Algorithmic Trace Effect Analysis

Masters thesis defense
University of Vermont
29 March 2006

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

ignore

Slogan for today's talk

Trace effect analysis can be automated soundly.

Slogan for today's talk

Trace effect analysis can be automated soundly.

- *Trace effect analysis* — Present and recall analysis and give context for the contributions of the system.

Slogan for today's talk

Trace effect analysis can be automated soundly.

- *Trace effect analysis* — Present and recall analysis and give context for the contributions of the system.

- *Automation* — Show an algorithm for performing the analysis, provide an implementation.

Slogan for today's talk

Trace effect analysis can be automated soundly.

- *Trace effect analysis* — Present and recall analysis and give context for the contributions of the system.

- *Automation* — Show an algorithm for performing the analysis, provide an implementation.

- *Soundness* — Prove safety result stating programs accepted by the algorithm meet their temporal specification.

Main contributions of thesis

Trace effect analysis can be automated soundly.

- Algorithmic safety proof

- Prototype implementation

Outline - Part I: Overview

- Introduction to Trace effect analysis
- Approach of Algorithmic trace effect analysis

Outline - Part II: Gritty details

- Language model λ_{trace}
- Logical system
- Algorithmic system
- Soundness proof
- Implementation
- Digressions
- Conclusion

Introduction to Trace effect analysis

- Example: SSL protocol

- Program correctness as temporal well-formedness

- Language-based Approach

- Static Analysis

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)
```

Correctness as temporal well-formedness

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

Trace effect analysis is 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*.

Benefits of a static analysis

The program logic, aka type system, is designed such that if the program is well-typed, then all inserted checks will succeed.

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.

Approach of Algorithmic trace effect analysis

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 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.

Part II: Gritty Details

Language model λ_{trace}

- Syntax
- Semantics (enforcing trace properties dynamically)
- Stuck expressions
- Operational semantics example

Language syntax

constants

$c \in \mathcal{C}$

booleans

$b ::= \mathsf{true} \mid \mathsf{false}$

values

$v ::= x \mid \lambda_z x.e \mid c \mid b \mid \neg \mid \vee \mid \wedge \mid ()$

expressions

$e ::= v \mid e\, e \mid ev(e) \mid \phi(e) \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \mid \mathsf{let}\ x = v\, \mathsf{in}\, e$

traces

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

evaluation contexts

$E ::= [\,] \mid v\, E \mid E\, e \mid ev(E) \mid \phi(E) \mid \mathsf{if}\, E\, \mathsf{then}\, e\, \mathsf{else}\, e$

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.

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

Stuck expressions

Definition 1 *A configuration* η, e *is* stuck *iff* e *is not a value and there does not exist* η' *and* e' *such that* $\eta, e \rightarrow \eta', e'$*. If* $\epsilon, e \rightarrow^{\star} \eta, e'$ *and* η, e' *is stuck, then* e *is said to* go wrong.

Operational semantics example

Example 1

$$f \triangleq \lambda_z x.\mathsf{if}\, x \,\mathsf{then}\, ev_1(c) \,\mathsf{else}\, (ev_2(c); z(\mathsf{true}))$$

In the operational semantics:

$$\epsilon, f(\mathsf{false}) \to^\star ev_2(c); ev_1(c), ()$$

$$\begin{array}{rcl}
\epsilon, f(\mathsf{false}) & \to & \epsilon, \mathsf{if}\,\mathsf{false}\,\mathsf{then}\, ev_1(c) \,\mathsf{else}\, (ev_2(c); f(\mathsf{true})) \\
& \to & \epsilon, ev_2(c); f(\mathsf{true}) \\
& \to & ev_2(c), f(\mathsf{true}) \\
& \to & ev_2(c), \mathsf{if}\,\mathsf{true}\,\mathsf{then}\, ev_1(c) \,\mathsf{else}\, (ev_2(c); f(\mathsf{true})) \\
& \to & ev_2(c), ev_1(c) \\
& \to & ev_2(c); ev_1(c), ()
\end{array}$$

Logical system

- Static approximations of traces
- Trace effect interpretation
- Type syntax
- Typing rules
- Trace approximation and Type safety

Static approximation of traces

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.

Trace effect interpretation

Definition 2 *The interpretation of trace effects is defined via strings, possibly terminated by* $\downarrow$*, (called* traces*) denoted* θ*, over the following alphabet:*

$$\begin{aligned} s &::= ev(c) \mid \epsilon \mid s\ s \\ a &::= s \mid s\downarrow \end{aligned}$$

Definition 3 (Trace effect transition relation)

$$ev(c) \xrightarrow{ev(c)} \epsilon \qquad H_1|H_2 \xrightarrow{\epsilon} H_1 \qquad H_1|H_2 \xrightarrow{\epsilon} H_2$$

$$\mu h.H \xrightarrow{\epsilon} H[\mu h.H/h] \qquad \epsilon; H \xrightarrow{\epsilon} H \qquad H_1; H_2 \xrightarrow{a} H_1'; H_2 \text{ if } H_1 \xrightarrow{a} H_1'$$

Definition 4 (Trace effect interpretation)

$$[\![H]\!] = \{a_1 \cdots a_n \mid H \xrightarrow{a_1} \cdots \xrightarrow{a_n} H'\} \cup \{a_1 \cdots a_n\!\downarrow \mid H \xrightarrow{a_1} \cdots \xrightarrow{a_n} \epsilon\}$$

Definition 5 *A trace effect H is* valid *iff for all $\theta\, ev_\phi(c) \in [\![H]\!]$ it is the case that:*

$$\Pi(\phi(c), \theta\, ev_\phi(c))$$

holds.

We now turn to a type system for λ_{trace} that incorporates trace effects into the type language.

Type syntax

$\delta \in \mathcal{V}_s, t \in \mathcal{V}_\tau, h \in \mathcal{V}_H, \alpha, \beta \in \mathcal{V}_s \cup \mathcal{V}_\tau \cup \mathcal{V}_H$			*variables*
s	$::=$	$\delta \mid c$	*singletons*
τ	$::=$	$t \mid \{s\} \mid \tau \xrightarrow{H} \tau \mid$ *bool* $\mid$ *unit* $\mid s \mid H$	*types*
σ	$::=$	$\forall \bar{\alpha}.\tau$	*type schemes*
H	$::=$	$\epsilon \mid h \mid ev(s) \mid H; H \mid H \vert H \mid \mu h.H$	*trace effects*
Γ	$::=$	$\varnothing \mid \Gamma; x : \sigma$	*type environments*

fv(τ) denotes the set of free variables in τ.

Logical typing rules

Event

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

Weaken

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

If

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

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

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

Weakening

Weakening relies on trace effect containment relation:

Definition 6 (Trace effect containment) *$H \preccurlyeq H'$ iff $[\![\rho(H)]\!] \subseteq [\![\rho(H')]\!]$ for all interpretations ρ.*

Where ρ is any mapping of effect variables to closed effects.

Example 2

$$\text{if}\, x\, \text{then}\, ev(c_1)\, \text{else}\, ev(c_2)$$

Trace approximation and Logical type safety

Theorem 1 (Trace approximation) *If* $\Gamma, H \vdash e : \tau$ *is derivable for closed* e *and* $\epsilon, e \rightarrow^\star \eta, e'$ *then* $\widehat{\eta} \in [\![H]\!]$.

Definition 7 *A type judgment* $\Gamma, H \vdash e : \tau$ *is* valid *iff it is derivable and* H *is valid.*

Theorem 2 (Type safety) *If* $\Gamma, H \vdash e : \tau$ *is valid for closed* e *then* e *does not go wrong.*

Algorithmic system

- Type and effect constraints
- Algorithmic typing rules
- Relating algorithmic and logical judgements
- Constraint solution algorithm

Type and effect constraints

C	$::=$	$\mathbf{true} \mid \tau \sqsubseteq \tau \mid C \wedge C$	*type and effect constraints*
k	$::=$	τ / C	*constrained types*
ς	$::=$	$\forall \bar{\alpha}.k$	*constrained type schemes*

Judgements:

$$\Gamma, H \vdash_{\mathcal{V}} e : \tau / C$$

Algorithmic rules

Var

$$\frac{\Gamma(x) = \forall\bar{\alpha}.k}{\Gamma, \epsilon \vdash_{\bar{\alpha}} x : k[\bar{\alpha}'/\bar{\alpha}]}$$

Const

$$\frac{}{\Gamma, \epsilon \vdash_{\varnothing} c : \{c\}/\mathbf{true}}$$

Event

$$\frac{\Gamma, H \vdash_{\mathcal{V}} e : \tau/C}{\Gamma, H; ev(\delta) \vdash_{\mathcal{V}\cup\{\delta\}} ev(e) : \mathit{unit}/C \wedge \tau \sqsubseteq \{\delta\}}$$

Check

$$\frac{\Gamma, H \vdash_{\mathcal{V}} e : \tau/C}{\Gamma, H; ev_{\phi}(\delta) \vdash_{\mathcal{V}\cup\{\delta\}} \phi(e) : \mathit{unit}/C \wedge \tau \sqsubseteq \{\delta\}}$$

If

$$\frac{\begin{array}{c}\Gamma, H_1 \vdash_{\mathcal{V}_1} e_1 : \tau_1/C_1 \\ \Gamma, H_2 \vdash_{\mathcal{V}_2} e_2 : \tau_2/C_2 \qquad \Gamma, H_3 \vdash_{\mathcal{V}_3} e_3 : \tau_3/C_3 \qquad \mathcal{V}_1 \sharp \mathcal{V}_2 \sharp \mathcal{V}_3\end{array}}{\Gamma, H_1; H_2|H_3 \vdash_{\mathcal{V}_1\cup\mathcal{V}_2\cup\mathcal{V}_3\cup\{t\}} \text{if}\, e_1 \,\text{then}\, e_2 \,\text{else}\, e_3 : t/C_{1,2,3} \wedge \tau_1 \sqsubseteq \mathit{bool} \wedge \tau_2 \sqsubseteq t \wedge \tau_3 \sqsubseteq t}$$

Algorithmic rules (cont.)

App

$$\frac{\Gamma, H_1 \vdash_{\mathcal{V}_1} e_1 : \tau_1/C_1 \qquad \Gamma, H_2 \vdash_{\mathcal{V}_2} e_2 : \tau_2/C_2 \qquad \mathcal{V}_1 \sharp \mathcal{V}_2}{\Gamma, H_1; H_2; h \vdash_{\mathcal{V}_1 \cup \mathcal{V}_2 \cup \{t,h\}} e_1 \ e_2 : t/C_{1,2} \wedge \tau_1 \sqsubseteq \tau_2 \xrightarrow{h} t}$$

Fix

$$\frac{\Gamma; x : t; z : t \xrightarrow{h} t', H \vdash_{\mathcal{V}} e : \tau/C}{\Gamma, \epsilon \vdash_{\mathcal{V} \cup \{t,t',h\}} \lambda_z x.e : t \xrightarrow{h} t'/C \wedge \tau \sqsubseteq t' \wedge H \sqsubseteq h}$$

Let

$$\frac{\begin{array}{c}\Gamma, \epsilon \vdash_{\mathcal{V}_1} v : \tau'/C' \\ \Gamma; x : \forall \bar{\alpha}.\tau'/C', H \vdash_{\mathcal{V}_2} e : \tau/C \qquad \bar{\alpha} = \mathsf{fv}(\tau', C') - \mathsf{fv}(\Gamma) \qquad \mathcal{V}_1 \sharp \mathcal{V}_2\end{array}}{\Gamma, H \vdash_{\mathcal{V}_1 \cup \mathcal{V}_2} \mathsf{let}\ x = v \,\mathsf{in}\, e : \tau/C \wedge C'}$$

Definition 8 (Canonical judgment) *A canonical judgment is a judgment having distinct bound variables in the type environment.*

Relating logical and algorithmic judgements

Definition 9 (Substitution) *A* substitution $\psi : \mathcal{V} \rightarrow \mathcal{T}$ *is a well-kinded, finite mapping from type variables to types.*

Definition 10 (Solution) *A substitution* ψ *is a* solution *to a constraint* C*, written* $\psi \vdash C$*, iff it is derivable according to the following rules:*

$$\frac{}{\psi \vdash \mathbf{true}} \qquad \frac{\psi(\tau_1) \preccurlyeq \psi(\tau_2)}{\psi \vdash \tau_1 \sqsubseteq \tau_2} \qquad \frac{\psi \vdash C_1 \qquad \psi \vdash C_2}{\psi \vdash C_1 \wedge C_2}$$

Relating logical and algorithmic judgements

Definition 11 (Most general solution) *If ψ and ψ' are solutions of C, the ψ is* more general *than ψ' iff there exists a substitutions ψ'' such that $\psi' = \psi'' \circ \psi$. A substitution is a* most general solution *(MGS) of C iff ψ is a solution of C and is more general than any other solution of C.*

Definition 12 (Satisfiable) *A canonical derivable judgment $\mathcal{J}$ is* satisfiable *iff there exists ψ, such that ψ solves the conjunction of all the contraints in the judgement, written $\psi \vdash \mathcal{J}$.*

Relating logical and algorithmic judgements

Definition 13 (Solved form*) *Given a derivable judgment $\mathcal{J}$, satisfied under ψ, the logical judgment $\psi(\mathcal{J})$ is a* solved form *of $\mathcal{J}$.*

Well, not quite... More precisely:

$$
\begin{aligned}
\mathcal{J} &\triangleq x_1 : \forall\bar{\alpha}_1.\tau_1/C_1; \ldots ; x_n : \forall\bar{\alpha}_n.\tau_n/C_n, H \vdash_{\mathcal{W}} e : \tau_0/C_0 \\
\mathcal{J}' &\triangleq x_1 : \forall\bar{\alpha}'_1.\psi(\tau_1); \ldots ; x_n : \forall\bar{\alpha}'_n.\psi(\tau_n), \psi(H) \vdash e : \psi(\tau_0)
\end{aligned}
$$

Where $\bar{\alpha}'_i$ are the *truly* quantifiable variables in $\psi(\tau_i)$. Everything you always wanted to know about solved forms but were afraid to ask is in the thesis.

Constraint solution algorithm

Although trace equivalence is undecidable in general, the inference algorithm maintains a form on constraints such that constraint satisfaction is decidable.

Namely, (equality) constraints between (non-trace effect) types contain only trace effect variables. Eg:

$$\tau_1 \xrightarrow{h_1} \tau_1' \sqsubseteq \tau_2 \xrightarrow{h_2} \tau_2'$$

Constraints between trace effects are always variable in the upper bound.

$$H \sqsubseteq h$$

Constraint solution algorithm

So, unification can solve type constraints.

Trace effect constraints can be solved by exploiting system of lower bounds. For example, if

$$C \triangleq H_1 \sqsubseteq h \wedge H_2 \sqsubseteq h \wedge \ldots \wedge H_n \sqsubseteq h$$

Then:

$$[(\mu h.H_1|H_2|\ldots|H_n)/h] \vdash C$$

Because:

$$H_i \preccurlyeq \mu h.H_1|H_2|\ldots|H_n$$

NB: μ needed since h may appear in H_i.

Constraint solution algorithm

$$\begin{array}{rcl} MGS(C) & = & \text{let } \psi_1 = U(C \setminus C') \text{ in } MGS_{\mathbf{H}}(\psi_1(C')) \circ \psi_1 \\ & & \text{where } C' = \left\{H \sqsubseteq H' \mid H \sqsubseteq H' \in C\right\} \end{array}$$

$$bounds(h, C) \;=\; H_1|\cdots|H_n \text{ where } \{H_1, \ldots, H_n\} = \{H \mid H \sqsubseteq h \in C\}$$

$$\begin{array}{rcl} MGS_{\mathbf{H}}(\varnothing) & = & \varnothing \\ MGS_{\mathbf{H}}(C) & = & \text{let } \psi = [h'|\mu h.bounds(h, C)/h] \text{ in} \\ & & MGS_{\mathbf{H}}(\psi(C \setminus \{H \sqsubseteq h \mid H \sqsubseteq h \in C\})) \circ \psi \\ & & \text{where } h' \text{ fresh} \end{array}$$

Where U is the standard unification algorithm.

Constraint solution algorithm

Lemma 1 (Correctness of MGS) *For any friendly C, $MGS(C)$ is a most general solution of C.*

Where *friendly* refers to the invariant on constraints maintained by inference.

Proof of the friendliness invariant is a straightforward induction on derivations.

Soundness proof

- Main lemma
- Soundness of inference
- Algorithmic type safety

Main lemma

Lemma 2 *If* $\Gamma, H \vdash_{\mathcal{W}} e : \tau/C$ *is derivable, then so is any most general solved form of* $\Gamma, H \vdash_{\mathcal{W}} e : \tau/C \wedge C_G$*, where* C_G *is arbitrary.*

Proof. By induction on the derivation of $\mathcal{J} \triangleq \Gamma, H \vdash_{\mathcal{W}} e : \tau/C$, reasoning by case analysis on the last rule used in the derivation. In each case, a logical judgment is constructed such that it is a most general solved form of $\Gamma, H \vdash_{\mathcal{W}} e : \tau/C \wedge C_G$ under ψ and then is shown to be logically derivable. □

Exemplary case: Fix

By inversion of the inference relation, $e = \lambda_z x.e'$, $\tau = t \xrightarrow{h} t'$, $H = \epsilon$, and there exists a judgment:

$$\mathcal{J}_1 \;\triangleq\; \Gamma; x : t; z : t \xrightarrow{h} t', H' \vdash_{\mathcal{W}} e' : \tau'/C'$$

Where:

$$C \;=\; C' \wedge \tau' \sqsubseteq t' \wedge H' \sqsubseteq h$$

$C_G \wedge C' \wedge \tau' \sqsubseteq t' \wedge H' \sqsubseteq h$ has a solution, so the inductive hypothesis applies to the judgment $\Gamma; x : t; z : t \xrightarrow{h} t', H' \vdash e' : \tau'/C_G \wedge C' \wedge \tau' \sqsubseteq t' \wedge H' \sqsubseteq h$, which therefore has a derivable most general solved form under ψ, namely $\Gamma'; x : \psi(t); z : \psi(t \xrightarrow{h} t'), \psi(H') \vdash e' : \psi(\tau')$.

Note that $\psi(t \xrightarrow{h} t') = \psi(t) \xrightarrow{\psi(h)} \psi(t')$.

Exemplary case: Fix

Therefore, the following derivation can be constructed using the logical rules Weaken and Fix:

$$\dfrac{\dfrac{\Gamma'; x : \psi(t); z : \psi(t) \xrightarrow{\psi(h)} \psi(t'), \psi(H') \vdash e' : \psi(t') \qquad \psi(H') \preccurlyeq \psi(h)}{\Gamma'; x : \psi(t); z : \psi(t) \xrightarrow{\psi(h)} \psi(t'), \psi(h) \vdash e' : \psi(t')}}{\Gamma', \epsilon \vdash \lambda_z x.e' : \psi(t) \xrightarrow{\psi(h)} \psi(t')}$$

Which shows a most general solved form of $\Gamma, H \vdash e : \tau$ is derivable. So the case holds.

NB: Cases Var and Let are not so easy...

Corollaries

Theorem 3 (Soundness of inference) *If* $\varnothing, H \vdash_{\mathcal{W}} e : \tau/C$ *is satisfiable, then* $\varnothing, \psi(H) \vdash e : \psi(\tau)$ *is derivable where* $\psi = MGS(C)$*.*

Proof. Immediate from main lemma and Correctness of MGS. □

Theorem 4 (Algorithmic Type Safety) *If* $\Gamma, H \vdash_{\mathcal{W}} e : \tau/C$ *is valid for closed* e*, then* e *does not go wrong.*

Proof. Immediate from Soundness of inference and Logical type safety. □

Implementation

A prototype implementation is available online.

It implements all the algorithms I've discussed today.

Written in OCaml.

Proved valuable when doing the *theoretical* development (providing counterexamples and a testable framework).

Inlcudes many features not covered today: subtyping, trace effect transformations, direct inference rules.

Digressions

- Trace effect transformations
- Direct inference rules
- Most generality

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.

Conclusion

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.

Trace effect analysis can be automated soundly.

The End

Thank you.