Types for Information Flow Analyses
It is often desirable to control the flow of information through these systems, so as to preserve data security or integrity.

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

The containment problem (Lamport, 1973)
especially concerning programs.

Thus, it requires trust, which is often misplaced.

only restricts the initial release of data.

Access control, a widespread, authentication-based mechanism,
and an automated information flow analysis.

a security policy,

which requires a notion of flow,

one must check that all flows of information are acceptable,

In the absence of trust,

Information flow control
Introduction

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

“System-wide” information flow control.
has a well-defined abstract semantics.

can be analyzed before being run.

does not interact, except by receiving input and producing output.

A program written in a deterministic, sequential language

Language-based information flow control
ultimately reconcile the two approaches. By enriching the programming language at hand, it may be possible to

as „black boxes“.

It must be a component of „system-wide“ security, lest processes be viewed

It is much easier.

Why begin with language-based security?
non-interference (Goguen and Meseguer, 1982).

The absence of dependence (i.e., the negation of criterion \#2) is called

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

Several definitions can be proposed:

- When does a statement \( y \) cause information to flow from \( x \) to \( y \)? Several
such as the military classification lattice.

Taking the product of several such lattices allows forming composite policies.

produced a piece of data.

The powerset of a set of principals allows telling who may consult (or who
produced) the trusted vs. untrusted data.

Sample lattices:

\[ \tau \leq x \implies \forall \eta \exists \xi \in \text{acceptable iff} \]

\[ \text{piece of input and output). Then, a flow } x \] for every variable \( x \) (or, more generally, to every

\( (\subseteq, \mathcal{J}) \) and assigning a label to every variable \( x \) suggest adopting a security lattice.

\[ \text{Bell & Lapadula (1973) and Denning (1972) suggest adopting a security lattice.} \]
Encoding non-interference assertions. •

rely upon the original type structure;

Such specifications

\[ \forall \ell, \ell_1, \ell_2 \quad \text{let } \quad \text{let } \quad d = \text{let } \]

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

\[ H \times H \to H \times H \]

Given the lattice $H \models I$, the assertion

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

encoded in a more polymorphic, lattice-independent manner:

\[
\forall y \geq \mathcal{H} \mathcal{Y} \forall \mathcal{Y} \mathcal{F} \rightarrow \mathcal{Y} \rightarrow \mathcal{F}.
\]

Defining (1975) points out that the same non-interference assertion can be

Specifying a program, more abstractly
A brief (incomplete) history of language-based information flow control

- Exceptions (ML).
  - Porter and Simonet (2002): Functional language with references and
  - Structures.
  - Porter and Conlan (2000): purely functional language with data
  - Heintze and Pierce (1998), Abadi, Barthelemy, Heintze, and Pierce (1999),
  - Imperative language without procedures.
  - Barthe, Borcea, and Le Métayer (1994); Volpano and Smith (1997);
  - Denning (1975-1982): imperative language with polymorphic, recursive

Introduction
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?

4. Potter and Simonet’s direct, syntactic approach.

3. An overview of Denning’s analysis?


1. Abdali et al.’s PPB-based approach.

Outline
The dependency core calculus (DCC)

Ideas from binding-time analysis.

Proposed by Abadi, Barendregt, Heintze, and Heeke (1999), drawing on existing constructs are no-ops.

For simplicity, take \( \{ h \supset \top \} = \mathcal{T} \). In the operational semantics, these

\[
\begin{align*}
(\top)_{H} & | \top \times \top | \top + \top | \text{unit} | \top \leftarrow \top \quad ::= \quad \top \\
\text{mark } c \ | \text{use } x = e \in e & \quad ::= \quad c \\
\cdots & \quad ::= \quad \cdots \\
x & \quad ::= \quad e
\end{align*}
\]

constructs that allow marking a value and using such a value.

A call-by-name \( \chi \)-calculus with products and sums, extended with two
If $t$ is protected, then it is isomorphic to $H(t)$, as we will see.

\[
\begin{array}{c}
\frac{t_1 \times t_2 \triangleright}{t_2 \triangleright \quad t_1 \triangleright} \\
\frac{t_2 \leftarrow t_1 \triangleright}{t_2 \triangleright}
\end{array}
\]

\[
(\ell)H \triangleright
\]

**Slight Generalization:**
Every use of a value of marked type must produce a value of protected type, a mixed use.

\[
\frac{\text{use } x = e_1 \in e_2}{t_2 \triangleright}
\]

\[
\frac{t_2 \triangleright}{t_2 : \ell_2 \triangleright}
\]

\[
\frac{t_1 : \ell_1 \triangleright}{(\ell_1)H : \ell_1 \triangleright}
\]

\[
(\ell)H : \ell \triangleright e : \ell
\]

The typing rules keep track of marks.
The dependency core calculus (DCC)

\[(\forall \bar{y}) \bar{y} R (x) \iff \forall \bar{y} \exists \bar{x}, \bar{A} \iff \exists (\forall \bar{y} \leftarrow \bar{y}) R \]

We write \( x : R \) for \( x R x \). We write \( \bar{y} \leftarrow R \) for the relation defined by

\[
\text{elements } x \in A \text{ such that } x R x \text{ holds.}
\]

An partial equivalence relation on a subset of \( A \) is a symmetric, transitive relation on \( A \).
The dependency core calculus (DCC)

A model of DCC

The condition on morphisms is the non-inference statement associated with

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.

Consider the category where

an object is a cpo $f$ equipped with a DCP, also written $f$.

A model of DCC

the type $t \leftarrow u$. 

\[ f \] is a continuous function such that $f$
A model of DCC

The dependency core calculus (DCC)

Let $f$ require to be a constant function:

\[(\mathcal{H})f = (x)f \quad \forall \text{bool} \in \mathcal{H}, x\]

That is,

\[(\mathcal{H})f \quad \text{bool} \quad (x)f \iff \mathcal{H} \text{ bool} \quad x \quad \forall \text{bool} \in \mathcal{H}, x\]

is syntactic sugar for

\[\text{bool} \iff \text{bool} ; f\]

true relation). Then, the assertion is everywhere.

By equipping bool with the diagonal relation (resp. the everywhere.

For instance, consider the hat copo bool = $\{\text{true}, \text{false}\}$. Define the objects bool.
true relation.

In other words, a low-level observer’s view of a protected type is the everywhere distinguishable values of a marked type.

Lemma. If \( t \geq t' \) then \( t \) and \( t' \) are isomorphic.

The marked type of a low-level observer must not be able to distinguishable inputs.

\( \text{That is, a low-level observer equips the co-interpretation with the function } \{ \neg n \} \) to the relation \( \{ \neg n \} \) so that maps indistinguishable inputs to indistinguishable outputs.

The function type \( \{ n \} \) is interpreted as the space of continuous functions from interpreting types.
A model of DCG

The dependency core calculus (DCG)

approach gives direct meaning to annotated types. The fact that this category is a model of DCG shows that every program

\[ (\mathcal{t}_2 \leftarrow (\mathcal{t}_1) H : e \quad \text{if } e \in \mathcal{t}_2 \text{ and } \mathcal{t}_2 \implies \mathcal{t}_1) \]

\[ \forall x, \exists y, \forall t \in \mathcal{t}_2 \text{, then } e : \mathcal{t}_2 \leftarrow (\mathcal{t}_1) H : e \]

Interpreting USE requires checking

Interpreting DCG holds down to

Interpreting expressions
The dependency core calculus (DCC) can also be seen as a vehicle for providing other systems correct.

Subtyping is translated into correspondence programming using `mark` and `use`Thus:

\((\tau + \tau_1)_{\mathcal{L}} \equiv \tau_2 + \tau_1 \) \hspace{1cm} \((\tau \times \tau_1)_{\mathcal{L}} \equiv \tau_2 \times \tau_1 \) \hspace{1cm} \((\tau_2 \leftarrow \tau_1)_{\mathcal{L}} \equiv \tau_2 \leftarrow \tau_1 \)

Easily translated down to full DCC:

A "homogeneous" type system, such as that of Heintel and Riecke (1998), is

\[
\text{use } x \in \tau_2 \quad \vdash \quad \begin{array}{l}
\tau_2 \triangleright \emptyset \\
\text{use } x : \tau_1 \vdash \emptyset \\
\text{use } x : \tau_1 \vdash \mathcal{L} : \tau_1 \vdash \emptyset \\
\cup
\end{array}
\]

\( \emptyset \triangleright \emptyset \) holds iff \( \mathcal{L} \triangleright \emptyset \) holds for each \( \mathcal{L} \) within \( \emptyset \), per security level \( \emptyset \in \mathcal{L} \)

Full DCC has one "mark" type constructor, written \( \mathcal{L} \), per security level \( \emptyset \in \mathcal{L} \)

This suppresses the need to guess what the typing rules should be.

The latter is viewed as a black box, yielding a modular proof.

Then, interacting it with the latter requires a translation.

The former can be expressed as an Instrumented Semantics.

Compose a dynamic dependency analysis with a static type checker.
Defining the labelled calculus

\[(\text{\textcopyright} \cdot \text{\textcopyright}) : \text{\textcopyright} \leftarrow ((\text{\textcopyright} \cdot \text{\textcopyright}) (\text{\textcopyright} \cdot \text{\textcopyright}) (\text{\textcopyright} \cdot \text{\textcopyright})) : \text{\textcopyright} \leftarrow (\text{\textcopyright} \cdot \text{\textcopyright}) (((\text{\textcopyright} \cdot \text{\textcopyright}) (\text{\textcopyright} \cdot \text{\textcopyright}) : \text{\textcopyright}) \leftarrow \text{\textcopyright}) \leftarrow \text{\textcopyright})\]

For instance,

\[(\text{\textcopyright} \cdot \text{\textcopyright}) (e_1 \cdot e_2) : e_1 \leftarrow e_1 (e_1 : e)\]

Operational semantics:

\[(e \in e) \mid \cdots \mid e \in e \mid \text{let } e = e \mid (e : e) \mid e \cdot e \mid x =: e\]


Defining the labelled calculus
Stability. Assume $e$ is a prefix and $f$ is an expression. If $\varepsilon \in e$ and $\varepsilon \notin f$, then $f \leftarrow e$.

Monotonicity. Let $\varepsilon$, $\varepsilon'$ be prefixes such that $\varepsilon \supseteq e \supseteq \varepsilon'$. If $f$ is an expression such obtained from $e$ by replacing some sub-terms with holes, prefixes are defined by autonomous expressions with a hole — write $e \supset e_1 \supset e_2$ if $e$ is
The labelled calculus approach

\[ \text{join} \quad m \sqcap l \quad \leftrightarrow \quad m \uplus l \]

The target calculus must have label constants, and a \text{join} operation: \text{join}

\text{Joining multiple labels, } \text{exploiting the fact that } \mathcal{J} \text{ that is a lattice.}

- For homogeneity, every value should carry exactly one label, which requires

- A labelled value is mapped to a \textit{pair} of the value and its label.

\text{A translation must map the labelled } \lambda\text{-calculus into a more standard } \lambda\text{-calculus.}

\text{Defining the translation}
Translation

\[(\mathcal{T} \odot \mathcal{I})'x\]

\[
\begin{align*}
\text{open } \mathcal{T} 'x \text{ as } \mathcal{E} & \rightarrow \mathcal{I} = \mathcal{E} : \mathcal{I} \\
\left[\mathcal{Z} \mathcal{E}\right] \text{ in } \left[\mathcal{E} \mathcal{I}\right] = x & \rightarrow \mathcal{E} \mathcal{I} = x \mathcal{E} = x \mathcal{I} \\
\end{align*}
\]

\[(n \odot \mathcal{T} '\mathcal{I})\]

\[
\begin{align*}
\text{open } n '\mathcal{I} \text{ as } \mathcal{Z} \mathcal{E} & \rightarrow x \mathcal{E} = \mathcal{Z} \mathcal{E} \\
\text{open } \mathcal{T} 'x \text{ as } \mathcal{E} \mathcal{I} & \rightarrow \mathcal{Z} \mathcal{E} \mathcal{I} \\
(\top ' [\mathcal{E}] \mathcal{X} \mathcal{V}) & \rightarrow [\mathcal{E} \mathcal{X} \mathcal{V}] \\
x & \rightarrow [x] \\
(\top ' \mathcal{Y}) & \rightarrow [\mathcal{Y}]
\end{align*}
\]
The labelled calculus approach

\[
\begin{align*}
\varepsilon & \equiv \varepsilon \uplus \top \\
(\varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon_3) & \equiv (\varepsilon_1 \uplus \varepsilon_2) \uplus \varepsilon_3 \\
\varepsilon & \equiv (\text{test }, \varepsilon )
\end{align*}
\]

Correctness of the translation

Simulation. If \( \varepsilon \) then \( f \leftarrow [f] \) reduces to \([\varepsilon]\) modulo an administrative congruence, whose axioms include:

\[
\text{Translation}
\]
Axiomatising a type system for the target calculus

The labelled calculus approach

\[ n : f \quad \text{iff} \quad n \times t : f(t) \]

There is a type function \( x \times t \) such that \( \text{(e) } \).

\[ \text{There is a type int. A value satisfies } \text{val} \quad \text{iff} \quad \text{it is an integer constant.} \]

\[ \text{Labels are types; } \text{t} : \text{Int} \quad \text{implies } \text{t} : \text{value.} \]

Every well-typed, irreducible expression is a value.

\[ \text{Reduction and admissible configuration preserve types.} \]

\[ \text{Among our requirements are} \]

\[ \text{t} : \text{Int} \quad \text{with} \quad \text{t} : \text{Int} \quad \text{and} \quad \text{expression} \]

\[ \text{closed expressions and types, written } \epsilon : \text{t}. \]

\[ \text{For the sake of modularity, we view a type system as an opaque set of types,} \]

Axiomatising a type system for the target calculus
Putting it all together

The labelled calculus approach

Putting it all together

which, by stability, implies $\exists v. ~ [\sigma] v \rightarrow \exists v. ~ \exists u. ~ \beta \in \gamma \land \forall \eta \in \gamma. \neg \forall \eta \in \gamma. \exists u. ~ \beta \in \gamma$

which implies $\exists u. ~ \beta \in \gamma \land \forall \eta \in \gamma. \neg \forall \eta \in \gamma. \exists u. ~ \beta \in \gamma$

which implies $\exists u. ~ \beta \in \gamma \land \forall \eta \in \gamma. \neg \forall \eta \in \gamma. \exists u. ~ \beta \in \gamma$

which implies $\exists u. ~ \beta \in \gamma \land \forall \eta \in \gamma. \neg \forall \eta \in \gamma. \exists u. ~ \beta \in \gamma$

which implies $\exists u. ~ \beta \in \gamma \land \forall \eta \in \gamma. \neg \forall \eta \in \gamma. \exists u. ~ \beta \in \gamma$

which implies $\exists u. ~ \beta \in \gamma \land \forall \eta \in \gamma. \neg \forall \eta \in \gamma. \exists u. ~ \beta \in \gamma$

Thus, $\forall \eta \in \gamma. \exists u. ~ \beta \in \gamma$

Proof. Subject reduction yields $\forall \eta \in \gamma. \exists u. ~ \beta \in \gamma$

Non-interference. If $e : \mathit{int} \land e : \mathit{int} \land e : \mathit{int}$ then $e \in \mathit{int} \land e : \mathit{int} \land e : \mathit{int}$

Thus, $\forall \eta \in \gamma. \exists u. ~ \beta \in \gamma$

Given a source expression $e$, let $e : \mathit{int} \land e : \mathit{int}$ hold if and only if $[\sigma] \exists e : \mathit{int} \land e : \mathit{int}$ holds.

Progress is straightforward.

Subject reduction is an immediate consequence of the simulation lemma and of

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

<table>
<thead>
<tr>
<th>$\varphi$</th>
<th>$\psi$</th>
<th>$\chi$</th>
<th>$\Delta$</th>
<th>$\Xi$</th>
<th>$\Omega$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vdash \phi$</td>
<td>$\vdash \psi$</td>
<td>$\vdash \chi$</td>
<td>$\Omega \vdash \Delta$</td>
<td>$\Xi \vdash \Omega$</td>
<td></td>
</tr>
</tbody>
</table>

**Labeling:**

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

| $\vdash \phi$ | $\Phi + \Phi \times \Phi$ | $\Phi \leftrightarrow \Phi$ | $\Phi \equiv \text{int}$ |

**Types are given by**

Example: a simply-typed $\lambda$-calculus with subtyping.
Example: a simply-typed $\lambda$-calculus with subtyping

The labelled calculus approach

$$\llbracket \cdot \rrbracket \Downarrow S \quad S + S \mid S \times S \mid S \leftarrow S \mid \text{int} \Downarrow \Downarrow \neg \$$

One may write $\llbracket \cdot \rrbracket$ for $\llbracket \cdot \rrbracket \times \Downarrow$ and require all types to be generated by

$$(\mathcal{L} \cap \mathcal{L}) \times \llbracket \cdot \rrbracket \Downarrow \Downarrow$$

App

$$(\mathcal{L} \cap \mathcal{L}) \times \llbracket \cdot \rrbracket \Downarrow$$

$$(\mathcal{L} \cap \mathcal{L}) \times \llbracket \cdot \rrbracket \Downarrow \Downarrow$$

$$(\mathcal{L} \cap \mathcal{L}) \times \llbracket \cdot \rrbracket \Downarrow$$

Composing the translation rules with the typing rules.

Deriving Direct Rules

31
In short, the labeled calculus approach introduces labeled semantics. In many settings, type expressions can carry different labels, and inference rules govern their manipulation. This approach gives an operational intuition for the distinction between int and int functions. For example, a homogeneous type system.

However, the fact that the translation must be very simple imposes some design choices (e.g., a homogeneous type system). The approach is applicable to other calculi, e.g., the  \( \forall \)-calculus under System F.

The proof is modular: it also yields polymorphic constraint-based type

In short
No correctness proof is given.

- No global variables.
- Dynamic allocation or exceptions.
- Assignment, sequence, if, while, goto.
- Read-only or read/write parameters and local variables of base or array type.
- First-order, recursive, polymorphic procedures.

D. E. Dennie address a programming language equipped with:

Dennine's static analysis (1975–82)
timeout \leftarrow (\text{timeout} \times \text{timeout}) \times \text{timeout} \times \text{timeout} \times \text{timeout}

\max : \forall x, y, m' \vdash m' \geq x \land y \geq x

In type-theoretic notation, we would write:

\begin{verbatim}
begin
\forall x \text{ then } \forall y \text{ else } m := x \text{ if } x < m
procedure max (x); \forall m \text{ var } m \text{ in } \{ x, \ldots \};
\end{verbatim}

\begin{verbatim}
\end{verbatim}

Given:
The set of parameters on which every writable parameter depends must be

Specifying procedures
Specifying procedures

\[ \text{swap} : \forall x. (\exists x. \text{ref}) \times (\exists x. \text{ref}) \leftarrow (\exists x. \text{ref}) \times (\exists x. \text{ref}) \]

In type-theoretic notation, we would write:

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

The level of local variables must also be declared:
A procedure call is secure if flows between formals induce valid flows between actuals.

A conditional if-then $S$ is secure if

\[ \overline{x} \subseteq \overline{e} \quad \text{(indirect flow)} \]

for every variable $x$ that may be assigned within $S$, $e$.

$S$ is secure if

\[ S_1 \land S_2 \quad \text{are both secure.} \]

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

An assignment $x := e$ is secure if $\overline{e} \subseteq \overline{x}$ \quad \text{(direct flow)}

checks that every statement is secure (i.e., does not cause leaks).

From these definitions, one determines the information level of every expression $e$, written $\overline{e}$, a subset of the procedure's parameters. Then, one

Cheking procedures
Checking Procedures

It is insecure. Its specification should be amended by declaring

\[
\text{var } z \text{ } \{ \text{ x } \} \text{ var } y \text{ } \{ \text{ x } \}
\]

\end

\[
\text{\texttt{if then 0 = z \texttt{ end};} =: \texttt{z \texttt{ then 0 = z \texttt{ end;}}}
\]

\[
\text{\texttt{if then 0 = z \texttt{ then 0 = z \texttt{ end;}}} =: \texttt{z \texttt{ then 0 = z \texttt{ end;}}}
\]

\[
\text{\texttt{y \texttt{ =: z \texttt{ then 0 = z \texttt{ end;}}}}}
\]

\begin{procedure}
\text{\texttt{begin}}
\text{\texttt{var z;}}
\text{\texttt{procedure copy (x, var y)}}
\end{procedure}

The following procedure copies \( x \) into \( y \) assuming \( x \) \( y \) is initially \( 0 \) or \( 1 \).
Checking Procedures

Denning's static analyser

Towards a Generalization: pc becomes an additional (implicit) formal parameter to every procedure.

\[ \overline{s} \cap \overline{pc} \ni \text{if } S \text{ is secure at } pc \text{ if } \overline{s} \ni \text{secure at } pc \]

A conditional if-then S is secure at pc if S1 and S2 are both secure at pc.

An assignment \( x \vdash e \) is secure at pc if pc \( \ni \overline{x} \geq \overline{e} \).

Thus, Hentze and Riecke (1998) and Myers (1999) parameterise the judgment "S is secure" with an information level, written pc. This is no longer true in the presence of, say, first-class references and functions.

In Denning's restricted language, the set of variables that may be assigned
Proposed by Potter and Simmonet (2002).

prove it correct.

combine these ideas into a type system that supports type inference, and to

imperative languages. First-class references are addressed by Hiehle and Pitcher

We have described analyses for a purely functional language and for a simple
The semantics is call-by-value.

\[
[e] \text{bind } x \text{ in } e \rightarrow \begin{cases} \text{true} & \text{if } \text{let } e \rightarrow e_1 \text{ in } e_2 \text{ case } e_1 \text{ of } \text{Proj } a \rightarrow a_1 \mid a_2 \rightarrow \text{ret } a \mid a_v \rightarrow a \rightarrow a \rightarrow \text{raise } e \rightarrow a \mid (a, a) \rightarrow a_1 \mid a_2 \rightarrow e_{\chi} \rightarrow \chi(a_1, a_2) \mid () \rightarrow x \rightarrow a \end{cases}
\]
with the same consequences, this time via projection.

\[ \phi \cdot \chi \leftarrow \langle 68 \mid \mathcal{L} \rangle (\phi \cdot \chi \cdot \chi) \]

ensures, we will reason directly about two processes that share some structure.

which implies (by monotonicity and

\[ (\phi \cdot \chi) : \top \leftarrow (\top : \neg \mathbf{H}) ((\phi \cdot \chi \cdot \chi) : \top) \]

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

Problematic:

when it is skipped. As a result, designing a labeled semantics becomes

The statement \( \text{if } x \rightarrow z \text{ then } y = 0 \) causes information to flow from \( x \) to \( z \), even

Trouble with labels
The bracket calculus

Information flow inference for ML

Two projection functions map a ML\textsuperscript{2} term to the two ML terms which it encodes. In particular, \( \langle \exists n \mid \alpha \rangle \). To encode the pair \( \langle n \mid \alpha \rangle \) and \( \langle \exists n \mid \alpha \rangle \), the \textsuperscript{2}ML term encodes a pair of ML terms. For instance, for instance, \( \langle a \mid a \rangle \) is denoted as an extension of \textsuperscript{2}ML.

\[ \langle a \mid a \rangle | \cdots =:: a \]

\[ \langle a \mid a \rangle | \cdots =:: a \]

\[ \text{void} | \langle a \mid a \rangle | \cdots =:: a \]

The language \textsuperscript{2}ML\textsuperscript{2} is defined as an extension of \textsuperscript{2}ML.
The bracket calculus

Informatio: How inference for ML

Semantics of ML:

\[
\begin{align*}
\text{(pro-log)} \quad \langle \! \langle \varphi \text{, } \Gamma \mapsto \psi \rangle \! \rangle \ & \leftarrow \ \langle \! \langle \varphi \text{, } \Gamma \rangle \! \rangle \nabla \\
\text{(pro-log)} \quad \langle \! \langle \psi \rangle \! \rangle \ & \leftarrow \ \langle \! \langle \varphi \text{, } \Gamma \rangle \! \rangle \nabla \\
\text{(pro-log)} \quad \langle \! \langle \varphi \text{, } \Gamma \mapsto \psi \rangle \! \rangle \ & \leftarrow \ \langle \! \langle \varphi \text{, } \Gamma \rangle \! \rangle \nabla \\
\text{(pro-log)} \quad \langle \! \langle \psi \rangle \! \rangle \ & \leftarrow \ \langle \! \langle \varphi \text{, } \Gamma \rangle \! \rangle \nabla
\end{align*}
\]

the way. The latter are no-ops w.r.t. the term’s projections. Two reduction rules: a standard one, and one that moves (lifts) brackets out of

As in the labelled λ-calculus, each language construct is typically dealt with by
The bracket calculus

made as precise as possible.

The "hill" rules provide an explicit description of information flow, and must be specified as "secret":

while computationally correct, would cause the type system to view every

\[ e \leftarrow e \in \{ 0, 1 \} \]

The (hypothetical) reduction rule.

Brackets encode the differences between two programs, i.e., their "secret" parts.
The bracket calculus

\[
\frac{\pi / \langle e_2 | e_1 \rangle \leftrightarrow \pi / \langle e_2 | e_1 \rangle}{\{2, 1\} = \{2, 1\} \quad e \quad \pi / \langle e_2 | e_1 \rangle \leftrightarrow \pi / e}
\]

where \( \pi \in \{2, 1\} \). Write \( \pi / \cdot \) for \( \pi / \cdot \).

Reductions which take place inside a construct must use or affect only one

\[\pi / \cdot \] situation of the store. For this purpose, let configurations be of the form \( \pi / \cdot \). \n
Reductions which take place inside a construct must use or affect only one

\[\pi / \cdot \] situation of the store. For this purpose, let configurations be of the form \( \pi / \cdot \). \n
Reductions which take place inside a construct must use or affect only one

\[\pi / \cdot \] situation of the store. For this purpose, let configurations be of the form \( \pi / \cdot \). \n
Reductions which take place inside a construct must use or affect only one

\[\pi / \cdot \] situation of the store. For this purpose, let configurations be of the form \( \pi / \cdot \). \n
Reductions which take place inside a construct must use or affect only one

\[\pi / \cdot \] situation of the store. For this purpose, let configurations be of the form \( \pi / \cdot \). \n
Reductions which take place inside a construct must use or affect only one

\[\pi / \cdot \] situation of the store. For this purpose, let configurations be of the form \( \pi / \cdot \).
Analogous rules are given for dynamic allocation and dereferencing:

\[
\begin{align*}
& (\text{init-assgn}) & n / \langle \exists a \mid n[a] =: 1 \rangle & = n / \langle 1 \rangle \\
& (\text{assign}) & [a \leftarrow w]n \leftrightarrow w & = n / a =: w
\end{align*}
\]

Then, the reduction rules for assignment are:

\[
\begin{align*}
& \langle \exists a \mid n[a] \rangle = \text{update} z \cdot a \\
& \langle \exists a \mid n[a] \rangle = \text{update} \cdot a \\
& n = \text{update} \cdot a
\end{align*}
\]

Define the following auxiliary function:
\[
\langle 0 \mid I \rangle \leftrightarrow z, \langle I \mid 0 \rangle \leftrightarrow x / \langle () \mid () \rangle \leftarrow \\
0 \leftarrow z, \langle I \mid 0 \rangle \leftrightarrow x / \langle I \mid =: z \begin{cases} \text{false} & \text{then} \langle false \mid I \rangle \\ \text{true} & \text{then} \langle true \mid I \rangle \end{cases} \rangle \leftarrow \\
0 \leftarrow z, \langle I \mid 0 \rangle \leftrightarrow x / I =: z \begin{cases} \text{then} \langle 0 = I \mid 0 = 0 \rangle & \text{false} \\ \text{then} \langle I \mid 0 \rangle & \text{false} \end{cases} \langle \rangle \leftarrow \\
0 \leftarrow z, \langle I \mid 0 \rangle \leftrightarrow x / I =: z \begin{cases} \text{then} \langle 0 = I \mid 0 = 0 \rangle & \text{false} \\ \text{then} \langle I \mid 0 \rangle & \text{false} \end{cases} \langle \rangle \leftarrow \\
0 \leftarrow z, \langle I \mid 0 \rangle \leftrightarrow x / I =: z \begin{cases} \text{then} \langle 0 = I \mid 0 = 0 \rangle & \text{false} \\ \text{then} \langle I \mid 0 \rangle & \text{false} \end{cases} \langle \rangle \leftarrow \]

Example
if none of the above rules applies

\( n/\langle [\alpha]z [\mathcal{A}] \mid [\alpha]1 [\mathcal{A}] \rangle \quad \leftarrow \quad n/\langle \langle \alpha \rangle \mid [\alpha]1 \rangle \mathcal{T} \)

if \( \mathcal{E} \) handles neither \( \alpha \) nor \( \alpha \)

\( (\text{dod}) \quad \alpha \quad \leftarrow \quad [\alpha]\mathcal{E} \)

\( \text{Finally} \) \( \quad \alpha ; \quad \varepsilon \quad \leftarrow \quad \alpha \text{ Finally } \varepsilon \)

\( \text{Handle-Raise} \) \quad \alpha \text{ raise } \varepsilon \quad \leftarrow \quad \text{raise } \varepsilon \text{ handle } \alpha \text{ raise } \varepsilon

\( \text{Handle-Done} \) \quad \varepsilon \quad \leftarrow \quad \text{raise } \varepsilon \text{ handle } \alpha \text{ done } \varepsilon

\( \text{Handle} \) \quad [\alpha \Rightarrow x]\varepsilon \quad \leftarrow \quad \varepsilon < x \quad \varepsilon \text{ handle } \varepsilon \text{ raise } \alpha \text{ handle } \varepsilon \text{ raise } x

\( \text{Bind} \) \quad [\alpha \Rightarrow x]\varepsilon \quad \leftarrow \quad \varepsilon \text{ in } \alpha = x \quad \text{bind } x

\)
\[
\langle () \in \text{raise} \mid (0, x) \rangle \quad \leftarrow
\langle (0, x) \rangle \quad \text{bind} \quad \Rightarrow \quad \langle () \in \text{raise} \mid (0, x) \rangle = x \quad \text{bind}
\]

\[
\langle \forall \mid x \rangle \quad \leftarrow \quad \langle \forall \mid x \rangle \quad \text{bind} \quad \Rightarrow \quad \langle \forall \mid x \rangle = x \quad \text{bind}
\]
In short, brackets cannot commute or block reduction.

\[ \begin{array}{l}
\text{A conjectural conjecture is such that } \forall n \in \mathbb{N} \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exist
The types of ML

\[ 3 \equiv \{ \oplus \leftrightarrow 3 \} \quad \oplus (\oplus + \oplus) \quad \oplus \times \oplus \quad \oplus \text{rel} \quad \oplus (\oplus \leftarrow [\oplus] \oplus \oplus) \quad \oplus \text{int} \]

Subtyping extends the ordering on information levels.

\[ 3 \equiv \{ \text{pc} \leftrightarrow 3 \} =:: \quad \downarrow \]

\[ \llbracket t + t \rrbracket | \llbracket t \times t \rrbracket | \llbracket t \text{rel} t \rrbracket | \llbracket t \leftarrow [u] \text{pc} t \rrbracket | \llbracket \text{unit} t \rrbracket | \llbracket \text{int} t \rrbracket =:: \quad \downarrow \]

Types and rows are defined as follows:

The types of ML
The types of $\text{ML}$:

\[
\frac{\mathcal{E}(\ast + \ast) \triangleright \mathcal{F}}{\mathcal{E} \supseteq \mathcal{F}} \quad \frac{\mathcal{E} \times \mathcal{F_1} \triangleright \mathcal{F}}{\mathcal{E} \triangleright \mathcal{F_1} \triangleright \mathcal{F}} \quad \frac{\text{refl}}{\mathcal{E} \supseteq \mathcal{F}}
\]

\[
\frac{\mathcal{E}(\ast \left[\ast\right] \ast) \triangleright \mathcal{F}}{\mathcal{E} \supseteq \mathcal{F}} \quad \frac{\text{int}}{\mathcal{E} \triangleright \mathcal{F}} \quad \text{unit} \triangleright \mathcal{F}
\]

The auxiliary predicate $\triangleright \mathcal{F}$ holds if $\mathcal{F}$ holds (guards (guards))
They are connected via the following typing rule:

\[
\begin{align*}
[\ast] & \quad \forall : \alpha 
\quad \vdash \Gamma,

\Gamma & \quad \vdash \forall

\text{E-VALUE}
\end{align*}
\]

We distinguish two forms of typing judgments: one deals with values only, the other with arbitrary expressions.
ML₂'s type system

Information flow inference for ML₂

\[ \frac{\tau : \langle \alpha \mid \beta \rangle \vdash 1}{1} \]

\[ \frac{1 \vdash \emptyset \vdash \emptyset}{1 \vdash 1} \]

\[ \text{\textsc{Bracket}} \]

\[ H \subseteq \emptyset \]

: H is guarded by some level in \( \emptyset \), the "secret" levels. The system guarantees that the type of every bound variable in the heap is parametricized over an (upward-closed) set of information. Keeping track of brackets.
\[
\begin{align*}
[\ell] & \quad \exists \gamma_1, \gamma_2 \quad \ell \vdash \gamma_1 \gamma_2 \quad \frak{L} : \exists \gamma_1, \gamma_2 \quad \ell \vdash \gamma_1 \gamma_2 \quad \frak{L} \\
\quad & \quad \ell \vdash \gamma_1, \gamma_2 \quad \frak{L} : \exists \gamma_1, \gamma_2 \quad \ell \vdash \gamma_1 \gamma_2 \quad \frak{L} \\
\end{align*}
\]

Information about the function may leak through its side effects (\(\frak{L} \geq \gamma \)).

\textbf{Abstract and application (}\(\frak{L} \geq \gamma \)).
[\star] \quad \text{unit} : \forall \alpha . \alpha = : \forall \alpha . \alpha \rightarrow \perp_{\forall \alpha . \alpha} \quad \text{pc}

\vdash f \triangleright f \sqcap \text{pc} \quad \vdash f : \forall \alpha . \alpha \rightarrow \perp_{\forall \alpha . \alpha} \quad \text{refl}

\text{E-ASSIGN}

\text{Leak as well (f \triangleright f).}

\text{In the presence of first-class references, information about the reference's identity may leak as well.}

\text{Leakage occurs when encoding an indirect flow. Hence, follow the defining solution (pc \triangleright f) of the information encoded within the program counter may leak when writing a}

\text{imperative constructs}
ML\textsubscript{2}'s type system

\[
\begin{array}{c}
\text{E-Raise} \\
\text{PC: } \varepsilon \quad \varepsilon : \text{Raise } \varepsilon \quad \varepsilon \vdash \text{Raise } \varepsilon \quad \varepsilon \\
\hline
\text{Type } \text{Raise } \varepsilon \\
\text{The point where the exception is raised.}
\end{array}
\]

\[
\text{The amount of information carried by the exception itself is represented by PC at }
\]

\[
\text{Type } \text{Raise } \varepsilon \\
\text{The value carried by the exception must have fixed (declared, monomorphic)}
\]

\[
\text{Raising an exception}
\]
ML$^2$'s type system

\[
\begin{align*}
\phi' \Gamma \vdash e_1 \quad \phi \vdash e_2 \quad \text{handle } \epsilon \in \phi \quad \epsilon \text{ is caught } \quad \phi \Gamma \vdash e_3 \quad \phi' \Gamma \vdash e_4 \\
\phi \vdash e_5 \quad \epsilon \text{ is caught } \quad \phi \Gamma \vdash e_6
\end{align*}
\]

E-HANDLE

So may the whole expression's result \((\phi \vdash e_7)\).

The handler's side effects may reveal that an exception was caught \((\phi \vdash e_8)\).

The amount of information carried by the exception, namely \(\phi \vdash e_9\), is read off the

Handling a specific exception

58
Computing in sequence
Information Flow Inference for MIL

To avoid keeping track of this fact, we require it to convey no information. Successfully.

However, ë¿ is required to raise not to raise (informative) exceptions. Indeed, observing

\[
\begin{align*}
[\forall] \quad \forall : \forall &\quad \text{finally} \\
[\top \exists] \quad \exists : \exists &\quad \text{F- FINALLY}
\end{align*}
\]

constrainted further than the whole expression.

An event that must occur conveys no information, so ë¿'s side effects are not

Finally
Results

\[ \text{Subject reduction: } \Gamma \vdash e \rightarrow \rho \]

Information Flow Inference for ML
Thus, this approach also gives operational meaning to the distinction between int and Int.

Lemma. Let \( H \not\vdash \epsilon \). If \( \Gamma \vdash e \colon \text{int} \) and \( x \leftarrow e \), then \( \Gamma \vdash \nu x. \) By subject reduction, any expression which may reduce to such a form assigns \( \text{high} \) security levels (i.e., levels in \( H \)) to values of the form \( \nu x. \)
Results

\[ H \not\vDash \varphi \text{ yields } \not\exists \eta \text{ such that } \eta \supseteq \varphi \]

The result follows by the previous lemma.

For all \( \forall \varphi \in \mathcal{L} \), which implies that \( \varphi \) is a value. Lastly, \( \not\exists \eta \) yields \( \not\exists \eta \).

By completeness, there exists an answer \( x \) such that \( \forall \varphi \in \mathcal{L} \).

If \( \forall \varphi \in \mathcal{L} \).

Beware. A substitution lemma yields \( \not\exists \eta \text{ holds.} \)

Define \( \{ \eta \} \triangleright H \).

Proof. Let \( \exists \eta \in [\varphi \Rightarrow \text{Int}] \).

\[ \exists \eta \in [\varphi \Rightarrow \text{Int}] \]

Then, \( \forall \varphi \in \mathcal{L} \).

\[ \forall \varphi \in \mathcal{L} \]

and \( \forall \varphi \in \mathcal{L} \).

Choose \( \eta \in \mathcal{L} \) such that \( \eta \not\supseteq \varphi \). Let \( \eta \not\supseteq \varphi \).

Non-interference.
Type inference reduces to consistent solving.

\[
\begin{array}{c|c|c|c|c|c}
\top \supset \chi & \chi \supset \chi & \neg \top \supset \top & \chi_0 \vDash_\mathcal{E} & \chi \land \chi & \text{true false} & =:: & \mathcal{C} \\
\chi & d \supset d & \top \supset \top & \xi & \xi & \xi & =:: & \chi \\
\end{array}
\]

Constraints:

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
The usual MTL definition of lists:

\[ \text{list} = \text{unit} \]

\[ ((\text{list} \times g) + \text{unit}) = g \]

must be decorated:

\[ ((\text{list} \times g) + \text{unit}) = g \]

Examples
Examples

Information flow inference for ML

```
val iter : int = function
  let rec iter = function
    val length : int = length
    in
      iter
    end

val g = list [1, 2, 3]

unit "g" = unit [g]
```

```
Language-based information flow control?

Is it feasible?

Never put into practice so far;

Stirs much interest;

25 years old.

Language-based information flow control?
and Integrity. POPL’98.
Nevin Heimizte and Jon C. Riecke. The Slam Calculus: Programming with Security
Joseph Coguen and Jose Meseguer. Security policies and security models. SCP’82.
tection of information flow in sequential programs. EROSICS’94.
Jean-Pierre Banatre, Gianri Bivace and Daniell Le Metayer. Compile-time
Dependences. ICFP’96.
Martin Abadi, Butler Lampson and Jean-Jacques Levy. Analyses and Caching of
Calculus of Dependence. POPL’99.
Martin Abadi, Anindya Banerjee, Nevin Heimizte and Jon C. Riecke. A Core
Selected References
• Tapsopt’97. Dennis Volpano and Geoffrey Smith. A Type-Based Approach to Program Security.


• Francois Pottier and Vincent Simonet. Information Flow Inference for ML. ICFP’00.

• Francois Pottier and Sylvain Conchon. Information Flow Inference for Free. ICFP’00.

• Peter O’Hare and Jean-Paul Bertho. Trust in the X-calculus. JFP 7:6, 1997.
