Asian Programming Languages Symposium, APLAS 2003*, 2003.

[16] Skalka, C., *Trace effects and object orientation* (2005), submitted. http://www.cs.uvm.edu/~skalka/skalka-pubs/skalka-fjsec.ps.

[17] Skalka, C. and S. Smith, *Static enforcement of security with types*, in: *Proceedings of the the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, Montréal, Canada, 2000, pp. 34–45.

[18] Skalka, C. and S. Smith, *History effects and verification*, in: *Asian Programming Languages Symposium*, number 3302 in Lecture Notes in Computer Science (2004).

[19] Steffen, B. and O. Burkart, *Model checking for context-free processes*, in: *CONCUR'92, Stony Brook (NY)*, Lecture Notes in Computer Science (LNCS) **630** (1992), pp. 123–137.

[20] Wallach, D. S. and E. Felten, *Understanding Java stack inspection*, in: *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, 1998.