Algorithmic Trace Effect Analysis

Context Based Security in Programming Languages

Computer Science Research Day, University of Vermont
26 August 2005

David Van Horn
<dvanhorn@cs.uvm.edu>
http://www.cs.uvm.edu/~dvanhorn/

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

ignore

Outline

- Example of temporal correctness: SSL handshakes
- Overview of trace effect analysis
  - Dynamic enforcement
  - Static program logic
  - Proof discovery algorithm
  - Transformations for flexible extension
- Results
- Review

Program correctness as temporal well-formedness

Many program correctness properties are expressible as properties of *program event traces*.

Trace Effect Analysis allows for the static verification of temporal properties of higher-order programs.

Example: Secure Socket Layer (SSL)

For a program sending and receiving data over an SSL socket, e.g. a web browser that supports `https`, the relevant events are opening and closing of sockets, and reading and writing of data packets.

An example event trace produced by a program run could be:

```
ssl_open("snork.cs.jhu.edu",socket_1);
ssl_hs_begin(socket_1);
ssl_hs_success(socket_1);
ssl_put(socket_1);
ssl_get(socket_1);
ssl_open("moo.cs.uvm.edu",socket_2);
ssl_hs_begin(socket_1);
ssl_put(socket_2);
ssl_close(socket_1);
ssl_close(socket_2)
```

Detecting security violations in SSL

Event traces can then be used to detect logical flaws or security violations. For SSL, sockets must first be opened, handshake begun, handshake success, and only then can data be get/put over the socket.

The above trace is illegal because data is put on `socket_2` before notification has been received that handshake was successful on that socket.

Fundamental abstraction: event traces

Many program correctness properties are expressible as properties of *program event traces*.

- Security handshake protocols, eg. SSL
- File open before read
- Allocate before use
- Access control: privilege activation before privileged action

Well-formedness of traces expressible and enforceable as program monitors or checks in program logics, i.e. at runtime.

Fundamental abstraction: event traces

We take a *language-based approach*, integrated the necessary abstractions into a programming language λ_{trace} so that a programmer can articulate temporal properties.

The language is endowed with notions of *events*, and *checks*.

- An *event* is an abstract program action, parameterized by a static constant. They are inserted by the programmer, or compiler.

- A *check* is a predicate, expressed in a temporal logic, over possibly inifinite sequences of events called a *trace*.

Enforcing well-formedness of traces (dynamic)

Event traces are a semantic configuration component that maintain order of events at run-time.

$$\eta ::= \epsilon \mid ev(c) \mid \eta ; \eta$$

Program evaluation is defined as a small-step reduction relation on a pair consisting of an event trace η and a program expression.

$$\begin{array}{rcll}
\eta, (\lambda_z x.e)v & \to & \eta, e[v/x][\lambda_z x.e/z] & \\
\eta, \neg\mathsf{true} & \to & \eta, \mathsf{false} & \\
\eta, \mathsf{if}\,\mathsf{true}\,\mathsf{then}\, e_1\,\mathsf{else}\, e_2 & \to & \eta, e_1 & \\
\eta, ev(c) & \to & \eta ; ev(c), () & \\
\eta, \phi(c) & \to & \eta ; ev_\phi(c), () & \text{if } \Pi(\phi(c), \widehat{\eta}\, ev_\phi(c)) \\
\eta, E[e] & \to & \eta', E[e'] & \text{if } \eta, e \to \eta', e'
\end{array}$$

Type analysis and model checking

Our approach is a synthesis of software verification methods. We use a type analysis with a rich notion of program safety to represent program abstractions. The abstractions are then model checked for verfication.

A polymorphic type and effect inference system automatically extracts a program abstraction conservatively approximating the events and assertions that will arise at run-time. Such an abstraction can then be model-checked to obtain a static verification of these temporal program logics for higher-order programs.

Static analysis

We have developed a program logic, aka type system, such that if the program has a proof in the system (ie. it is a well typed program), all inserted checks will succeed.

Our approach uses a *type and effect* program abstraction, coupled with *model-checking* to enforce temporal properties statically.

Benefits of a static analysis

Static enforcement of temporal specifications leads to

- Formal guarantees about the behaviour of *all possible program executions*
- Earlier error detection (compile-time v. run-time)
- The elimination of all run time checks and maintainence of trace information during executiion.

Static Approximation

We now turn to the problem of approximating the set of possible traces a program may have.

We use a *trace effect* to approximate a trace:

$$H ::= \epsilon \mid ev(c) \mid H;H \mid H|H \mid \mu h.H$$

Trace effect are interpreted as non-deterministic programming language or *labeled transition system*. The interpretation of an effect H, denoted $[\![H]\!]$, is the set of traces H may generate.

We now present a type system which includes the trace effect as part of the type.

Logical typing rules

Event

$$\frac{\Gamma, H \vdash e : \{s\}}{\Gamma, H; ev(s) \vdash ev(e) : \mathsf{unit}}$$

Weaken

$$\frac{\Gamma, H \vdash e : \tau \qquad H \preccurlyeq H'}{\Gamma, H' \vdash e : \tau}$$

If

$$\frac{\Gamma, H_1 \vdash e_1 : \mathsf{bool} \qquad \Gamma, H_2 \vdash e_2 : \tau \qquad \Gamma, H_2 \vdash e_3 : \tau}{\Gamma, H_1; H_2 \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : \tau}$$

Abs

$$\frac{\Gamma; x : \tau_1; z : \tau_1 \xrightarrow{H} \tau_2, H \vdash e : \tau_2}{\Gamma, \epsilon \vdash \lambda_z x.e : \tau_1 \xrightarrow{H} \tau_2}$$

App

$$\frac{\Gamma, H_1 \vdash e_1 : \tau' \xrightarrow{H_3} \tau \qquad \Gamma, H_2 \vdash e_2 : \tau'}{\Gamma, H_1; H_2; H_3 \vdash e_1 e_2 : \tau}$$

Let

$$\frac{\Gamma, \epsilon \vdash v : \tau' \qquad \bar{\alpha} \cap \mathsf{fv}(\Gamma) = \varnothing \qquad \Gamma; x : \forall\bar{\alpha}.\tau', H \vdash e : \tau}{\Gamma, H \vdash \mathsf{let}\ \ x = v\, \mathsf{in}\, e : \tau}$$

Inference

In addition to the logical system of Trace Effect Analysis, we have developed and implemented a proof discovery algorithm that given a program will automatically infer a proof that the program meets its specification.

The algorithm works by traversing the program to accumulate a set of constraints that must be satisfied for the program to be well-typed. Once these constraints are accumulated they are solved using unification and a novel algorithm for computing solutions of effect constraints.

Effect transformations for Flexibility

Trace effects can be post-processed to analyze variations to the core language.

- Simplification

  Traces may be simplified in a semantic-preserving way in order to improve model checking efficiency.

- Stack-based analysis

  In a stack trace model, event occuing during function execution are forgotten when the function returns. Function activations annotated with events; function return erases event.

- Exceptions

  "Pre-effect" constructs allow us to add exceptions to the language with a trivial extension to the algorithm.

Results

- Formal proof establishing correctness of logical system
- Formal proof establishing correctness of transformation algorithms
- Formal proof establishing soundness of algorithm
- Prototype implementation of analysis
- Prototype implementation of transformation algorithms
- Extension of analysis to Java

Publications

- David Van Horn. *Algorithmic Trace Effect Analysis*. Masters Thesis, University of Vermont, 2005.
- Christian Skalka. Trace Effects and Object Orientation. In *Proceedings of the ACM Conference on Principles and Practice of Declarative Programming*, 2005.
- Christian Skalka, Scott Smith, and David Van Horn. A Type and Effect System for Flexible Abstract Interpretation of Java. In *Proceedings of the ACM Workshop on Abstract Interpretation of Object Oriented Languages*, Electronic Notes in Theoretical Computer Science, January 2005.
- Christian Skalka and Scott Smith. History Effects and Verification. In *Asian Programming Languages Symposium*, November 2004.

Review

Many program correctness properties are expressible as properties of *program event traces*.

Trace Effect Analysis allows for the static verification of temporal properties of higher-order programs.

Algorithmic Trace Effect Analysis allows for the automatic, static verification of these properties.

Inferred traces can be transformed to obtain language and analysis extensibility.

The End

Thank you.