Types for Information Flow Analysis

The confinement problem (Lampson, 1973)

- Information systems (computers) run *multiple* processes,
- on behalf of *multiple* users,
- which read and write *multiple* bodies of data.

It is often desirable to control the *flow* of information through these systems, so as to preserve data *secrecy* or *integrity*.

Access control

- *Access control*, a widespread, authentication-based mechanism,
- only restricts the *initial* release of data.
- Thus, it requires *trust*, which is often misplaced,
- especially concerning *programs*.

Information flow control

- In the *absence* of trust,
- one must *check* that all flows of information are acceptable,
- which requires a notion of *flow*,
- a security *policy*,
- and an *automated* information flow analysis.

“System-wide” information flow control

- Computer systems are *non-deterministic*, *concurrent*,
- *interact* with peripheral devices, networks, users,
- whose behavior *cannot* be analyzed beforehand,
- are *observable* through physical means: *time*, *power* consumption.

Language-based information flow control

- A program written in a *deterministic*, *sequential* language,
- does not interact, except by receiving *input* and producing *output*,
- can be analyzed *before* being run,
- has a well-defined, abstract *semantics*.

Why begin with language-based security?

- It is *much* easier.
- It *must* be a component of "system-wide" security, lest processes be viewed as "black boxes".
- By enriching the programming language at hand, it may be possible to ultimately *reconcile* the two approaches.

Defining information flow

When does a statement S cause information to *flow* from x to y? Several definitions can be proposed:

- when S causes the conditional *entropy* of x, given y, to decrease.
- when varying x's initial value causes y's final value to vary, i.e. when y's final value *depends* on x.
- when x's initial value can be *reconstructed* from y's final value.

The absence of dependency (i.e. the negation of criterion #2) is called *non-interference* (Goguen and Meseguer, 1982).

Defining an information flow policy

Bell & LaPadula (1973) and Denning (1975) suggest adopting a security lattice $(\mathcal{L}, \leq)$, and assigning a label $\underline{x}$ to every variable x (or, more generally, to every *piece* of input and output). Then, a flow $x \to y$ is acceptable iff $\underline{x} \leq \underline{y}$.

Sample lattices:

- $\{\mathrm{L}, \mathrm{H}\}$ allows distinguishing public vs. secret (or trusted vs. untrusted) data.
- The *powerset* of a set of principals allows telling who may consult (or who produced) a piece of data.
- Taking the *product* of several such lattices allows forming composite policies, such as the military classification lattice.

Specifying a program

Given the lattice $\{\text{L} \leq \text{H}\}$, the assertion

$$P : \mathsf{int}^{\text{L}} \times \mathsf{int}^{\text{H}} \rightarrow \mathsf{int}^{\text{L}} \times \mathsf{int}^{\text{H}}$$

claims that P's first output does not depend on its second input:

$$\forall k, k_1, k_2 \quad \text{fst}\ (P\ (k, k_1)) = \text{fst}\ (P\ (k, k_2))$$

Such specifications

- *rely upon* and *enrich* the original type structure;
- *encode* non-interference assertions.

Specifying a program, more abstractly

Denning (1975) points out that the same non-interference assertion can be encoded in a more polymorphic, lattice-independent manner:

$$P : \forall \ell, h[\ell \leq h].\mathsf{int}^{\ell} \times \mathsf{int}^{h} \rightarrow \mathsf{int}^{\ell} \times \mathsf{int}^{h}$$

This emphasizes the fact that information flow analysis is a *pure* dependency analysis.

A brief (incomplete) history of language-based information flow control

- Denning (1975-1982): imperative language with polymorphic, recursive first-order procedures. No correctness proof.
- Banâtre, Bryce and Le Métayer (1994); Volpano and Smith (1997): imperative language without procedures.
- Palsberg and Ørbæk (1995): pure λ-calculus. No correctness proof.
- Heintze and Riecke (1998), Abadi, Banerjee, Heintze and Riecke (1999), Pottier and Conchon (2000): purely functional language with data structures.
- Myers (1999): analysis of Java, with dynamic aspects. No correctness proof.
- Pottier and Simonet (2002): functional language with references and exceptions (ML).

Outline

1. Abadi *et al.*'s PER-based approach;
2. Pottier and Conchon's translation-based approach;
3. An overview of Denning's analysis;
4. Pottier and Simonet's direct, syntactic approach.

Non-interference is not a *safety* property: it requires *relating* two processes in execution. How does one attack it? What is the meaning of security annotations, i.e. how does the interpretation of int^{L} differ from that of int^{H}?

The *dependency core calculus* (DCC)

A call-by-name λ-calculus with products and sums, extended with two constructs that allow *marking* a value and *using* such a value.

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.e \mid e\,e \mid \ldots \mid \mathsf{mark}\, e \mid \mathsf{use}\, x = e\, \mathsf{in}\, e \\ t & ::= & t \to t \mid \mathsf{unit} \mid t + t \mid t \times t \mid H(t) \end{array}$$

(For simplicity, take $\mathcal{L} = \{\textsc{l} \leq \textsc{h}\}$.) In the operational semantics, these constructs are no-ops.

Proposed by Abadi, Banerjee, Heintze and Riecke (1999), drawing on existing ideas from binding-time analysis.

Typing DCC

The typing rules keep track of marks.

$$\textsc{Mark}\quad \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathsf{mark}\, e : H(t)} \qquad \textsc{Use}\quad \frac{\Gamma \vdash e_1 : H(t_1) \qquad \Gamma; x : t_1 \vdash e_2 : t_2 \qquad \lhd\, t_2}{\Gamma \vdash \mathsf{use}\, x = e_1\, \mathsf{in}\, e_2}$$

Every use of a value of marked type must produce a value of *protected* type, a slight generalization:

$$\lhd\, H(t) \qquad \frac{\lhd\, t_2}{\lhd\, t_1 \rightarrow t_2} \qquad \frac{\lhd\, t_1 \qquad \lhd\, t_2}{\lhd\, t_1 \times t_2}$$

If t is protected, then it is *isomorphic* to $H(t)$, as we will see.

PER Basics

A *partial equivalence relation* on A is a symmetric, transitive relation on A. It can be viewed as an equivalence relation on a subset of A, formed of those elements $x \in A$ such that $x\ R\ x$ holds.

We write $x : R$ for $x\ R\ x$. We write $R \multimap R'$ for the relation defined by

$$f\ (R \multimap R')\ g \iff (\forall x, y \quad x\ R\ y \Rightarrow f(x)\ R'\ g(y)).$$

A model of DCC

Consider the category where

- an *object* t is a cpo $|\,t\,|$ equipped with a PER, also written t.
- a *morphism* from t to u is a continuous function f such that $f : t \rightarrowtriangle u$.

The relation t specifies a low-level observer's *view* of t: it groups values of type t into classes whose elements must not be distinguished by such an observer.

The condition on morphisms is the *non-interference* statement associated with the type $t \rightarrow u$.

For instance, consider the flat cpo $\mathsf{bool} = \{\mathsf{true}, \mathsf{false}\}$. Define the objects boolL and boolH by equipping bool with the diagonal relation (resp. the everywhere true relation). Then, the assertion

$$f : \mathsf{boolH} \multimap \mathsf{boolL}$$

is syntactic sugar for

$$\forall x, y \in \mathsf{bool} \quad x \;\mathsf{boolH}\; y \Rightarrow f(x) \;\mathsf{boolL}\; f(y)$$

that is,

$$\forall x, y \in \mathsf{bool} \quad f(x) = f(y)$$

i.e. requires f to be a *constant* function.

Interpreting types

The function type $t \to u$ is interpreted as the space of continuous functions from $|\, t \,|$ to $|\, u \,|$, equipped with the relation $t \rightarrowtriangle u$. (That is, two functions are indistinguishable to a low-level observer if they map indistinguishable inputs to indistinguishable outputs.)

The marked type $H(t)$ is interpreted as the cpo $|\, t \,|$, equipped with the *everywhere true* relation. (That is, a low-level observer must not be able to distinguish values of a marked type.)

Lemma. If $\lhd\, t$, then t and $H(t)$ are isomorphic.

In other words, a low-level observer's view of a protected type is the everywhere true relation.

Interpreting expressions

Interpreting MARK boils down to

Lemma. If $e : t$, then $e : H(t)$.

Interpreting USE requires checking

Lemma. If $e : t_1 \to t_2$ and $\lhd\, t_2$, then $e : H(t_1) \to t_2$.

Proof. Because t_2 is protected, we have $\forall x, y \quad (e\,x)\; t_2\; (e\,y)$. So, *a fortiori*, $\forall x, y \quad x\; H(t_1)\; y \Rightarrow (e\,x)\; t_2\; (e\,y)$ holds. This is $e\; (H(t_1) \to t_2)\; e$, that is, $e : H(t_1) \to t_2$.

The fact that this category is a *model* of DCC shows that every program satisfies the *non-interference* assertion encoded within its type. The PER approach gives *direct* meaning to annotated types.

Full DCC

Full DCC has one "mark" type constructor, written T_ℓ, per security level $\ell \in \mathcal{L}$. $\ell \lhd T_{\ell'}(t)$ holds iff $\ell \leq \ell'$.

$$\textsc{Use}\quad \frac{\Gamma \vdash e_1 : T_\ell(t_1) \qquad \Gamma; x : t_1 \vdash e_2 : t_2 \qquad \ell \lhd t_2}{\Gamma \vdash \mathsf{use}\, x = e_1 \,\mathsf{in}\, e_2}$$

A "homogeneous" type system, such as that of Heintze and Riecke (1998), is easily translated down to full DCC:

$$(t_1 \to t_2)^\ell \equiv T_\ell(t_1 \to t_2) \qquad (t_1 \times t_2)^\ell \equiv T_\ell(t_1 \times t_2) \qquad (t_1 + t_2)^\ell \equiv T_\ell(t_1 + t_2)$$

Subtyping is translated into *coercions* programmed using mark and use. Thus, DCC can also be seen as a *vehicle* for proving other systems correct.

The *labelled calculus* approach

Compose a *dynamic* dependency analysis with a *static* type checker.

- The former can be expressed as an *instrumented* semantics.
- Then, interfacing it with the latter requires a *translation*.
- The latter is viewed as a *black box*, yielding a modular proof.
- This suppresses the need to *guess* what the typing rules should be.

Proposed by Pottier & Conchon (2000).

Defining the labelled calculus

Following Abadi, Lampson & Lévy (1996).

$$e ::= x \mid \lambda x.e \mid (e\ e) \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \ldots \mid l : e \qquad (l \in \mathcal{L})$$

Operational semantics:

$$(l : e_1)\ e_2 \quad \to \quad l : (e_1\ e_2) \quad (\textit{lift})$$

For instance,

$$(\textsc{l} : (\lambda xy.y))\ (\textsc{h} : 27) \to \textsc{l} : ((\lambda xy.y)\ (\textsc{h} : 27)) \to \textsc{l} : (\lambda y.y)$$

The meaning of labels: *stability*

Prefixes are defined by augmenting expressions with a hole $\_$. Write $e \preceq e'$ if e is obtained from e' by replacing some sub-terms with holes.

Monotonicity. Let e, e' be prefixes such that $e \preceq e'$. If f is an expression such that $e \rightarrow^\star f$, then $e' \rightarrow^\star f$.

Let $\lfloor e \rfloor$ be the prefix of e where every sub-term labelled $\textsc{h}$ has been pruned. (Still assuming $\mathcal{L} = \{\textsc{l} \le \textsc{h}\}$.)

Stability. Assume e is a prefix and f is an expression. If $e \rightarrow^\star f$ and $\lfloor f \rfloor = f$, then $\lfloor e \rfloor \rightarrow^\star f$.

Defining the translation

A *translation* must map the labelled λ-calculus into a more standard λ-calculus.

- A labelled value is mapped to a *pair* of the value and its label.
- For homogeneity, every value should carry exactly *one* label, which requires *joining* multiple labels, exploiting the fact that $\mathcal{L}$ is a lattice.

The target calculus must have label constants, and a *join* operation:

$$l \,@\, m \quad \rightarrow \quad l \sqcup m \quad (join)$$

Translation

$$
\begin{array}{rcl}
[\![k]\!] & = & (k,\ \bot) \\
[\![x]\!] & = & x \\
[\![\lambda x.e]\!] & = & (\lambda x.[\![e]\!],\ \bot) \\
[\![e_1\ e_2]\!] & = & \textsf{open } [\![e_1]\!] \textsf{ as } (x,\ t) \textsf{ in} \\
 & & \textsf{open } x\ [\![e_2]\!] \textsf{ as } (y,\ u) \textsf{ in} \\
 & & (y,\ t @ u) \\
[\![\textsf{let } x = e_1 \textsf{ in } e_2]\!] & = & \textsf{let } x = [\![e_1]\!] \textsf{ in } [\![e_2]\!] \\
[\![l : e]\!] & = & \textsf{open } [\![e]\!] \textsf{ as } (x,\ t) \textsf{ in} \\
 & & (x,\ l @ t)
\end{array}
$$

Correctness of the translation

Simulation. If $e \to f$, then $[\![e]\!]$ reduces to $[\![f]\!]$, modulo an *administrative* congruence, whose axioms include:

$$\begin{array}{rcl} (\mathrm{fst}\, e,\ \mathrm{snd}\, e) & \equiv & e \\ (e_1 \mathbin{@} e_2) \mathbin{@} e_3 & \equiv & e_1 \mathbin{@} (e_2 \mathbin{@} e_3) \\ \perp \mathbin{@} e & \equiv & e \end{array}$$

Axiomatizing a type system for the target calculus

For the sake of modularity, we view a type system as an (opaque) set of *types*, together with a relation between (closed) expressions and types, written $e : t$. Among our requirements are

- Reduction and administrative congruence preserve types.
- Every well-typed, irreducible expression is a value.
- Labels are types; $l : l'$ implies $l \leq l'$.
- There is a type int. A value satisfies $v : \mathsf{int}$ iff it is an integer constant.
- There is a type function $\times$ such that $(e,\ f) : t \times u$ iff $e : t$ and $f : u$.

Putting it all together

Given a source expression e, let $e : t$ hold if and only if $[\![e]\!] : t$ holds.

Subject reduction is an immediate consequence of the simulation lemma and of our requirements. *Progress* is straightforward.

Non-interference. If $e : \mathsf{int} \times \textsc{l}$ and $e \rightarrow^\star v$, then $\lfloor e \rfloor \rightarrow^\star v$.

Proof. Subject reduction yields $v : \mathsf{int} \times \textsc{l}$, that is, $[\![v]\!] : \mathsf{int} \times \textsc{l}$. Thus, v must be of the form $l_1 : l_2 : \ldots : l_n : k$. $[\![v]\!]$ must then reduce to $(k,\ l_1 \sqcup l_2 \sqcup \ldots \sqcup l_n)$, which implies $l_1 \sqcup l_2 \sqcup \ldots \sqcup l_n$ has type $\textsc{l}$. So, every l_i is $\textsc{l}$. So, $\lfloor v \rfloor$ equals v, which, by stability, implies $\lfloor e \rfloor \rightarrow^\star v$.

Example: a simply-typed λ-calculus with subtyping

Types are given by

$$\tau ::= \mathsf{int} \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \mid l$$

Subtyping extends the ordering on $\mathcal{L}$.

$$\textsc{Label} \quad \Gamma \vdash l : l \qquad\qquad \textsc{Join} \quad \frac{\Gamma \vdash e_1 : l_1 \qquad \Gamma \vdash e_2 : l_2}{\Gamma \vdash e_1 \mathbin{@} e_2 : l_1 \sqcup l_2}$$

One checks that this type system meets all of our requirements.

Deriving Direct Rules

Compose the translation rules with the typing rules.

$$\textsc{Int}\quad \Gamma \vdash k : \mathsf{int} \times \bot$$

$$\textsc{Var}\quad \frac{\Gamma(x) = \varsigma}{\Gamma \vdash x : \varsigma}$$

$$\textsc{Abs}\quad \frac{\Gamma; x : \varsigma \vdash e : \varsigma'}{\Gamma \vdash \lambda x.e : (\varsigma \to \varsigma') \times \bot}$$

$$\textsc{App}\quad \frac{\Gamma \vdash e_1 : (\varsigma_2 \to \tau \times l) \times l' \qquad \Gamma \vdash e_2 : \varsigma_2}{\Gamma \vdash e_1\, e_2 : \tau \times (l \sqcup l')}$$

$$\textsc{Label}\quad \frac{\Gamma \vdash e : \tau \times l'}{\Gamma \vdash (l : e) : \tau \times (l \sqcup l')}$$

One may write τ^l for $\tau \times l$ and require all types to be generated by

$$\tau ::= \mathsf{int} \mid \varsigma \to \varsigma \mid \varsigma \times \varsigma \mid \varsigma + \varsigma \qquad \varsigma ::= \tau^l$$

In short

- The proof is *modular*: it also yields polymorphic, constraint-based type systems.
- The approach is applicable to *other* calculi, e.g. the π-calculus under may-testing equivalence.
- However, the fact that the translation must be very simple *imposes* some design choices (e.g. a homogeneous type system).
- This approach gives an *operational* intuition for the distinction between $\mathsf{int}^{\mathrm{L}}$ and $\mathsf{int}^{\mathrm{H}}$: these types represent integers that carry different labels, in a labelled semantics.

Denning's static analysis (1975–82)

D. E. Denning addresses a programming language equipped with:

- first-order, recursive, *polymorphic* procedures,
- read-only or read/write parameters and local variables of base or array type,
- assignment, sequence, if, while, goto,
- no global variables, dynamic allocation, or exceptions.

No correctness proof is given.

Specifying procedures

The set of parameters on which every writable parameter depends must be given.

```
procedure max (x; y; var m { x, y });
begin
  if x > y then m := x else m := y
end;
```

In type-theoretic notation, we would write:

$$\texttt{max} : \forall x, y, m.(x \leq m, y \leq m) \Rightarrow \mathsf{int}^x \times \mathsf{int}^y \times (\mathsf{int}^m\ \texttt{ref}) \rightarrow \texttt{unit}$$

The level of local variables must also be declared:

```
procedure swap (var x { y }; var y { x });
var t { x, y };
begin
  t := x; x := y; y := t
end;
```

In type-theoretic notation, we would write:

$$\texttt{swap} : \forall x, y.(y \leq x, x \leq y) \Rightarrow (\mathsf{int}^{x}\ \texttt{ref}) \times (\mathsf{int}^{y}\ \texttt{ref}) \rightarrow \texttt{unit}$$

or, equivalently,

$$\texttt{swap} : \forall x.(\mathsf{int}^{x}\ \texttt{ref}) \times (\mathsf{int}^{x}\ \texttt{ref}) \rightarrow \texttt{unit}$$

Checking procedures

From these declarations, one determines the information level of every expression e, written $\underline{e}$, a subset of the procedure's parameters. Then, one checks that every statement is *secure* (i.e. does not cause leaks).

An assignment $x := e$ is secure if $\underline{e} \leq \underline{x}$ (*direct* flow).

A sequence $S_1; S_2$ is secure if S_1 and S_2 are both secure.

A conditional $\textsf{if}\, e \,\textsf{then}\, S$ is secure if

- S is secure;
- for every variable x that *may* be assigned within S, $\underline{e} \leq \underline{x}$ (*indirect* flow).

A procedure call is secure if flows between formals induce valid flows between actuals.

Example

The following procedure copies x into y, assuming x is initially 0 or 1.

```
procedure copy (x; var y)
var z;
begin
  y := 0; z := 0;
  if x = 0 then z := 1;
  if z = 0 then y := 1
end;
```

It is insecure. Its specification should be amended by declaring `var y { x }`, `var z { x }`.

The information flow from `x` to `y`, through `z`, is caused by the combined effect of *both* if statements, even though *every* execution skips one of them.

Towards a generalization: $\underline{pc}$

In Denning's restricted language, the set of variables that may be assigned within a given statement is *known*.

This is no longer true in the presence of, say, first-class references and functions. Thus, Heintze and Riecke (1998) and Myers (1999) *parameterize* the judgement "S is secure" with an information level, written $\underline{pc}$.

An assignment $x := e$ is secure at $\underline{pc}$ if $\underline{pc} \sqcup \underline{e} \leq \underline{x}$.

A sequence $S_1; S_2$ is secure at $\underline{pc}$ if S_1 and S_2 are both secure at $\underline{pc}$.

A conditional $\mathsf{if}\, e \,\mathsf{then}\, S$ is secure at $\underline{pc}$ if S is secure at $\underline{pc} \sqcup \underline{e}$.

pc becomes an additional (implicit) formal parameter to every procedure.

Information flow inference for ML

We have described analyses for a purely functional language and for a simple imperative language. First-class references are addressed by Heintze and Riecke (1998), exceptions by Myers (1999), albeit only informally. There remains to combine these ideas into a type system that supports type inference, and to prove it correct.

Proposed by Pottier and Simonet (2002).

Syntax

The language, dubbed ML, has *second-class* exceptions. It restricts certain expression forms to be built out of *values*.

$$
\begin{array}{rcl}
v & ::= & x \mid () \mid k \mid \lambda x.e \mid m \mid (v,\ v) \mid \mathsf{inj}_j\ v \\
a & ::= & v \mid \mathsf{raise}\ \varepsilon\, v \\
e & ::= & a \mid v\, v \mid \mathsf{ref}\ v \mid v := v \mid !\, v \mid \mathsf{proj}_j\ v \mid v\ \mathsf{case}\ x \succ e\ e \mid \mathsf{let}\ x = v\ \mathsf{in}\ e \mid E[e] \\
E & ::= & \mathsf{bind}\ x = [\,]\ \mathsf{in}\ e \\
 & \mid & [\,]\ \mathsf{handle}\ \varepsilon\, x \succ e \mid [\,]\ \mathsf{handle}\ e\ \mathsf{done} \mid [\,]\ \mathsf{handle}\ e\ \mathsf{raise} \mid [\,]\ \mathsf{finally}\ e
\end{array}
$$

The semantics is call-by-value.

Trouble with labels

The statement $\textsf{if}\ x = 0\ \textsf{then}\ z := 1$ causes information to flow from x to z, even when it is skipped. As a result, designing a labelled semantics becomes problematic.

Instead of using labels and prefixes to reason about arbitrary data:

$$(\textsc{l} : (\lambda xy.y))\ (\textsc{h} : \_) \rightarrow^{\star} \textsc{l} : (\lambda y.y)$$

which implied $(\lambda xy.y)\ 27 \rightarrow^{\star} \lambda y.y$ and $(\lambda xy.y)\ 68 \rightarrow^{\star} \lambda y.y$ by *monotonicity* and *erasure*, we will reason directly about *two* processes that share some structure:

$$(\lambda xy.y)\ \langle 27 \mid 68 \rangle \rightarrow^{\star} \lambda y.y$$

with the same consequences, this time via *projection*.

The *bracket* calculus

The language ML^2 is defined as an extension of ML.

$$
\begin{array}{rcl}
v & ::= & \ldots \mid \langle v \mid v \rangle \mid \mathsf{void} \\
a & ::= & \ldots \mid \langle a \mid a \rangle \\
e & ::= & \ldots \mid \langle e \mid e \rangle
\end{array}
$$

A ML^2 term encodes a pair of ML terms. For instance, $\langle v_1 \mid v_2 \rangle\, v$ and $\langle v_1\, v \mid v_2\, v \rangle$ both encode the pair $(v_1\, v, v_2\, v)$.

Two *projection* functions map a ML^2 term to the two ML terms which it encodes. In particular, $\lfloor \langle e_1 \mid e_2 \rangle \rfloor_i = e_i$, for $i \in \{1, 2\}$.

Semantics of ML^2: principles

As in the labelled λ-calculus, each language construct is typically dealt with by *two* reduction rules: a standard one, and one that moves (*lifts*) brackets out of the way. The latter are no-ops w.r.t. the term's projections.

$$\begin{array}{rcll} (\lambda x.e)\,v & \rightarrow & e[x \Leftarrow v] & (\beta) \\ \langle v_1 \mid v_2 \rangle\, v & \rightarrow & \langle v_1\, \lfloor v \rfloor_1 \mid v_2\, \lfloor v \rfloor_2 \rangle & \text{(lift-app)} \end{array}$$

$$\begin{array}{rcll} \mathsf{proj}_j\ (v_1,\ v_2) & \rightarrow & v_j & \text{(proj)} \\ \mathsf{proj}_j\ \langle v_1 \mid v_2 \rangle & \rightarrow & \langle \mathsf{proj}_j\ v_1 \mid \mathsf{proj}_j\ v_2 \rangle & \text{(lift-proj)} \end{array}$$

(Slightly simplified versions shown.)

Brackets encode the *differences* between two programs, i.e. their "secret" parts. The (hypothetical) reduction rule

$$e \rightarrow \langle \lfloor e \rfloor_1 \mid \lfloor e \rfloor_2 \rangle,$$

while computationally correct, would cause the type system to view *every* expression as "secret".

The "lift" rules provide an explicit description of information flow, and must be made as precise as possible.

Semantics of ML^2: imperative constructs

The meaning of memory locations is given by a *store* μ, i.e. a partial map from memory locations to values – which may contain brackets. Store bindings of the form $m \mapsto \langle v \mid \mathsf{void} \rangle$ or $m \mapsto \langle \mathsf{void} \mid v \rangle$ account for situations where the two programs at hand have different allocation patterns.

Reductions which take place inside a $\langle \cdot \mid \cdot \rangle$ construct must use or affect only one projection of the store. For this purpose, let *configurations* be of the form $e /_i \mu$, where $i \in \{\bullet, 1, 2\}$. Write e / μ for $e /_\bullet \mu$.

$$\frac{e_i /_i \mu \to e_i' /_i \mu' \qquad e_j = e_j' \qquad \{i, j\} = \{1, 2\}}{\langle e_1 \mid e_2 \rangle / \mu \to \langle e_1' \mid e_2' \rangle / \mu'} \text{ (bracket)}$$

Define the following auxiliary function:

$$\begin{array}{rcl} \mathrm{update}_\bullet\, v\, v' & = & v' \\ \mathrm{update}_1\, v\, v' & = & \langle v' \mid \lfloor v \rfloor_2 \rangle \\ \mathrm{update}_2\, v\, v' & = & \langle \lfloor v \rfloor_1 \mid v' \rangle \end{array}$$

Then, the reduction rules for assignment are:

$$\begin{array}{rcll} m := v \;/_i\; \mu & \rightarrow & () \;/_i\; \mu[m \mapsto \mathrm{update}_i\, \mu(m)\, v] & \text{(assign)} \\ \langle v_1 \mid v_2 \rangle := {\color{blue}v} \;/\; \mu & \rightarrow & \langle v_1 := {\color{blue}\lfloor v \rfloor_1} \mid v_2 := {\color{blue}\lfloor v \rfloor_2} \rangle \;/\; \mu & \text{(lift-assign)} \end{array}$$

Analogous rules are given for dynamic allocation and dereferencing.

Example

$$
\begin{array}{ll}
& \mathsf{if}\ !x = 0\ \mathsf{then}\ z := 1\ /\ x \mapsto \langle 0 \mid 1 \rangle, z \mapsto 0 \\
\rightarrow & \mathsf{if}\ \langle 0 \mid 1 \rangle = 0\ \mathsf{then}\ z := 1\ /\ x \mapsto \langle 0 \mid 1 \rangle, z \mapsto 0 \\
\rightarrow & \mathsf{if}\ \langle 0 = 0 \mid 1 = 0 \rangle\ \mathsf{then}\ z := 1\ /\ x \mapsto \langle 0 \mid 1 \rangle, z \mapsto 0 \\
\rightarrow & \mathsf{if}\ \langle \mathsf{true} \mid \mathsf{false} \rangle\ \mathsf{then}\ z := 1\ /\ x \mapsto \langle 0 \mid 1 \rangle, z \mapsto 0 \\
\rightarrow & \langle \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ z := 1 \mid \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ z := 1 \rangle\ /\ x \mapsto \langle 0 \mid 1 \rangle, z \mapsto 0 \\
\rightarrow & \langle () \mid () \rangle\ /\ x \mapsto \langle 0 \mid 1 \rangle, z \mapsto \langle 1 \mid 0 \rangle
\end{array}
$$

Semantics of ML^2: exceptions

$$
\begin{array}{rcll}
\textsf{bind}\ x = v\ \textsf{in}\ e & \rightarrow & e[x \Leftarrow v] & \text{(bind)} \\
\textsf{raise}\ \varepsilon\, v\ \textsf{handle}\ \varepsilon\, x \succ e & \rightarrow & e[x \Leftarrow v] & \text{(handle)} \\
\textsf{raise}\ \varepsilon\, v\ \textsf{handle}\ e\ \textsf{done} & \rightarrow & e & \text{(handle-done)} \\
\textsf{raise}\ \varepsilon\, v\ \textsf{handle}\ e\ \textsf{raise} & \rightarrow & e;\ \textsf{raise}\ \varepsilon\, v & \text{(handle-raise)} \\
a\ \textsf{finally}\ e & \rightarrow & e;\ a & \text{(finally)} \\
E[a] & \rightarrow & a & \text{(pop)} \\
 & & \textit{if } E \textit{ handles neither } \lfloor a \rfloor_1 \textit{ nor } \lfloor a \rfloor_2 & \\
E[\langle a_1 \mid a_2 \rangle] \,/\, \mu & \rightarrow & \langle \lfloor E \rfloor_1[a_1] \mid \lfloor E \rfloor_2[a_2] \rangle \,/\, \mu & \text{(lift-context)} \\
 & & \textit{if none of the above rules applies} &
\end{array}
$$

Examples

$$\text{bind } x = \langle 3 \mid 4 \rangle \text{ in } (x,\ 0) \to (\langle 3 \mid 4 \rangle,\ 0)$$

$$\langle 3 \mid 4 \rangle \text{ handle } e \text{ done} \to \langle 3 \mid 4 \rangle$$

$$\begin{array}{ll} & \text{bind } x = \langle 3 \mid \text{raise } \epsilon\,() \rangle \text{ in } (x,\ 0) \\ \to & \langle \text{bind } x = 3 \text{ in } (x,\ 0) \mid \text{bind } x = \text{raise } \epsilon\,() \text{ in } (x,\ 0) \rangle \\ \to^\star & \langle (3,\ 0) \mid \text{raise } \epsilon\,() \rangle \end{array}$$

Relating ML^2 to ML

Soundness. Let $i \in \{1,2\}$. If $e \,/\, \mu \rightarrow e' \,/\, \mu'$, then $\lfloor e \,/\, \mu \rfloor_i \rightarrow \lfloor e' \,/\, \mu' \rfloor_i$.

Completeness. Assume $\lfloor e \,/\, \mu \rfloor_i \rightarrow^\star a_i \,/\, \mu'_i$ for all $i \in \{1,2\}$. Then, there exists a configuration $a \,/\, \mu'$ such that $e \,/\, \mu \rightarrow^\star a \,/\, \mu'$.

In short, brackets cannot *corrupt* or *block* reduction.

The bracket calculus can now be seen as a tool to attack the non-interference proof, provided we can *control* the proliferation of brackets during reduction. We will do so using a standard technique: a type system for ML^2, together with a *subject reduction* theorem.

The types of ML^2

Types and *rows* are defined as follows:

$$t ::= \mathsf{unit} \mid \mathsf{int}^{\ell} \mid (t \xrightarrow{pc\ [r]} t)^{\ell} \mid t\ \mathsf{ref}^{\ell} \mid t \times t \mid (t + t)^{\ell}$$
$$r ::= \{\varepsilon \mapsto pc\}_{\varepsilon \in \mathcal{E}}$$

Subtyping extends the ordering on information levels.

$$\mathsf{int}^{\oplus} \qquad (\ominus \xrightarrow{\ominus\ [\oplus]} \oplus)^{\oplus} \qquad \odot\ \ \mathsf{ref}^{\oplus} \qquad \oplus \times \oplus \qquad (\oplus + \oplus)^{\oplus} \qquad \{\varepsilon \mapsto \oplus\}_{\varepsilon \in \mathcal{E}}$$

The auxiliary predicate $\ell \lhd t$ holds if ℓ *guards* (*taints*) t:

$$\ell \lhd \mathsf{unit} \qquad \frac{\ell \leq \ell'}{\ell \lhd \mathsf{int}^{\ell'}} \qquad \frac{\ell \leq \ell'}{\ell \lhd (* \xrightarrow{*\ [*]} *)^{\ell'}}$$

$$\frac{\ell \leq \ell'}{\ell \lhd * \ \mathsf{ref}^{\ell'}} \qquad \frac{\ell \lhd t_1 \quad \ell \lhd t_2}{\ell \lhd t_1 \times t_2} \qquad \frac{\ell \leq \ell'}{\ell \lhd (* + *)^{\ell'}}$$

ML^2's type system

We distinguish two forms of typing judgements: one deals with values only, the other with arbitrary expressions.

$$\Gamma \vdash v : t$$

$$pc, \Gamma \vdash e : t \;\; [r]$$

They are connected via the following typing rule:

$$\textsc{e-Value}\quad \frac{\Gamma \vdash v : t}{*, \Gamma \vdash v : t \;\; [*]}$$

Keeping track of brackets

The type system is parameterized over an (upward-closed) set of information levels H, the "secret" levels. The system guarantees that the type of every bracket is guarded by some level in H:

$$\textsc{v-Bracket}\quad \frac{\Gamma \vdash v_1 : t \qquad \Gamma \vdash v_2 : t \qquad pc' \in H \qquad pc' \lhd t}{\Gamma \vdash \langle v_1 \mid v_2 \rangle : t}$$

Abstraction and application

Abstraction delays effects (pc, r); application forces them ($pc \leq pc'$).

$$\dfrac{pc, \Gamma[x \mapsto t'] \vdash e : t \;\; [r]}{\Gamma \vdash \lambda x.e : (t' \xrightarrow{pc \;\; [r]} t)^{*}} \;\textsc{v-Abs}$$

$$\dfrac{\Gamma \vdash v_1 : (t' \xrightarrow{pc' \;\; [r]} t)^{\ell} \qquad \Gamma \vdash v_2 : t' \qquad pc \leq pc' \qquad \ell \leq pc' \qquad \ell \lhd t}{pc, \Gamma \vdash v_1\, v_2 : t \;\; [r]} \;\textsc{e-App}$$

Information about the function may leak through its side effects ($\ell \leq pc'$) or through its result ($\ell \lhd t$).

Imperative constructs

Information encoded within the program counter may leak when writing a variable, forming an indirect flow. We follow Denning's solution ($pc \lhd t$). In the presence of first-class references, information about the reference's identity may leak as well ($\ell \lhd t$).

$$\frac{\Gamma \vdash v_1 : t\ \mathsf{ref}^{\ell} \qquad \Gamma \vdash v_2 : t \qquad pc \sqcup \ell \lhd t}{pc, \Gamma \vdash v_1 := v_2 : \mathsf{unit}\ \ [*]}\ \textsc{e-Assign}$$

Raising an exception

The value carried by the exception must have fixed (declared, monomorphic) type $typexn(\varepsilon)$.

The amount of information carried by the exception itself is represented by pc at the point where the exception is raised.

$$\text{E-RAISE}\quad \frac{\Gamma \vdash v : typexn(\varepsilon)}{pc, \Gamma \vdash \mathsf{raise}\,\varepsilon\,v : * \;\; [\varepsilon : pc; *]}$$

Handling a specific exception

The amount of information carried by the exception, namely pc_ε, is read off the row associated with e_1.

The handler's side effects may reveal that an exception was caught ($pc \sqcup pc_\varepsilon$). So may the whole expression's result ($pc_\varepsilon \lhd t$).

E-HANDLE

$$\frac{\begin{array}{c} pc, \Gamma \vdash e_1 : t \;\; [\varepsilon : pc_\varepsilon ; r] \\ pc \sqcup pc_\varepsilon, \Gamma[x \mapsto \mathit{typexn}(\varepsilon)] \vdash e_2 : t \;\; [\varepsilon : pc' ; r] \qquad pc_\varepsilon \lhd t \end{array}}{pc, \Gamma \vdash e_1 \;\mathsf{handle}\; \varepsilon\, x \succ e_2 : t \;\; [\varepsilon : pc' ; r]}$$

Computing in sequence

e_2's side effects may reveal that that e_1 completed successfully, i.e. did not raise *any* exception. The level $\sqcup r_1$ is used as an approximation of the information disclosed in that case.

$$\textsc{e-Bind}\quad \frac{\begin{array}{c} pc, \Gamma \vdash e_1 : t' \;\; [r_1] \\ pc \sqcup (\sqcup r_1), \Gamma[x \mapsto t'] \vdash e_2 : t \;\; [r_2] \end{array}}{pc, \Gamma \vdash \mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2 : t \;\; [r_1 \sqcup r_2]}$$

Finally

An event that *must* occur conveys no information, so e_2's side effects are not constrained further than the whole expression's.

$$\textsc{e-Finally}\quad \frac{\begin{array}{c} pc, \Gamma \vdash e_1 : t \;\; [\, r \,] \\ pc, \Gamma \vdash e_2 : * \;\; [\, \partial\bot \,] \end{array}}{pc, \Gamma \vdash e_1 \; \mathsf{finally} \; e_2 : t \;\; [\, r \,]}$$

However, e_2 is required *not* to raise (informative) exceptions. Indeed, observing an exception originally raised by e_1 betrays the fact that e_2 has completed successfully. To avoid keeping track of this fact, we require it to convey no information.

Results

Subject reduction. If $\vdash e \,/\, \mu : t \ \ [r]$ and $e \,/\, \mu \rightarrow e' \,/\, \mu'$ then $\vdash e' \,/\, \mu' : t \ \ [r]$.

The type system assigns "high" security levels (i.e. levels in H) to values of the form $\langle v_1 \mid v_2 \rangle$. By subject reduction, any expression which may reduce to such a value must also carry a "high" annotation. Conversely, no expression with a "low" annotation can produce such a value.

Lemma. Let $\ell \notin H$. If $\vdash_H e : \mathsf{int}^\ell$ and $e \rightarrow^\star v$ then $\lfloor v \rfloor_1 = \lfloor v \rfloor_2$.

Thus, this approach also gives *operational* meaning to the distinction between $\mathsf{int}^{\mathrm{L}}$ and $\mathsf{int}^{\mathrm{H}}$.

Results

Non-interference. Choose $\ell, h \in \mathcal{L}$ such that $h \not\leq \ell$. Let $h \triangleleft t$. Assume $(x \mapsto t) \vdash e : \mathsf{int}^{\ell}$, where e is a ML expression. If, for all $i \in \{1, 2\}$, $\vdash v_i : t$ and $e[x \Leftarrow v_i] \rightarrow^{\star} v'_i$ hold, then $v'_1 = v'_2$.

Proof. Let $H = \uparrow\{h\}$. Define $v = \langle v_1 \mid v_2 \rangle$. By V-BRACKET, $\vdash_H v : t$ holds. A substitution lemma yields $\vdash_H e[x \Leftarrow v] : \mathsf{int}^{\ell}$. Now, $\lfloor e[x \Leftarrow v] \rfloor_i$ is $e[x \Leftarrow v_i]$, which, by hypothesis, reduces to v'_i. By *completeness*, there exists an answer a such that $e[x \Leftarrow v] \rightarrow^{\star} a$. Then, by *soundness*, we have $\lfloor a \rfloor_i = v'_i$ for all $i \in \{1, 2\}$, which implies that a is a value. Lastly, $h \not\leq \ell$ yields $\ell \notin H$. The result follows by the previous lemma.

Type inference

The type system presented so far is *ground*. We can build, on top of it, a type system with *type variables*, *finite syntax* for rows and type schemes, and *constraints*.

$$
\begin{array}{rcl}
\tau & ::= & \beta \mid \mathsf{unit} \mid \mathsf{int}^{\lambda} \mid (\tau \xrightarrow{\pi\ [\rho]} \tau)^{\lambda} \mid \tau\ \mathsf{ref}^{\lambda} \mid \tau \times \tau \mid (\tau + \tau)^{\lambda} \\
\rho & ::= & \gamma \mid (\varepsilon : \lambda; \rho) \mid \partial\lambda \\
\lambda, \pi & ::= & \delta \mid \ell \\
C & ::= & \mathbf{true} \mid \mathbf{false} \mid C \wedge C \mid \exists \alpha.C \\
 & \mid & \tau \leq \tau \mid \rho \leq \rho \mid \lambda \leq \lambda \\
 & \mid & \lambda \lhd \tau
\end{array}
$$

Type inference reduces to *constraint solving*.

Examples

The usual ML definition of lists:

$$\beta\,\mathsf{list} = \mathsf{unit} + (\beta \times \beta\,\mathsf{list})$$

must be decorated:

$$\beta\,\mathsf{list}^{\delta} = (\mathsf{unit} + (\beta \times \beta\,\mathsf{list}^{\delta}))^{\delta}$$

```
let rec length = function
  | []     -> 0
  | _ :: l -> 1 + length l
```

`val length:` $\forall \beta \delta . \beta \, \mathsf{list}^{\delta} \to \mathsf{int}^{\delta}$

```
let rec iter f = function
  | []     -> ()
  | x :: l -> f x; iter f l
```

`val iter:` $\forall [\delta_1 \sqcup \delta_2 \sqcup \delta_3 \sqcup (\sqcup \gamma) \leq \delta].(\beta \xrightarrow{\delta \;\; [\gamma]} \mathsf{unit})^{\delta_1} \to \beta \, \mathsf{list}^{\delta_2} \xrightarrow{\delta_3 \;\; [\gamma]} \mathsf{unit}$

$$\forall [\sqcup \gamma \leq \delta].(\beta \xrightarrow{\delta \;\; [\gamma]} \mathsf{unit})^{\delta} \to \beta \, \mathsf{list}^{\delta} \xrightarrow{\delta \;\; [\gamma]} \mathsf{unit}$$

Language-based information flow control?

- 25 years old,
- stirs much interest,
- *never* put into practice so far!
- *is* it feasible?

Selected References

- Martín Abadi, Anindya Banerjee, Nevin Heintze and Jon G. Riecke. *A Core Calculus of Dependency.* POPL'99.
- Martín Abadi, Butler Lampson and Jean-Jacques Lévy. *Analysis and Caching of Dependencies.* ICFP'96.
- Jean-Pierre Banâtre, Ciarán Bryce and Daniel Le Métayer. *Compile-time detection of information flow in sequential programs.* ESORICS'94.
- D. E. Bell and Leonard J. LaPadula. *Secure Computer Systems: Unified Exposition and Multics Interpretation.* Technical Report MTR-2997, 1975.
- Dorothy E. Denning. *Cryptography and Data Security.* Addison-Wesley, 1982.
- Joseph Goguen and José Meseguer. *Security policies and security models.* S&P'82.
- Nevin Heintze and Jon G. Riecke. *The SLam Calculus: Programming with Secrecy and Integrity.* POPL'98.

- Butler W. Lampson. *A Note on the Confinement Problem.* CACM 16:10, 1973.
- Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control.* Technical Report MIT/LCS/TR-783, 1999.
- Peter Ørbæk and Jens Palsberg. *Trust in the λ-calculus.* JFP 7:6, 1997.
- François Pottier and Sylvain Conchon. *Information Flow Inference for Free.* ICFP'00.
- François Pottier and Vincent Simonet. *Information Flow Inference for ML.* POPL'02.
- Andrei Sabelfeld and David Sands. *A PER Model of Secure Information Flow in Sequential Programs.* ESOP'99.
- Dennis Volpano and Geoffrey Smith. *A Type-Based Approach to Program Security.* TAPSOFT'97.