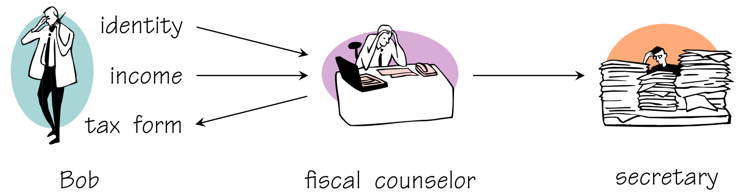
Type-Based Information Flow Analyses

François Pottier

June 23–25, 2004

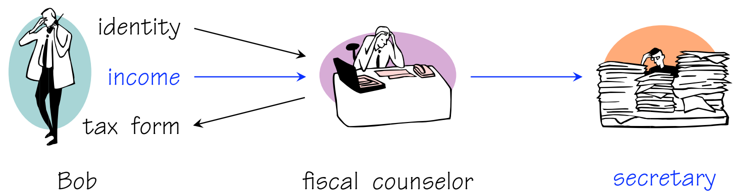
INRIA

Why control information flow?



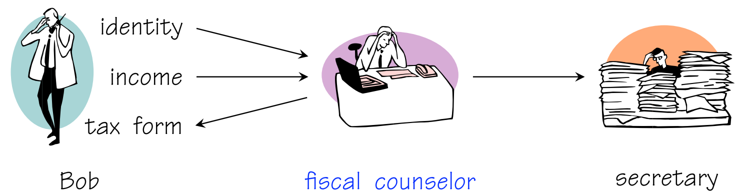
Here is a common real life scenario.

Why control information flow?



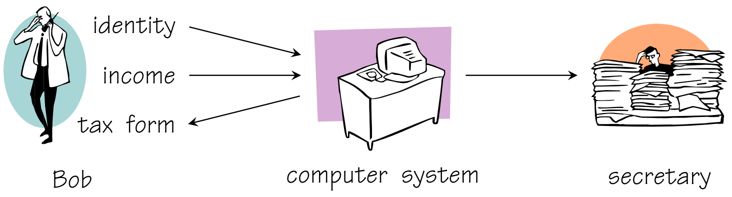
Bob wishes the secretary to have no *idea* of his income.

Why control information flow?



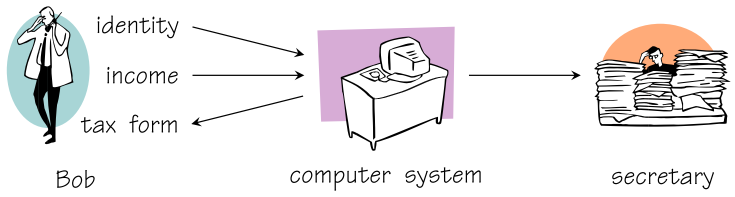
Bob has to *trust* his counselor.

Why control information flow?



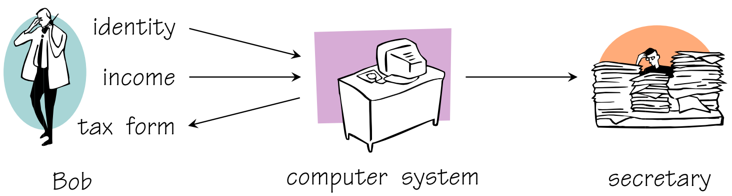
The fiscal counselor is replaced...

Why control information flow?



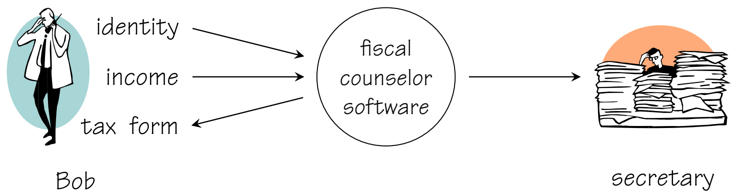
Bob must still trust the system not to leak information. *Access control* does not help.

Why control information flow?



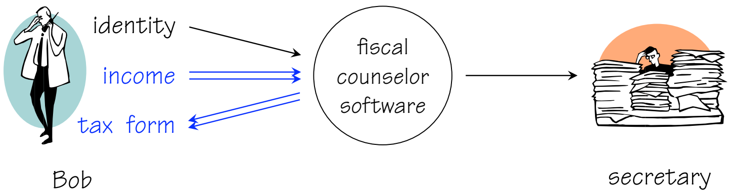
Information flow control requires the computer system to conform to Bob's confidentiality policy.

Why control information flow?



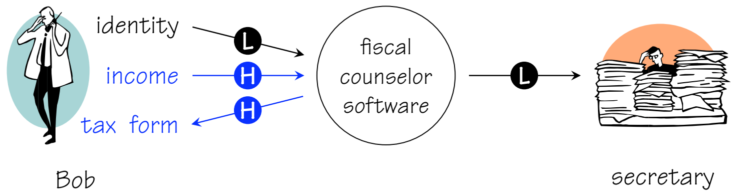
Language-based information flow control consists of an analysis of *a single piece of software* with a well-defined semantics.

Why control information flow?



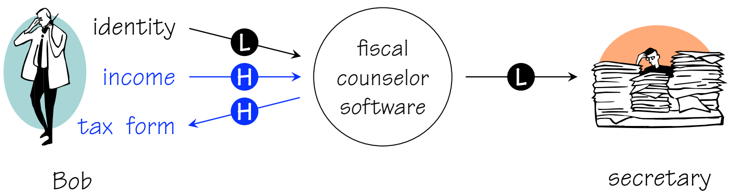
The property that Bob desires is *noninterference*: even if he were to supply a different income figure, the secretary wouldn't be able to tell the difference. In other words, the data sent to the secretary does not *depend* on Bob's income.

Why control information flow?



To specify this property succinctly, one assigns ordered information *levels* to each input and output channel — here, **L** < **H** — and one allows information to flow *up* only.

Why control information flow?



The specification may take the form of a *type*, such as

$$\text{string}^{L} \times \text{int}^{H} \longrightarrow \text{string}^{H} \times \text{int}^{L}$$

Part I

Information flow in pure functional languages

An overview of DCC

DCC with multiple security levels

A proof by encoding into DCC

A direct, syntactic proof

An overview of DCC

DCC with multiple security levels

A proof by encoding into DCC

A direct, syntactic proof

For simplicity, I assume that the security lattice $\mathcal{L}$ consists of two levels **L** and **H**, ordered by $\mathbf{L} \leq \mathbf{H}$. Next, I will move to the general case.

Syntax

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

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

In the semantics, these constructs are no-ops.

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

Types

DCC is a standard simply-typed λ-calculus. Only the two new constructs have nonstandard typing rules.

Mark
$$\frac{\Gamma \vdash e : t}{\Gamma \vdash (\mathbf{H} : e) : \mathbf{H}(t)}$$

Use
$$\frac{\Gamma \vdash e_1 : \mathbf{H}(t_1) \qquad \Gamma; x : t_1 \vdash e_2 : t_2 \qquad \lhd\, t_2}{\Gamma \vdash \mathbf{use}\ x = e_1\ \mathbf{in}\ e_2 : t_2}$$

When marking a value, its type is marked as well.

When using a marked value, the mark is taken off its type, but the end result must have a *protected* type.

An overview of DCC DCC with multiple security levels A proof by encoding into DCC A direct, syntactic proof

Protected types

The predicate $\lhd\, t$ ("t is protected") is defined inductively.

Intuitively, the information carried by a value of a protected type t must be accessible only to high-level observers.

Protected types form a superset of the marked types, that is, *every marked type is protected*:

$$\lhd\, \mathbf{H}(t)$$

Furthermore, some types that do *not* carry a mark at their root may safely be considered protected.

François Pottier Type-Based Information Flow Analyses

An overview of DCC DCC with multiple security levels A proof by encoding into DCC A direct, syntactic proof

Example

Define bool as unit + unit. Define **if**, **true**, and **false** accordingly.

This function negates a high-security Boolean value:

$$\lambda x.\mathbf{use}\ x = x\ \mathbf{in}\ \mathbf{H} : (\mathbf{if}\ x\ \mathbf{then\ false\ else\ true}) : \mathbf{H}(\text{bool}) \rightarrow \mathbf{H}(\text{bool})$$

François Pottier Type-Based Information Flow Analyses

An overview of DCC DCC with multiple security levels A proof by encoding into DCC A direct, syntactic proof

Protected types, continued

For instance, a function type is protected if its codomain is protected:

$$\frac{\lhd\, t_2}{\lhd\, t_1 \rightarrow t_2}$$

This makes intuitive sense because the only way of obtaining information out of a function is to observe its result.

François Pottier Type-Based Information Flow Analyses

Protected types, continued

A product type is protected if both its components are protected:

$$\frac{\lhd\, t_1 \qquad \lhd\, t_2}{\lhd\, t_1 \times t_2}$$

This makes intuitive sense because the only way of obtaining information out of a pair is to observe its components.

Protected types, continued

The unit type is protected:

$$\lhd\, \text{unit}$$

This makes intuitive sense because there is no way of obtaining information out of the unit value.

PER Basics

Definition

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.

I write $x : R$ for $x\,R\,x$. I write $R \rightarrow R'$ for the relation defined by

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

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 $f : t \rightarrow u$.

As usual, types are interpreted by objects, and typing judgements by morphisms.

In particular, a typing judgement of the form $\vdash e : t$ is interpreted as an element e of $|t|$ such that $e : t$, that is, e is related to itself by the PER t.

The intuition behind PERs

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

For instance, consider the flat cpo bool $= \{\mathbf{true}, \mathbf{false}\}$.

The object boolL is obtained by equipping bool with the *diagonal* relation.

The object boolH is obtained by equipping bool with the *everywhere true* relation.

The intuition behind morphisms

The requirement that every morphism f from t to u satisfy $f : t \twoheadrightarrow u$ is a *noninterference* statement.

For instance, the assertion $f : \text{boolH} \twoheadrightarrow \text{boolL}$ is syntactic sugar for

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

that is,

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

which requires f to *ignore* its argument.

Interpreting types

The interpretation of the type constructors $\rightarrow$, $\times$ and $+$ is standard.

The marked type $\mathbf{H}(t)$ is interpreted as the cpo $|t|$, equipped with the *everywhere true* relation.

Then, the low-level observer's view of *every protected type* is the everywhere true relation:

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

Interpreting typing judgements

$$\text{Mark}\ \frac{\Gamma \vdash e : t}{\Gamma \vdash (\mathbf{H} : e) : \mathbf{H}(t)}$$

Interpreting Mark boils down to

Lemma
$e : t$ implies $e : \mathbf{H}(t)$.

Proof.
The PER $\mathbf{H}(t)$ is everywhere true. □

Interpreting typing judgements, continued

Use

$$\frac{\Gamma \vdash e_1 : \mathbf{H}(t_1) \qquad \Gamma; x : t_1 \vdash e_2 : t_2 \qquad \lhd\, t_2}{\Gamma \vdash \mathbf{use}\ x = e_1\ \mathbf{in}\ e_2 : t_2}$$

Interpreting Use boils down to

Lemma

$e : t_1 \rightarrow t_2$ and $\lhd\, t_2$ imply $e : \mathbf{H}(t_1) \rightarrow t_2$.

Proof.

The type t_2 is protected, so the PER t_2 is everywhere true. As a result, we have $\forall x, y \quad x\ \mathbf{H}(t_1)\ y \Rightarrow (e\,x)\ t_2\ (e\,y)$, that is, $e : \mathbf{H}(t_1) \rightarrow t_2$. □

Interpreting typing judgements, continued

Thus:

Theorem

This category is a model of DCC.

This shows that every program satisfies the *noninterference* assertion encoded by its type.

The PER approach gives *direct* meaning to annotated types.

Syntax

In fact, DCC is defined on top of an arbitrary security lattice $\mathcal{L}$. A value may be marked with any security level ℓ.

$$\begin{array}{lcl} e & ::= & x \mid \lambda x.e \mid e\,e \mid \ldots \mid \ell : e \mid \mathbf{use}\ x = e\ \mathbf{in}\ e \\ t & ::= & t \rightarrow t \mid \mathrm{unit} \mid t + t \mid t \times t \mid T_\ell(t) \end{array}$$

Types

The typing rules are generalized as follows:

Mark

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash (\ell : e) : T_\ell(t)}$$

Use

$$\frac{\Gamma \vdash e_1 : T_\ell(t_1) \qquad \Gamma; x : t_1 \vdash e_2 : t_2 \qquad \ell \lhd t_2}{\Gamma \vdash \textbf{use } x = e_1 \textbf{ in } e_2 : t_2}$$

Protected types

The predicate $\ell \lhd t$ ("t is protected at level ℓ" or "ℓ guards t") is defined inductively.

Intuitively, when ℓ guards t, the information carried by a value of type t must be accessible only to observers of level ℓ or greater.

$$\frac{\ell \leq \ell' \vee \ell \lhd t}{\ell \lhd T_{\ell'}(t)} \qquad \frac{\ell \lhd t_2}{\ell \lhd t_1 \rightarrow t_2} \qquad \frac{\ell \lhd t_1 \qquad \ell \lhd t_2}{\ell \lhd t_1 \times t_2} \qquad \ell \lhd \text{unit}$$

Subtyping

Technically, DCC does *not* have subtyping, because it can be simulated using *coercions*, that is, functions that have no computational content.

For instance, whenever $\ell \leq \ell'$ holds, we have

$$\lambda x.\textbf{use } x = x \textbf{ in } (\ell' : x) \quad : \quad T_\ell(t) \rightarrow T_{\ell'}(t)$$

The very existence of coercions indicates that the addition of true subtyping would be compatible with the model.

DCC as a target language

Writing programs in DCC is hard, because explicit uses of Mark and Use must be inserted by the programmer.

One really wants a programming language with *no* ad hoc term constructs, where all security-related information is carried by ad hoc *types*.

In other words, one needs an *ad hoc* type system for a *standard* term language. This is what I refer to as "type-based information flow analysis."

DCC as a target language, continued

To prove the correctness of the ad hoc type system, one exhibits a semantics-preserving encoding of it into DCC. Thus, DCC may be viewed as a *target language* for proving the correctness of several type-based information flow analyses.

A simple ad hoc type system

To illustrate the idea, I now define a nonstandard type system for a standard λ-calculus with products and sums.

Since the calculus is standard, a distinguished type constructor T_ℓ would not make any sense. Instead, *some*, but *not necessarily all*, of the standard type constructors must now carry a security level.

Types

For instance, let

$$t ::= \text{unit} \mid t \rightarrow t \mid t \times t \mid (t + t)^\ell$$

The encoding of types into DCC is

$$\begin{aligned} [\![\text{unit}]\!] &= \text{unit} \\ [\![t_1 \rightarrow t_2]\!] &= [\![t_1]\!] \rightarrow [\![t_2]\!] \\ [\![t_1 \times t_2]\!] &= [\![t_1]\!] \times [\![t_2]\!] \\ [\![(t_1 + t_2)^\ell]\!] &= T_\ell([\![t_1]\!] + [\![t_2]\!]) \end{aligned}$$

Protected types

As in DCC, I define the predicate $\ell \lhd t$.

$$\ell \lhd \text{unit} \qquad \frac{\ell \lhd t_2}{\ell \lhd t_1 \rightarrow t_2} \qquad \frac{\ell \lhd t_1 \quad \ell \lhd t_2}{\ell \lhd t_1 \times t_2} \qquad \frac{\ell \leq \ell'}{\ell \lhd (t_1 + t_2)^{\ell'}}$$

This definition is correct with respect to DCC:

Lemma
$\ell \lhd t$ implies $\ell \lhd [\![t]\!]$.

Sums

All typing rules are standard, except those that deal with sums:

$$\frac{\Gamma \vdash e : t_i}{\Gamma \vdash \mathbf{inj}_i\ e : (t_1 + t_2)^{\ell}} \qquad \frac{\Gamma \vdash e : (t_1 + t_2)^{\ell} \quad \ell \lhd t' \quad \forall i \in \{1,2\} \quad \Gamma; x : t_i \vdash e_i : t'}{\Gamma \vdash e\ \mathbf{case}\ x \succ e_1\ e_2 : t'}$$

These suggest the following encoding of expressions:

$$[\![\mathbf{inj}_i\ e]\!] = \ell : (\mathbf{inj}_i\ [\![e]\!])$$
$$[\![e\ \mathbf{case}\ x \succ e_1\ e_2]\!] = \mathbf{use}\ x = [\![e]\!]\ \mathbf{in}\ (x\ \mathbf{case}\ x \succ [\![e_1]\!]\ [\![e_2]\!])$$

Subtyping

The type system may be equipped with a simple, *structural* subtyping relation, which extends the security lattice. It is succinctly defined as follows:

$$\ominus \rightarrow \oplus \qquad \oplus \times \oplus \qquad (\oplus + \oplus)^{\oplus}$$

The subtyping rule is standard:

$$\frac{\Gamma \vdash e : t \quad t \leq t'}{\Gamma \vdash e : t'}$$

Its correctness follows from:

Lemma
If $t \leq t'$ holds, then there exists a coercion of type $[\![t]\!] \rightarrow [\![t']\!]$ in DCC.

Correctness of the encoding

The correctness of the encoding is given by

Theorem
$\Gamma \vdash e : t$ implies $[\![\Gamma]\!] \vdash [\![e]\!] : [\![t]\!]$.

Theorem
e and $[\![e]\!]$ have the same semantics.

Thus, a function of type $\text{bool}^{\mathbf{H}} \rightarrow \text{bool}^{\mathbf{L}}$ behaves like a DCC function of type $T_{\mathbf{H}}(\text{bool}) \rightarrow T_{\mathbf{L}}(\text{bool})$, which I have proved must ignore its argument.

A syntactic approach

DCC is a useful tool and allows giving meaning to annotated types via PERs and logical relations.

However, the simple ad hoc type system which I just presented can also be proved correct using a syntactic technique.

This technique is inspired by Abadi, Lampson and Lévy (1996) and by Pottier and Conchon (2000).

The idea

The idea is to reintroduce marked expressions:

$$e ::= \ldots \mid \ell : e$$

and to define a small-step operational semantics that keeps track of marks.

The semantics implements a sound *dynamic* dependency analysis.

The type system is a sound approximation of the semantics. Thus, it implements a sound *static* dependency analysis.

The operational semantics

The operational semantics has standard reduction rules that deal with functions, products, and sums, plus *ad hoc rules that deal with labels*:

$$(\ell : e_1)\, e_2 \longrightarrow \ell : (e_1\, e_2) \quad \text{(lift-app)}$$

$$\textbf{proj}_i\, (\ell : e) \longrightarrow \ell : (\textbf{proj}_i\, e) \quad \text{(lift-proj)}$$

$$(\ell : e)\; \textbf{case}\; x \succ e_1\, e_2 \longrightarrow \ell : (e\; \textbf{case}\; x \succ e_1\, e_2) \quad \text{(lift-case)}$$

These rules prevent labels from getting in the way, and *track dependencies*.

When labels are erased, these rules have no effect. So, the nonstandard semantics is faithful to a standard one, modulo erasure.

Computing with partial information

Let a *prefix* e be an expression that contains holes. Write $e \preceq e'$ if e' is obtained from e by replacing some holes with prefixes.

Reduction is extended to prefixes: holes block reduction.

Lemma (Monotonicity)
Let e, e' be prefixes such that $e \preceq e'$. If f is an expression such that $e \longrightarrow^* f$, then $e' \longrightarrow^* f$.

The operational semantics is sound

For an arbitrary set of security levels L, define $\lfloor e \rfloor_L$ as the prefix of e obtained by pruning every subexpression of the form $\ell : e$ where $\ell \notin L$.

Lemma (Stability)
Let e be a prefix and f an expression. If $e \longrightarrow^* f$ and $\lfloor f \rfloor_L = f$, then $\lfloor e \rfloor_L \longrightarrow^* f$.

Expressions that carry a label *not* found in f do not contribute to the computation of f.

Extending the type system

Since I have reintroduced marked expressions, I must slightly extend the type system.

Mark

$$\frac{\Gamma \vdash e : t \qquad \ell \lhd t}{\Gamma \vdash (\ell : e) : t}$$

This rule is reminiscent of DCC's. However, since the type constructor T_ℓ is gone, e and $\ell : e$ are given the *same* type t.

The premise $\ell \lhd t$ ensures that the type annotations carried by t are sufficiently high to reflect the presence of the mark ℓ.

The type system is sound

I now wish to prove that *reduction preserves types*.

The proof that the standard reduction rules preserve types is standard — well, not quite so, since types carry security annotations, but there is no surprise.

There remains to prove that the (*lift*) rules preserve types.

(lift-app) preserves types

Here is a type derivation for a (lift-app)-redex:

$$\dfrac{\dfrac{\Gamma \vdash e_1 : t \rightarrow t' \qquad \ell \lhd t \rightarrow t'}{\Gamma \vdash (\ell : e_1) : t \rightarrow t'}\,\text{Mark} \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash (\ell : e_1)\, e_2 : t'}\,\text{App}$$

One may transform it into:

$$\dfrac{\dfrac{\Gamma \vdash e_1 : t \rightarrow t' \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1\, e_2 : t'}\,\text{App} \qquad \ell \lhd t'}{\Gamma \vdash \ell : (e_1\, e_2) : t'}\,\text{Mark}$$

Type preservation

The cases of (lift-proj) and (lift-case) are left to the audience. Thus:

Lemma (Subject reduction)
$\Gamma \vdash e : t$ and $e \longrightarrow e'$ imply $\Gamma \vdash e' : t$.

Putting it all together

Together, the soundness of the semantics and that of the type system lead to noninterference.

Theorem (Noninterference)
$\vdash e : bool^\ell$ and $e \longrightarrow^\star v$ imply $\lfloor e \rfloor_{\downarrow\{\ell\}} \longrightarrow^\star v$.

Proof.
By subject reduction, v has type bool^ℓ. Thus, v must be of the form $\ell_1 : \ell_2 : \ldots : \ell_n : (\textbf{true} \mid \textbf{false})$, where $\ell_i \lhd \text{bool}^\ell$ holds for every $i \in \{1, \ldots, n\}$. This means $\ell_i \leq \ell$, that is, $\ell_i \in \downarrow\{\ell\}$, so $\lfloor v \rfloor_{\downarrow\{\ell\}}$ is v. The result follows by stability. □

A reformulation

This result is perhaps better known under a symmetric form:

Theorem (Noninterference)
Let $\vdash e_1 : bool^\ell$ and $\vdash e_2 : bool^\ell$ and $\lfloor e_1 \rfloor_{\downarrow\{\ell\}} = \lfloor e_2 \rfloor_{\downarrow\{\ell\}}$. Then, $e_1 \longrightarrow^\star v$ is equivalent to $e_2 \longrightarrow^\star v$.

Proof.
By the previous theorem and by monotonicity. □

Expressions that have a *low-level* type and that only differ in *high-level* components have the same behavior.

Part II

Flow Caml

Type system

Semantics

Noninterference

Type system

Semantics

Noninterference

Syntax

I refer to the programming language as ML. It has functions, products, sums, references, and exceptions.

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

Syntax

I refer to the programming language as ML. It has functions, products, sums, references, and exceptions.

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

Type system Semantics Noninterference

Syntax

I refer to the programming language as ML. It has functions, products, sums, references, and exceptions.

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

François Pottier Type-Based Information Flow Analyses

Type system Semantics Noninterference

Syntax

I refer to the programming language as ML. It has functions, products, sums, references, and exceptions.

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

François Pottier Type-Based Information Flow Analyses

Type system Semantics Noninterference

Syntax

I refer to the programming language as ML. It has functions, products, sums, references, and exceptions.

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

Exceptions are *second-class*. They are not values. the idioms "e **handle** $x \succ e$" and "**raise** x" are not available.

Syntax

I refer to the programming language as ML. It has functions, products, sums, references, and exceptions.

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

For the sake of simplicity, certain expression forms must be built out of *values*. However, this is not a deep restriction.

Syntax

I refer to the programming language as ML. It has functions, products, sums, references, and exceptions.

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

As usual in ML, polymorphism is introduced by **let** and restricted to *values*. Sequencing is expressed using **bind**.

Types

Types and *rows* are defined as follows:

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

The metavariables ℓ and pc range over $\mathcal{L}$.

Subtyping is structural and extends the security lattice.

$$
\mathrm{int}^\oplus \qquad (\ominus \xrightarrow{\ominus\ [\oplus]} \oplus)^\oplus \qquad \odot\ \mathrm{ref}^\oplus \qquad \oplus \times \oplus \qquad (\oplus + \oplus)^\oplus
$$

$$
\{\varepsilon \mapsto \oplus\}_{\varepsilon \in \mathcal{E}}
$$

Types, continued

In this definition, there are *no* (level, type, or row) variables.

This does not prohibit polymorphism. Although the type system does not have a ∀ quantifier, it has *infinitary intersection types*, introduced by **let**.

Furthermore, rows are infinite objects.

These design choices make it easier to prove noninterference.

A system that has (level, type, and row) variables, finite syntax for rows, and constraints, and that supports *type inference*, can be defined in *a second step*.

Protected types

The definition of $\ell \lhd t$ is unsurprising:

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

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

Typing judgements

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

$$\Gamma, M \vdash v : t \qquad\qquad pc, \Gamma, M \vdash e : t\ [r]$$

The level pc reflects how much information is associated with the knowledge that e is executed.

The row r reflects how much information is gained by observing the exceptions raised by e.

Connecting values and expressions

Values and expressions are connected as follows:

e-Value

$$\frac{\Gamma, M \vdash v : t}{*, \Gamma, M \vdash v : t\ [*]}$$

A value is an expression that has *no side effects*.

Abstraction and application

Abstraction delays effects. Application forces them ($pc \leq pc'$).

v-Abs

$$\frac{pc, \Gamma[x \mapsto t'], M \vdash e : t\ [r]}{\Gamma, M \vdash \lambda x.e : (t' \xrightarrow{pc\ [r]} t)^*}$$

e-App

$$\frac{\Gamma, M \vdash v_1 : (t' \xrightarrow{pc'\ [r]} t)^\ell \qquad \Gamma, M \vdash v_2 : t' \qquad pc \leq pc' \qquad \ell \leq pc' \qquad \ell \lhd t}{pc, \Gamma, M \vdash v_1\, v_2 : t\ [r]}$$

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, causing an indirect flow ($pc \lhd t$).

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

In the presence of first-class references, information about the reference's identity may leak as well ($\ell \lhd t$).

Raising an exception

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

e-Raise
$$\frac{\Gamma, M \vdash v : typexn(\varepsilon)}{pc, \Gamma, M \vdash \mathbf{raise}\ \varepsilon\, v : * \ [\varepsilon : pc; *]}$$

Raising an exception reveals that this program point was reached. Hence, the information gained by observing the exception is pc.

Handling a specific exception

Knowing that e_2 is executed allows deducing that an exception was caught. Thus, e_2 is typechecked under the stricter context $pc \sqcup pc_\varepsilon$, where pc_ε is the amount of information carried by the exception.

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

Examining the whole expression's result may also reveal that an exception was caught ($pc_\varepsilon \lhd t$).

Computing in sequence

Knowing that e_2 is executed allows deducing that e_1 did not raise any exception. The amount of information associated with this fact is bounded by $\sqcup r_1$.

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

Finally

Executing e_1 **finally** e_2 eventually leads to executing e_2, so observing that e_2 is executed yields no information. Thus, e_2 is typechecked under the context *pc*.

e-Finally

$$\frac{pc, \Gamma, M \vdash e_1 : t\ [r] \qquad pc, \Gamma, M \vdash e_2 : * \ [\partial\bot]}{pc, \Gamma, M \vdash e_1 \textbf{ finally } e_2 : t\ [r]}$$

Observing an exception originally raised by e_1 reveals that e_2 has completed successfully. To avoid keeping track of this fact, I require e_2 to always complete successfully.

Type system

Semantics

Noninterference

Reminder: a semantics with labels

In a labelled semantics, examining a *single* reduction sequence allows comparing it with other sequences. For instance, consider:

$$(\lambda xy.y)\,(\mathbf{H} : 27) \longrightarrow^* \lambda xy.y$$

By stability, this implies

$$(\lambda xy.y)\,[] \longrightarrow^* \lambda xy.y$$

By monotonicity, this implies

$$(\lambda xy.y)\,(\mathbf{H} : 68) \longrightarrow^* \lambda xy.y$$

Labels are limited

The statement

$$\textbf{if } !x = 0 \textbf{ then } z := 1$$

causes information to flow from x to z, *even when it is skipped*.

As a result, designing a sensible labelled operational semantics (one that enjoys *stability*) becomes problematic.

In fact, Denning (1982) claims that no dynamic dependency analysis is possible in the presence of mutable state.

A semantics with brackets

Instead, I will reason directly about *two* reduction sequences that *share* some structure.

I will design an ad hoc semantics where the following reduction sequence is valid:

$$(\lambda xy.y)\,\langle 27 \mid 68 \rangle \longrightarrow^{*} \lambda y.y$$

and where, by *projection*, one may deduce

$$(\lambda xy.y)\,27 \longrightarrow^{*} \lambda y.y$$
$$(\lambda xy.y)\,68 \longrightarrow^{*} \lambda y.y$$

Brackets encode the *differences* between two programs, that is, their high-level parts.

Why are brackets really useful?

In ML, references are dynamically allocated and do not have statically known names (they are not global variables).

One cannot tell in advance whether the references allocated at a certain site are high- or low-level. In fact, they might be both, depending on the calling context.

For these reasons, it is difficult to even *state* that the low-level slice of the store is the same in two executions of a program.

In the bracket semantics, the low-level slice of the store is *syntactically shared* between the two executions.

The bracket calculus ML^2

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

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

Brackets cannot not be nested.

Projections

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 that it encodes. In particular:

$$\lfloor \langle e_1 \mid e_2 \rangle \rfloor_i = e_i \qquad i \in \{1, 2\}$$

Functions

Each language construct is dealt with by *two* reduction rules. One performs computation. The other lifts brackets so that they never prevent computation.

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

Compare with the labelled semantics:

$$
(\mathbf{H} : e_1)\, e_2 \longrightarrow \mathbf{H} : (e_1\, e_2) \qquad \text{(lift-app)}
$$

Products

The treatment of products is analogous.

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

Designing the (lift) rules

The hypothetical reduction rule

$$
e \longrightarrow \langle \lfloor e \rfloor_1 \mid \lfloor e \rfloor_2 \rangle
$$

is computationally correct. However, in the presence of such a rule, achieving subject reduction would require the type system to view *every* expression as high-level.

The (lift) reduction rules track dependencies and must be made sufficiently precise to achieve subject reduction.

References

A *store* μ is a partial map from memory locations to values that *may contain brackets*.

Store bindings of the form $m \mapsto \langle v \mid \mathbf{void} \rangle$ or $m \mapsto \langle \mathbf{void} \mid v \rangle$ account for situations where the two programs that are being executed have different dynamic allocation patterns.

References, continued

Reductions which take place inside a $\langle \cdot \mid \cdot \rangle$ construct must read or write 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 $e \,/\, \mu$ for $e \,/_\bullet\, \mu$.

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

References, continued

The reduction rules that govern assignment are:

$$m := v \,/_i\, \mu \;\to\; () \,/_i\, \mu[m \mapsto \text{update}_i\, \mu(m)\, v] \qquad \text{(assign)}$$

$$\langle v_1 \mid v_2 \rangle := v \,/\, \mu \;\to\; \langle v_1 := \lfloor v \rfloor_1 \mid v_2 := \lfloor v \rfloor_2 \rangle \,/\, \mu \qquad \text{(lift-assign)}$$

where

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

Example

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

Exceptions at a glance

The semantics of exceptions is given by a number of standard rules and a single (lift) rule.

$$\begin{array}{rcl}
\textbf{bind } x = v \textbf{ in } e & \to & e[v/x] \\
\textbf{raise } \varepsilon\, v \textbf{ handle } \varepsilon\, x \succ e & \to & e[v/x] \\
\textbf{raise } \varepsilon\, v \textbf{ handle } e \textbf{ done} & \to & e \\
\textbf{raise } \varepsilon\, v \textbf{ handle } e \textbf{ propagate} & \to & e;\ \textbf{raise } \varepsilon\, v \\
a \textbf{ finally } e & \to & e;\ a \\
E[a] & \to & a \\
 & & \textit{if } E \textit{ handles neither } \lfloor a \rfloor_1 \textit{ nor } \lfloor a \rfloor_2 \\
E[\langle a_1 \mid a_2 \rangle] & \to & \langle \lfloor E \rfloor_1[a_1] \mid \lfloor E \rfloor_2[a_2] \rangle \\
 & & \textit{if none of the above rules applies}
\end{array}$$

Relating ML^2 to ML

Pairs of ML reduction sequences *that produce answers* are in one-to-one correspondence with ML^2 reduction sequences.

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

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

The basic idea

The bracket calculus is a tool to attack the noninterference proof. Indeed, to prove that *two* ML programs produce the same answer, it is sufficient to prove that a *single* ML^2 program produces an answer that contains *no brackets*.

Thus, the key is to keep track of brackets during reduction.

I do so via a standard technique: a type system for ML^2 and a *subject reduction* theorem.

Keeping track of brackets

To define a type system for ML^2, it suffices to give typing rules for brackets.

These rules are parameterized by an (upward-closed) set of "high" levels H. They require the value and the side effects of every bracket to be "high."

v-Bracket

$$\frac{\Gamma, M \vdash v_1 : t \qquad \Gamma, M \vdash v_2 : t \qquad H \lhd t}{\Gamma, M \vdash \langle v_1 \mid v_2 \rangle : t}$$

Type preservation

In ML^2, reduction preserves types:

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

A final lemma

An expression with a "low" type cannot produce a value whose projections differ.

Lemma
Let $\ell \notin H$. If $\vdash e : \text{int}^\ell$ and $e \longrightarrow^* v$ then $\lfloor v \rfloor_1 = \lfloor v \rfloor_2$.

Proof.
By subject reduction, $\vdash v : \text{int}^\ell$ holds. So, v must be either an integer constant k or a bracket $\langle k_1 \mid k_2 \rangle$. Because $\ell \notin H$, the latter is impossible. □

Noninterference

Theorem (Noninterference)
Choose $\ell, h \in \mathcal{L}$ such that $h \not\leq \ell$. Let $h \lhd t$. Assume $(x \mapsto t) \vdash e : \text{int}^\ell$, where e is an ML expression. If, for every $i \in \{1, 2\}$, $\vdash v_i : t$ and $e[v_i/x] \longrightarrow^* v'_i$ hold, then $v'_1 = v'_2$.

Proof.
Let $H = \uparrow\{h\}$. Define $v = \langle v_1 \mid v_2 \rangle$. $h \lhd t$ and v-Bracket imply $\vdash v : t$. By substitution, this yields $\vdash e[v/x] : \text{int}^\ell$.

Now, $\lfloor e[v/x] \rfloor_i$ is $e[v_i/x]$, which, by hypothesis, reduces to v'_i. By *completeness*, there exists an answer a such that $e[v/x] \longrightarrow^* a$. Then, by *soundness*, we have $\lfloor a \rfloor_i = v'_i$, so a is a value.

$h \not\leq \ell$ implies $\ell \notin H$. The previous lemma then shows that the projections of a coincide. □

Part III

Conclusion

Some open problems

- The type systems that I have presented are sometimes not flexible enough. *Dynamic labels* are an interesting extension. What *other extensions* are possible and useful?
- Noninterference is often too drastic a requirement. *Declassification* appears useful but is unsafe. How can it be tamed?
- Despite a huge number of publications, nobody seems to be using these type systems in practice. There may be a need for a few *killer applications*!

Selected References I

- Martín Abadi, Anindya Banerjee, Nevin Heintze, Jon G. Riecke.
  A Core Calculus of Dependency.
  POPL, 1999.
- Dorothy E. Denning.
  Cryptography and Data Security.
  Addison-Wesley, 1982.
- Andrew C. Myers.
  Mostly-Static Decentralized Information Flow Control.
  Technical Report MIT/LCS/TR-783, 1999.
- Andrew C. Myers and Andrei Sabelfeld.
  Language-Based Information-Flow Security.
  IEEE JSAC 21(1), 2003.

Selected References II

- François Pottier and Vincent Simonet.
  Information Flow Inference for ML.
  ACM TOPLAS 25(1), 2003.
- Vincent Simonet.
  The Flow Caml system: documentation and user's manual.
  INRIA Technical Report 0282.
- Steve Zdancewic and Andrew C. Myers.
  Secure Information Flow via Linear Continuations.
  HOSC 15(2–3), 2002.