

# Trace Effects and Object Orientation

Christian Skalka

Department of Computer Science

The University of Vermont

## Trace Based Program Properties

Large class of program safety and security properties expressible as *regular sequences of events*:

- File usage protocols (open before read or write)
- Memory allocation before memory usage
- Access control: privilege activation before privileged action

A variety of proposed enforcement mechanisms exists: program monitors, security automata, checks in program logics.

# Trace Based Program Properties

*Trace based* security models provide a general linguistic framework for enforcing program properties.

Language model is based on two fundamental abstractions:

- *Traces of events*
  - Events are explicit records of program actions, inserted by programmer or compiler
  - Traces are a component of the run-time system maintaining ordered sequence of events as they're encountered
- *Trace assertions*
  - Local or global assertions expressible in temporal logic

## Static Enforcement of Trace Properties

Trace based properties can be enforced statically, by a two-phase process:

1. Static *program abstractions* conservatively approximate dynamic program traces
2. Automatic analysis of program abstractions allows verification of temporal logic-specified trace properties

We propose a *type and effect* analysis to construct program abstractions, coupled with *model checking* for property verification.

## 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*

*Christian Skalka and Scott Smith, History Effects and Verification, APLAS04*

- Type-theoretic program abstractions for core functional language

*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)

## Trace Properties and Object Orientation

Goal of current research: to lift static analysis of trace based properties to an Object Oriented language model.

To address fundamental OO features, we consider *Featherweight Java* (FJ), extended with events, yielding *FJ<sub>sec</sub>*:

`ev[i]`     `i` is an identifier

Events annotate program points in FJ<sub>sec</sub> code:

```
class Writer extends Object {
  void write(Formatter x, File f){
    ev[writeInvoked];
    String s = x.format()
    fwrite(s,f);
  }
}
```

## Object Oriented Language Model: FJ<sub>sec</sub>

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

The run-time semantics of FJ<sub>sec</sub> 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 $FJ_{\text{sec}}$

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*       $\Gamma$  *type environment*       $C$  *type constraints*       $H$  *trace effects*

- Trace effects  $H$  approximate program traces
- Object type forms  $[TC]$  combine nominal typing and higher-order effects:
  - $C$  is object class name
  - $T$  lists object method types  $\bar{T} \xrightarrow{H} S$  with their *latent effects*

## Trace Effects

Trace effect program abstractions are *label transition systems*, similar to basic process algebras:

$\varepsilon$  *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, where labels are events.

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

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

$$\llbracket H \rrbracket = \{ \text{ev}[1], \text{ev}[2] \text{ev}[1], \text{ev}[2] \text{ev}[2] \text{ev}[1], \dots \}$$

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

## Trace Effects and Object Orientation

Trace effect analysis is significantly complicated by interaction with Object Oriented features, addressed by trace effect analysis:

- Inheritance, override, and dynamic dispatch
  - Parametric *effect polymorphism* allows flexibility for analysis of dynamically dispatched methods.
- Subtyping, object downcasts
  - *Constraint type representation* allows selective “pruning” of constraint graph for sound analysis

## Trace Effects and Dynamic Dispatch

To illustrate, consider the example of *history-based access control* \*.

Code is signed by its *owner*, and access control lists associate owners with their authorized *privileges*.

P code owners/signers

R privileges

$\mathcal{A}$  access control lists

An access control check `demand(R)` ensures that all code affecting control flow so far is signed by owners authorized for R.

`demand` access control check

\* *Martín Abadi and Cédric Fournet. Access Control Based on Execution History, NDSS03.*

## Trace Effects and Dynamic Dispatch

If  $P_1, \dots, P_n$  are the sequence of owners signing code affecting the flow of control:

$$\text{demand}(R) \iff R \in \mathcal{A}(P_1) \cap \dots \cap \mathcal{A}(P_n)$$

In  $\text{FJ}_{\text{SEC}}$ , this model can be implemented by annotating methods with an initial *code signing event*, and demand defined in temporal logic.

For example, imagine:

$$\begin{aligned}\mathcal{A}(\text{System}) &= \{\text{FileWrite}, \text{FileRead}, \dots\} \\ \mathcal{A}(\text{Applet}) &= \emptyset\end{aligned}$$

## Trace Effects and Dynamic Dispatch

Continuing the example, imagine:

```
class Writer extends Object {  
    void safewrite(Formatter x, File f){  
        System;  
        String s = x.format()  
        demand(FileWrite);  
        fwrite(s,f);  
    }  
}
```

At run-time:

- The check will succeed only if the dispatched version of `Formatter.format` is owned by a principal authorized for `FileWrite`.
- If `x` is a `System`-owned object, the check succeeds. If `x` is an `Applet`-owned object, the check fails.

## Trace Effects and Dynamic Dispatch

The method `safewrite` can be statically assigned the following effect, where `H` is *the effect of `x.format`*:

```
System;H;demand(FileWrite)
```

The problem is, what is `H`?

- In the presence of dynamic dispatch, `x.format` could be any version of `format`.
- As a first approximation, `H` could be the non-deterministic choice of the effects of all versions of `format`.

## Trace Effects and Dynamic Dispatch

*But*, imagining:

```
class Formatter extends Object {  
    String format(){ System; ... }  
}
```

```
class AppFormatter extends Formatter {  
    String format(){ Applet; ... }  
}
```

This implies  $H = (\text{System}|\text{Applet})$ :

```
System; (System|Applet); demand(FileWrite)
```

Thus, *any invocation* of `Writer.safeWrite` will be *statically rejected*.

## Trace Effects and Dynamic Dispatch

The “join of all version effects” approach has the following fatal flaws:

- Authorization levels of dynamically dispatched methods will be determined by the least authorized version in the inheritance hierarchy.
- All code must be known in advance, precluding modularity.

*Our solution:* exploit parametric polymorphism. Assign abstract effects  $h$  to dynamically dispatched methods.

## Polymorphic Effects For Dynamic Dispatch

```
class Writer extends Object {  
  void safewrite(Formatter x, File f){  
    System;  
    String s = x.format()  
    demand(FileWrite);  
    write(s,f);  
  }  
}
```

The type assigned to `x.format` is abstract in its effect:

$$x.format : () \xrightarrow{h} \text{StringT}$$
$$\text{Writer.safewrite} : \forall h. (...) \xrightarrow{\text{System};h;\text{demand}(\text{FileWrite})} \text{void}$$

If `Writer.safewrite` is invoked on a `Formatter` object:

`System;System;demand(FileWrite)`

If `Writer.safewrite` is invoked on a `AppFormatter` object:

`System;Applet;demand(FileWrite)`

## Subtyping and Downcasts

Suppose  $\text{Triangle} <: \text{Polygon}$  and  $e : T$ , where  $e$  may evaluate to either a `Triangle` or `Polygon`, as reflected in constraints on  $T$ :

$$[\text{S Polygon}] <: T \qquad [\text{R Triangle}] <: T$$

Cast checking ensures that a *downcast* of the form  $(\text{Triangle})e$  will be stuck if  $e$  does not evaluate to a `Triangle`.

Suppose:

$$(\text{Triangle})e : [T' \text{Triangle}]$$

Type soundness does not allow  $T <: [T' \text{Triangle}]$ , but only *selective* flow from  $T$  to  $[T' \text{Triangle}]$ .

## Soft Subtyping for Downcasts

Our solution is to impose a *soft subtyping* relation between  $T$  and  $[T' \text{ Triangle}]$ :

$$T \triangleleft [T' \text{ Triangle}]$$

Recalling:

$$[S \text{ Polygon}] \triangleleft : T \qquad [R \text{ Triangle}] \triangleleft : T$$

The relation  $T \triangleleft [T' \text{ Triangle}]$  entails  $R \triangleleft : T'$ , *not*  $S \triangleleft : T'$ .

- Constraint representation of types allows “selective pruning” of constraint graph for soft subtyping.
- Soft subtyping also allows for greater precision in analysis, ignoring program flow that would cause cast checking failure at run-time.

## Conclusion

Research focus: static analysis of trace based properties of Object Oriented programs.

- Type based trace effect analysis allows automatic approximation of program trace behavior, via type and effect inference.
- Application of particular type features allow precision and flexibility in static analysis of OO programs:
  - Polymorphism allows effect abstraction of dynamically dispatched methods.
  - Constraint type representation implements object subtyping, allows soft subtyping for sound analysis of downcasts.

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