Information Flow Inference For Free

François Pottier* Francois.Pottier@inria.fr

Sylvain Conchon* Sylvain.Conchon@inria.fr

Abstract

This paper shows how to systematically extend an arbitrary type system with dependency information, and how soundness and non-interference proofs for the new system may rely upon, rather than duplicate, the soundness proof of the original system. This allows enriching virtually any of the type systems known today with information flow analysis, while requiring only a minimal proof effort.

Our approach is based on an untyped operational semantics for a labelled calculus akin to core ML. Thus, it is simple, and should be applicable to other computing paradigms, such as object or process calculi.

The paper also discusses access control, and shows it may be viewed as entirely independent of information flow control. Letting the two mechanisms coexist, without interacting, yields a simple and expressive type system, which allows, in particular, "selective" declassification.

1 Introduction

Today, considerable amounts of military, commercial, or personal data are processed and stored in computer systems. Thus, valuable data must be protected against deliberate or accidental release or corruption, which may be caused not only by *individuals*, but also by *programs*. *Access control* mechanisms provide some protection, but require the programs to which access is granted to be unconditionally *trusted*. Allowing inspection or update of the data by *untrusted* programs requires analyzing their code, to ensure that it meets some security policy. This process is called *information flow* analysis.

The need for such a form of protection was identified very early [13]. Following military practice, several authors suggested assigning a *security level* to every program variable, and requiring that information be allowed to flow only from lower level variables to higher ones. The techniques proposed to enforce this restriction involved security checks at run-time [8], at compile-time [7], or as part of a manual proof process [4]. Security levels offer a simple way of guaranteeing *non-interference* – a property which allows describing many security policies, including (combinations of) secrecy and integrity requirements [11].

Non-interference states the absence of dependency between (part of) a program's inputs and (part of) its outputs. Thus, information flow analysis is nothing but a *dependency* analysis. This fact was pointed out by Abadi *et al.* [1], who described a general-purpose dependency analysis in terms of a *type system* for an extended λ-calculus.

In our eyes, expressing dependency properties in terms of *types* is a highly commendable approach. Indeed, it has no run-time cost, and offers a correctness guarantee prior to program execution. Types are usually simple, predictable, and may serve as a specification language. Lastly, this allows automating the construction of correctness proofs, provided type inference is available[1]. In fact, later versions of Denning's compile-time certification system [6] bear a strong resemblance with today's polymorphic, constraint-based type systems. Yet, to the best of our knowledge, the first type-based information flow analysis is due to Palsberg and Ørbæk, who developed a typed λ-calculus with integrity annotations [18, 17]. This work was, unfortunately, not supported by a non-interference proof. It was followed by several others, concerned with preserving secrecy in a first-order imperative language [27], a higher-order functional language [12, 1], or in Java [14].

Although these works offer a wide variety of techniques and ideas, none of them provides polymorphism and type inference – features routinely found in modern typed programming languages – *together* with a non-interference proof. For instance, Myers [14] offers a very powerful system, but is unable to prove its correctness, due to the sheer number of its features. Heintze, Riecke, Abadi and Banerjee [12, 1] only propose simply-typed λ-calculi. One may conjecture that their non-interference proofs, based on logical relations and (in the later paper) on a categorical semantics of λ-calculus, cannot easily deal with recursive or polymorphic types. Volpano and Smith's type inference algorithm [27] infers principal type schemes, but does *not* use them to its own advantage. Rather, it textually expands "let" definitions, which takes exponential time. The reason for this weakness may be the effort involved in duplicating a correctness proof for a truly polymorphic, constraint-based type system.

This paper describes a *systematic* way of extending an existing type system with information flow control. Given an arbitrary type system for core ML, we build a related system, whose types carry security annotations, and whose correctness, including non-interference, rests upon the origi-

*INRIA, BP 105, F-78153 Le Chesnay Cedex, France.

To be presented at the International Conference on Functional Programming, Montréal, Canada, September 2000.

[1] These are, of course, only well-known advantages of strong typing.

nal's. Thus, we avoid proof duplication, and obtain a formal non-interference result at a very small cost.

Varying the original system yields a whole *family* of provably correct, information flow-aware type systems, showing that information flow analysis may be readily combined with recursive types, polymorphism, type inference, or other advanced type-theoretic features. Our proofs rely on an untyped operational semantics. This approach has several advantages. It requires no domain- or category-theoretic knowledge. It does not make any assumptions about the form of types, allowing any type system to be used as a starting point. Furthermore, it should be applicable to other languages, such as λ-calculi with side effects, object calculi, or process calculi, which already enjoy rich type systems (e.g. [9]), yet may not have a denotational semantics.

We conclude with a comparison of information flow control and access control, and argue that these mechanisms may usefully coexist, while remaining entirely *independent* at a theoretical level. On the whole, we believe this paper gives rise to the most powerful type systems equipped with static information flow and access control to date, with minimal theoretical overhead. We hope it will serve as a useful practical guide for programming language implementors.

2 Overview

It seems intuitively obvious that the simplest way of dynamically tracking information flow throughout a computation is to *mark* its inputs with *labels*, and to require every operation to *copy* the labels carried by its arguments to its result. Provided labels are properly copied, none can be dropped along the way; so, the final result must carry the labels of all inputs which have effectively been used. Reversing this statement, we see that the final result *cannot depend* on any input whose label it does not carry. This route is taken by Abadi, Lampson and Lévy [2], who develop a labelled λ-calculus in order to dynamically analyze dependencies. It exhibits two differences with ordinary λ-calculus. First, expressions may carry a label; second, a new reduction rule copies labels, forcing the result of every function application to retain the label attached to the function. We review this calculus in section 3, and prove that "labels cannot be dropped" by stating a *stability theorem*.

Abadi, Lampson and Lévy's labelled λ-calculus, extended with a "let" construct, is our starting point. However, we wish to analyze dependencies *statically*, rather than dynamically; in fact, we plan to do so by building a *type system* for the labelled calculus. To this end, it seems natural to pick some existing type system for core ML, and enrich it with information about labels. However, if we *modify* an existing system, then we must also modify its correctness proof – which essentially means *duplicating* it. (This approach is followed in all papers to date.) Instead, we wish to build on the system's correctness *theorem*, regardless of its *proof*. Thus, we cannot afford to modify the existing system; instead, we must use it as a building block – a *black box* – in the definition of the new system.

However, neither labelled expressions, nor the extra reduction rule are known to a type system for core ML. How to "explain" these features to it? Our answer is to devise a *translation* scheme. Using it, we turn a program written in the labelled calculus into a core ML program, which we can then submit to a traditional type system. Because our translation scheme is computationally meaningful, the type thus produced makes sense: it describes the behavior of the original program with respect to data *and* to labels.

The encoding is extremely simple. It maps every source expression to a pair, whose first component represents the expression's computational content, and whose second component contains (an approximation of) its label. Of course, the target calculus must have pairs, as well as a family of constants which represent labels. We define such a calculus in section 4. We then describe the translation scheme, and give a fundamental *simulation* lemma, in section 5.

In section 6, we assume given a type system for the target calculus. We define a small number of requirements about it, most notably a syntactic type soundness theorem. Then, we impose this type system on the *source* calculus, *through* our translation. We show that the system thus obtained is also sound, and enjoys a non-interference property.

In section 7, we note that our translation scheme is a bit naïve, and may cause an exponential increase in the size of programs. To remedy this, we define a variant of it, which only has linear overhead, and show that it is equivalent, as far as typing is concerned.

In section 8, we illustrate our theoretical construction with a concrete example. We pick an existing type inference system for core ML [20], show that it meets our requirements, and give a concrete description of the corresponding information flow-aware system. We illustrate its power by running it on a small, but typical, program.

Section 9 discusses *access control*, and argues that it is independent of information flow control. The two mechanisms may of course coexist; we show that this allows *selective* declassification à la Myers and Liskov [15].

Section 10 discusses our contribution and concludes.

3 Source calculus

3.1 Presentation

Our source calculus is exactly Abadi, Lampson and Lévy's labelled λ-calculus [2], extended with a "let" construct in the style of ML.

$$\begin{array}{rcll}
e & ::= & & \text{source terms} \\
& | & k & \text{integer constant } (k \in \mathbb{N}) \\
& | & x & \text{variable } (x \in \mathcal{V}) \\
& | & \lambda x.e & \text{abstraction } (x \in \mathcal{V}) \\
& | & (e\ e) & \text{application} \\
& | & \text{let } x = e \text{ in } e & \text{local definition } (x \in \mathcal{V}) \\
& | & l : e & \text{labelled term } (l \in \mathcal{L})
\end{array}$$

The set $\mathcal{V}$ of program variables is assumed to be denumerable. No assumptions are made, at this point, about the set of labels $\mathcal{L}$.

The operational semantics for the calculus follows. Rules (β), (*let*) and (*context*) are standard. In rule (*context*), C ranges over arbitrary contexts: we do not yet choose a particular evaluation strategy. Rule (*lift*) allows moving labels outwards as little as possible to permit β-reduction. In other words, (*lift*) moves the label of a function to its result, so as to record the fact that it has been used in producing it, and, consequently, that the result *depends* on it.

$$\begin{array}{rcll}
(\lambda x.e_1)\ e_2 & \rightarrow & e_1[e_2/x] & & (\beta) \\
\text{let } x = e_1 \text{ in } e_2 & \rightarrow & e_2[e_1/x] & & (let) \\
(l : e_1)\ e_2 & \rightarrow & l : (e_1\ e_2) & & (lift) \\
C[e_1] & \rightarrow & C[e_2] & \text{if } e_1 \rightarrow e_2 & (context)
\end{array}$$

From a practical point of view, labels may be inserted into the code as a way of supplying security information. Specific labels may be used, for instance, to indicate that the result of certain expressions must be kept secret (e.g. if it is deemed to contain confidential information), or must not be trusted (e.g. if it was read from a public input channel). Notice, however, that no fixed meaning is built into labels. Labels merely track dependencies; their "meaning" only exists in the user's mind. After we define a type system for this calculus, the user will be able to add *static* typing assertions, to programs, thereby defining a *security policy*, and giving "meaning" to labels. Because the policy is statically enforced, the semantics does not have "security violations".

Example To illustrate every step taken throughout the paper, we will use a very simple running example. Assuming L and H are labels, let $e = (\text{L} : (\lambda xy.y))\ (\text{H} : 27)$. Then,

$$\begin{array}{rcll} e & \rightarrow & \text{L} : ((\lambda xy.y)\ (\text{H} : 27)) & \text{by } (\textit{lift}) \\ & \rightarrow & \text{L} : (\lambda y.y) & \text{by } (\beta) \end{array}$$

Rule (*lift*) moves the label L up one node. This exposes the β-redex, allowing the function $\lambda xy.y$ to receive its argument. At the same time, this guarantees that the application's result is tagged L, thus recording its dependency on the sub-term $\text{L} : (\lambda xy.y)$. Notice how the sub-term $\text{H} : 27$, which is unused, is dropped during β-reduction, label included.

3.2 A stability theorem

Thanks to the presence of explicit labels, this calculus enjoys a simple, *constructive* stability theorem. Given a computation sequence $e \rightarrow^\star f$, it guarantees that f does not depend on any sub-term of e which carries a label not found in f. More precisely, it states that the *prefix* of e obtained by "pruning" all such sub-terms is still able to produce f, in spite of the missing information.

We begin with some standard definitions. A *prefix* is an expression which may have missing sub-expressions:

$$\begin{array}{rcll} e & ::= & & \text{prefixes} \\ & | & \_ & \text{hole} \\ & | & \ldots & \end{array}$$

A prefix e is a prefix of another prefix (or expression) e' if e matches e', except e may have more holes; we write $e \preceq e'$. For the purposes of reduction, we treat $\_$ like a free variable. Prefixes enjoy the following monotonicity property:

Lemma 3.1 *Let e, e' be prefixes such that $e \preceq e'$. If f is an expression such that $e \rightarrow^\star f$, then $e' \rightarrow^\star f$.*

Given an arbitrary set of labels $L \subseteq \mathcal{L}$, we define the function $\lfloor\cdot\rfloor_L$, which maps any prefix e to a prefix of itself:

$$\begin{array}{rcll} \lfloor l : e \rfloor_L & = & \_ & \text{when } l \notin L \\ \lfloor l : e \rfloor_L & = & l : \lfloor e \rfloor_L & \text{when } l \in L \\ \lfloor \_ \rfloor_L & = & \_ & \\ \lfloor k \rfloor_L & = & k & \\ \lfloor x \rfloor_L & = & x & \\ \lfloor \lambda x.e \rfloor_L & = & \lambda x.\lfloor e \rfloor_L & \\ \lfloor e_1\ e_2 \rfloor_L & = & (\lfloor e_1 \rfloor_L\ \lfloor e_2 \rfloor_L) & \\ \lfloor \text{let } x = e_1 \text{ in } e_2 \rfloor_L & = & \text{let } x = \lfloor e_1 \rfloor_L \text{ in } \lfloor e_2 \rfloor_L & \end{array}$$

Informally speaking, $\lfloor\cdot\rfloor_L$ removes every sub-term which carries a label not found in L.

We are now ready to state the stability theorem. We supply its proof, not because of its interest, but rather because we wish to insist on its simplicity. It begins with the following auxiliary lemma.

Lemma 3.2 *If $e \rightarrow f$ may be derived by applying (β) or (let), or by (lift)-ing a label $l \in L$, then $\lfloor e \rfloor_L \rightarrow \lfloor f \rfloor_L$.*

Proof. Case (β). Then, $e = ((\lambda x.e_1)\ e_2) \rightarrow e_1[e_2/x] = f$. So, $\lfloor e \rfloor_L = ((\lambda x.\lfloor e_1 \rfloor_L)\ \lfloor e_2 \rfloor_L) \rightarrow \lfloor e_1 \rfloor_L[\lfloor e_2 \rfloor_L/x] = \lfloor e_1[e_2/x] \rfloor_L = \lfloor f \rfloor_L$. Case (*let*) is similar.

Case (*lift*). Then, $e = ((l : e_1)\ e_2) \rightarrow l : (e_1\ e_2) = f$. Because $l \in L$, $\lfloor e \rfloor_L$ is $((l : \lfloor e_1 \rfloor_L)\ \lfloor e_2 \rfloor_L$. By (*lift*), this reduces to $l : (\lfloor e_1 \rfloor_L\ \lfloor e_2 \rfloor_L) = \lfloor f \rfloor_L$. ☐

Theorem 3.1 (Stability) *Assume e is a prefix and f is an expression. If $e \rightarrow^\star f$ and $\lfloor f \rfloor_L = f$, then $\lfloor e \rfloor_L \rightarrow^\star f$.*

Proof. By induction over the length of the derivation of $e \rightarrow^\star f$. In the base case, e equals f, and the result is immediate. In the inductive case, we assume $e \rightarrow g \rightarrow^\star f$, where, by induction hypothesis, $\lfloor g \rfloor_L \rightarrow^\star f$. Let C be a context such that $e = C[e']$, $g = C[g']$, and $e' \rightarrow g'$, where $e' \rightarrow g'$ follows directly from (β), (*let*) or (*lift*). Define $D = \lfloor C \rfloor_L$. (Extend $\lfloor\cdot\rfloor_L$ to contexts in the obvious way.) Now, either lemma 3.2 is applicable to $e' \rightarrow g'$, or it isn't.

If it is, then $\lfloor e' \rfloor_L \rightarrow \lfloor g' \rfloor_L$ holds. So, $\lfloor e \rfloor_L = \lfloor C[e'] \rfloor_L = D[\lfloor e' \rfloor_L] \rightarrow^\star D[\lfloor g' \rfloor_L] = \lfloor C[g'] \rfloor_L = \lfloor g \rfloor_L$. Recall that $\lfloor g \rfloor_L \rightarrow^\star f$; the result follows.

If it isn't, then $e' \rightarrow g'$ is an instance of a (*lift*) rule involving a label $l \notin L$. So, g' is of the form $l : g''$. Since $g = C[l : g'']$ and $l \notin L$, $\lfloor g \rfloor_L$ equals $D[\_]$. Thus, it is a prefix of $D[\lfloor e' \rfloor_L] = \lfloor C[e'] \rfloor_L = \lfloor e \rfloor_L$. Recall that $\lfloor g \rfloor_L \rightarrow^\star f$; by lemma 3.1, this entails $\lfloor e \rfloor_L \rightarrow^\star f$. ☐

Example The program $e = (\text{L} : (\lambda xy.y))\ (\text{H} : 27)$ produces the result $\text{L} : (\lambda y.y) = f$. The only label that appears in f is L, so $\lfloor f \rfloor_{\{\text{L}\}} = f$. According to theorem 3.1, the prefix $\lfloor e \rfloor_{\{\text{L}\}}$ must reduce to f as well. Indeed, we find

$$\begin{array}{rcll} \lfloor e \rfloor_{\{\text{L}\}} & = & (\text{L} : (\lambda xy.y))\ \_ & \\ & \rightarrow & \text{L} : ((\lambda xy.y)\ \_) & \text{by } (\textit{lift}) \\ & \rightarrow & \text{L} : (\lambda y.y) & \text{by } (\beta) \end{array}$$

By lemma 3.1, any expression which has $\lfloor e \rfloor_{\{\text{L}\}}$ as a prefix must reduce to f as well. For instance, $(\text{L} : (\lambda xy.y))\ (\text{H} : 68)$ does. Using labels, we have determined that the sub-term 27 does not contribute to the computation $e \rightarrow^\star f$.

4 Target calculus

As explained in section 2, we now wish to translate our source calculus into a more conventional, unlabelled calculus, so as to be able to use some off-the-shelf type system. As our target calculus, we choose core ML, extended with pairs and label constants.

$$\begin{array}{rcll} e & ::= & & \text{target terms} \\ & | & k & \text{integer constant } (k \in \mathbb{N}) \\ & | & x & \text{variable } (x \in \mathcal{V}) \\ & | & \lambda x.e & \text{abstraction } (x \in \mathcal{V}) \\ & | & (e\ e) & \text{application} \\ & | & \text{let } x = e \text{ in } e & \text{local definition } (x \in \mathcal{V}) \\ & & & \\ & | & \langle e, e \rangle & \text{pair} \\ & | & \text{fst} & \text{first pair projection} \\ & | & \text{snd} & \text{second pair projection} \\ & & & \\ & | & l & \text{label constant } (l \in \mathcal{L}) \\ & | & @ & \text{label join} \end{array}$$

$$
\begin{array}{rcl}
(\!| e |\!) & = & \langle (\!| e |\!)^1, (\!| e |\!)^2 \rangle \\[1em]
(\!| k |\!)^1 & = & k \\
(\!| x |\!)^1 & = & \mathrm{fst}\ x \\
(\!| \lambda x.e |\!)^1 & = & \lambda x.(\!| e |\!) \\
(\!| e_1\ e_2 |\!)^1 & = & \mathrm{fst}\ ((\!| e_1 |\!)^1\ (\!| e_2 |\!)) \\
(\!| \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 |\!)^1 & = & \mathrm{let}\ x = (\!| e_1 |\!)\ \mathrm{in}\ (\!| e_2 |\!)^1 \\
(\!| l : e |\!)^1 & = & (\!| e |\!)^1 \\[1em]
(\!| k |\!)^2 & = & \epsilon \\
(\!| x |\!)^2 & = & \mathrm{snd}\ x \\
(\!| \lambda x.e |\!)^2 & = & \epsilon \\
(\!| e_1\ e_2 |\!)^2 & = & (\!| e_1 |\!)^2 \mathbin{@} \mathrm{snd}\ ((\!| e_1 |\!)^1\ (\!| e_2 |\!)) \\
(\!| \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 |\!)^2 & = & \mathrm{let}\ x = (\!| e_1 |\!)\ \mathrm{in}\ (\!| e_2 |\!)^2 \\
(\!| l : e |\!)^2 & = & l \mathbin{@} (\!| e |\!)^2
\end{array}
$$

Figure 1: Translation

For clarity, double applications of @ will be written using infix notation.

At this point, we assume the set of labels $\mathcal{L}$ is an upper semi-lattice, whose least element, ordering relation, and least upper bound operation are denoted ϵ, $\preccurlyeq$ and $\curlyvee$, respectively. For $l \in \mathcal{L}$, $\downarrow l$ denotes the lower cone $\{m \in \mathcal{L}\ ;\ m \preccurlyeq l\}$.

The calculus is equipped with a standard operational semantics, augmented with a new rule, (*join*), which states that @ returns the least upper bound of its arguments.

$$
\begin{array}{rcll}
(\lambda x.e_1)\ e_2 & \to & e_1[e_2/x] & (\beta) \\
\mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 & \to & e_2[e_1/x] & (\mathit{let}) \\
\mathrm{fst}\ \langle e_1, e_2 \rangle & \to & e_1 & (\pi_1) \\
\mathrm{snd}\ \langle e_1, e_2 \rangle & \to & e_2 & (\pi_2) \\
l \mathbin{@} m & \to & l \curlyvee m & (\mathit{join}) \\
C[e_1] & \to & C[e_2] \quad \text{if } e_1 \to e_2 & (\mathit{context})
\end{array}
$$

In rule (*context*), C again ranges over arbitrary contexts.

In the source calculus, an expression may carry zero, one or more labels. However, our translation scheme (to be presented in section 5) associates exactly one label with every expression. The least label ϵ will also be used to represent the absence of any label; the join operation, @, will be used to compute a conservative approximation of two labels. In other words, the source expressions e and $\epsilon : e$ will be translated to equivalent target expressions; so will $l : m : e$ and $(l \curlyvee m) : e$. This lack of precision will not be a problem. In fact, even if the translation itself was not approximate, the subsequent typing stage would certainly be: there is no point, in practice, in keeping track of *lists* of labels.

Lastly, to allow formal reasoning about the fact that @ stands for an associative operation, with neutral element ϵ, we define an extended semantics, denoted $\to_@$, by adding two extra rules to those above:

$$
\begin{array}{rcll}
(e_1 \mathbin{@} e_2) \mathbin{@} e_3 & \to_@ & e_1 \mathbin{@} (e_2 \mathbin{@} e_3) & (\mathit{assoc}) \\
\epsilon \mathbin{@} e & \to_@ & e & (\mathit{neutral})
\end{array}
$$

In particular, $\to_@$ contains $\to$.

5 Translation

A translation from the source calculus to the target calculus is defined in figure 1. The translation function maps every source expression e to a pair $(\!| e |\!)$, whose first (resp. second) component is denoted $(\!| e |\!)^1$ (resp. $(\!| e |\!)^2$). The functions $(\!| \cdot |\!)$, $(\!| \cdot |\!)^1$ and $(\!| \cdot |\!)^2$ are defined using mutual induction.

A source expression e is mapped to a pair, whose first component represents e's "actual value" (i.e. its computational content), and whose second component represents e's label. For instance, the unlabeled integer k is translated to $\langle k, \epsilon \rangle$ – we use the least label ϵ to denote the absence of a label. λ-abstractions are handled similarly. A variable x is translated to $\langle \mathrm{fst}\ x, \mathrm{snd}\ x \rangle$. Although it would be possible to translate it as x, this is more homogeneous.

The translation of an application expression $(e_1\ \ e_2)$ makes the (*lift*)-ing process explicit. According to our convention, $(\!| e_1 |\!)$ is a pair of a function and a label. We extract the former, namely $(\!| e_1 |\!)^1$, and apply it to its argument as a whole, namely $(\!| e_2 |\!)$. This returns – again – a pair, whose first component – the computational content – we then keep unchanged, and whose second component – the label – we join with e_1's label, namely $(\!| e_1 |\!)^2$. Thus, the label attached, in the translation, with $(e_1\ e_2)$, includes (i.e. is greater than, according to $\preccurlyeq$) the one attached with e_1. Joining labels allows keeping track of a single label per expression, rather than a list thereof. This is simpler, while still precise enough for our purposes.

Because the expression $(\!| e_1 |\!)^1\ (\!| e_2 |\!)$ appears in $(\!| e_1\ e_2 |\!)^1$ *and* in $(\!| e_1\ e_2 |\!)^2$, the size of $(\!| e |\!)$ is exponential in the size of e. From a purely theoretical point of view, this is not a problem. We favor this formulation for its simplicity: thanks to it, stating and proving a simulation lemma is very easy. In practice, however, efficiency demands a linear encoding. We will define one, and prove that it is equivalent to this one, as far as typing is concerned, in section 7. The reader may wish to immediately have a look at its definition, given in figure 2.

Expressions of the form let $x = e_1$ in e_2 and $l : e$ are translated in a straightforward way (again, inducing exponential behavior). Notice how $(\!| l : e |\!)$ has the same computational content as $(\!| e |\!)$, but has a greater label, due to the join operation $l \mathbin{@} \cdot$.

It is a matter of pure routine to check the following lemma, whose proof we therefore omit.

Lemma 5.1 (Simulation) *If $e \to f$, then $(\!| e |\!) \to_@^\star (\!| f |\!)$.*

Example The translation of $\mathrm{H} : 27$ is $\langle 27, \mathrm{H} \mathbin{@} \epsilon \rangle$. The translation of $\mathrm{L} : (\lambda xy.y)$ is $\langle \lambda x.\langle \lambda y.\langle \mathrm{fst}\ y, \mathrm{snd}\ y \rangle, \epsilon \rangle, \mathrm{L} \mathbin{@} \epsilon \rangle$. Thus, the term $e = (\mathrm{L} : (\lambda xy.y))\ (\mathrm{H} : 27)$ is translated to $\langle \mathrm{fst}\ a, (\mathrm{L} \mathbin{@} \epsilon) \mathbin{@} \mathrm{snd}\ a \rangle$, where a stands for

$$(\lambda x.\langle \lambda y.\langle \mathrm{fst}\ y, \mathrm{snd}\ y \rangle, \epsilon \rangle)\ \langle 27, \mathrm{H} \mathbin{@} \epsilon \rangle$$

Through (β), (π_1), (π_2), (*assoc*) and (*neutral*), $(\!| e |\!)$ reduces to $\langle \lambda y.\langle \mathrm{fst}\ y, \mathrm{snd}\ y \rangle, \mathrm{L} \mathbin{@} \epsilon \rangle$, which is exactly $(\!| \mathrm{L} : (\lambda y.y) |\!) = (\!| f |\!)$, in accordance with lemma 5.1.

6 Typing

6.1 Fixing a strategy

So far, we have not committed to a particular evaluation strategy in the source or target language. We must now do so, mainly because it seems we cannot otherwise state a meaningful *progress* theorem – one of the two fundamental type soundness theorems [28]. Let us settle on call-by-name evaluation; we will discuss call-by-value when appropriate.

In the source language, let $\rightarrow_{\text{CBN}}$ be the reduction relation obtained by restricting rule (*context*) to the following subset of contexts. Furthermore, define *values*, a subset of expressions, as follows.

$$C ::= [] \mid (C\ e) \mid l : C$$
$$v ::= k \mid \lambda x.e \mid l : v$$

We proceed similarly with the target language:

$$C ::= [] \mid (C\ e) \mid \text{fst}\ C \mid \text{snd}\ C \mid (@\ C) \mid (@\ l\ C)$$
$$v ::= k \mid \lambda x.e \mid \langle e, e\rangle \mid \text{fst} \mid \text{snd} \mid l \mid @ \mid (@\ l)$$

It is interesting to notice that the second fundamental type soundness theorem, namely *subject reduction*, *can* be stated independently of the reduction strategy: it suffices to require that types be preserved along all reduction paths. We will in fact do so in the following.

6.2 Assumptions

From here on, we assume the target calculus is equipped with a type system. As explained in section 2, we view it as a "black box": that is, we make no assumptions about its definition. Rather, we simply regard it as a relation between *closed* target expressions and *types*, satisfying a small number of axioms. This frees us from caring about typing rules, environments, constraints, universal quantification, or other subtleties involved in the system's inner workings. Thus, we assume *typing judgements* are of the form $e : t$, where e is a closed target expression, and t belongs to some (unspecified) set of types $\mathcal{T}$.

We now present our assumptions about the type system, in the form of 6 axioms. The first one states that any (closed) sub-expression of a well-typed expression is well-typed. This axiom is satisfied by all systems defined in terms of structural typing rules.

Axiom 1 (Compositionality) *If e is a closed expression such that $C[e] : t$, then $e : u$ holds for some $u \in \mathcal{T}$.*

Our next two axioms constitute a syntactic type soundness hypothesis [28]. The subject reduction axiom refers to $\rightarrow_{@}$, whereas $\rightarrow_{\text{CBN}}$ would be expected, since we have chosen a call-by-name evaluation strategy. This strengthens it in two ways. First, replacing $\rightarrow_{\text{CBN}}$ with $\rightarrow$ requires types to be preserved by *all* reductions, rather than only by call-by-name reductions. Many common type systems, such as Hindley/Milner's, are unaware of the evaluation strategy, and satisfy this stronger axiom. Although working with the usual (weak) version of the axiom may be possible, this choice simplifies our proofs. Second, replacing $\rightarrow$ with $\rightarrow_{@}$ requires (*assoc*) and (*neutral*) to preserve types as well.

Axiom 2 (Subj. red.) *If $e : t$ and $e \rightarrow_{@} f$, then $f : t$.*

Axiom 3 (Progress) *If $e : t$, then either $\exists f \quad e \rightarrow_{\text{CBN}} f$, or e is a value.*

The next axiom requires that every label l, which is already a valid *expression* in the target calculus, be also a valid *type* (possibly modulo some implicit embedding). It also states that, if m is a valid type for l, where both l and m are labels, then l must be below m in the semi-lattice $\mathcal{L}$.

Clearly, one way of implementing these requirements is to define a set of types $\mathcal{T}$ which syntactically contains $\mathcal{L}$, to make $l : l$ a valid typing judgement for every $l \in \mathcal{L}$, and to define a *subtyping* relationship whereby l is a subtype of m if and only if $l \preccurlyeq m$ holds. We illustrate this approach in section 8. However, it is interesting to note that this axiom does not *demand* subtyping. A system without subtyping, but with a sufficient degree of polymorphism, may also be used. For instance, if $\mathcal{L}$ happens to be the power-set of a set $\mathcal{P}$ (which represents, say, principals), then labels may be typed using $\mathcal{P}$-indexed *rows* [21]. A similar idea underlies Objective ML [22], a typed object-oriented language which does not rely on subtyping.

Axiom 4 (Labels) *Every label is a type: $\mathcal{L} \subset \mathcal{T}$. If $l, m \in \mathcal{L}$, then $l : m$ implies $l \preccurlyeq m$.*

Our last two axioms concern integers and pairs. They are far less important than axioms 1–4: their main use is to help formulate the non-interference theorem in a nice way.

Axiom 5 (Integers) *There is a type* $\texttt{int} \in \mathcal{T}$. *A value v satisfies $v : \texttt{int}$ if and only if it is an integer constant $k \in \mathbb{N}$.*

Axiom 6 (Pairs) *There is a partial function $\times : \mathcal{T}^2 \rightarrow \mathcal{T}$ such that $\langle e, f\rangle : t \times u$ implies $e : t \wedge f : u$. Conversely, if e and f are well-typed, then $\langle e, f\rangle$ is well-typed.*

Note that `int` and $\times$ may not directly correspond to the type system's own `int` and $\times$ type constructors. Indeed, what is known as a *type* in this axiomatization may be known e.g. as a *type scheme* in the system's actual definition.

6.3 Typing the source calculus

We now define the type system of the source calculus as the composition of the translation defined in section 5 with the type system of the target calculus. That is, for any closed source expression e, $e : t$ holds if and only if $(\!| e |\!) : t$ holds. We notice that if the chosen type system enjoys the existence of principal types, or of a type inference algorithm, then so does the newly defined system.

This abstract definition suffices to prove soundness and non-interference theorems about the derived system. Of course, if one is given the rules which define the target system, then one may combine them with the definition of $(\!| \cdot |\!)$, yielding a set of *derived* rules which allow direct type checking/inference in the source calculus. We will illustrate this in section 8.

The new system enjoys the following two soundness results. We omit the proof of theorem 6.2, which is straightforward, but slightly verbose.

Theorem 6.1 (Subj. red.) *If $e : t$ and $e \rightarrow f$, then $f : t$.*

Proof. According to lemma 5.1, $e \rightarrow f$ implies $(\!| e |\!) \rightarrow^{\star}_{@} (\!| f |\!)$. Furthermore, by definition of the type system in the source calculus, our hypothesis $e : t$ may be read as $(\!| e |\!) : t$, and our goal may be read as $(\!| f |\!) : t$. The result follows from the fact that $\rightarrow_{@}$ preserves types (axiom 2). $\square$

Theorem 6.2 (Progress) *If $e : t$, then either $\exists f \quad e \rightarrow_{\text{CBN}} f$, or e is a value.*

Next, we prove a non-interference theorem, which states that types in the new system do contain useful dependency information. The interesting aspect of our proof is that it is written in an entirely operational style: it essentially relies on two properties of the labelled calculus: stability (theorem 3.1) and subject reduction (theorem 6.1).

For simplicity, the theorem only concerns integer results. A more general statement would be possible.

Theorem 6.3 (Non-interference) *If $e : \texttt{int} \times l$ and $e \to^\star v$, where v is a value, then $\lfloor e \rfloor_{\downarrow l} \to^\star v$.*

Proof. According to theorem 6.1, $v : \texttt{int} \times l$ holds. This may be read $(\!| v |\!) : \texttt{int} \times l$, which, according to axiom 6, implies $(\!| v |\!)^1 : \texttt{int}$. So, according to axiom 5, $(\!| v |\!)^1$ cannot be a λ-abstraction. Considering v is a value, v must be of the form $l_1 : l_2 : \ldots : l_n : k$, for some $n \geq 0$.

Thus, $(\!| v |\!)$ is $\langle k, l_1 @ \ldots @ l_n @ \epsilon \rangle$. From the fact that this expression has type $\texttt{int} \times l$, we may deduce, through axiom 6, that $l_1 @ \ldots @ l_n @ \epsilon$ has type l. However, this expression may be reduced, by repeated application of rule (*join*), to $l_1 \curlyvee \ldots \curlyvee l_n$. According to axioms 2 and 4, it follows that $l_1 \curlyvee \ldots \curlyvee l_n \preccurlyeq l$. In other words, every l_i is an element of $\downarrow l$. So, $\lfloor v \rfloor_{\downarrow l}$ equals v, which, according to theorem 3.1, implies $\lfloor e \rfloor_{\downarrow l} \to^\star v$. □

The non-interference theorem may be better known under the following symmetric form:

Corollary 6.4 *Assume $e, f : \texttt{int} \times l$ and $\lfloor e \rfloor_{\downarrow l} = \lfloor f \rfloor_{\downarrow l}$. Then, either both e and f diverge, or both e and f converge and produce the same value.*

Proof. Assume e converges. Then, according to theorem 6.3 and lemma 3.1, f converges to the same value. By symmetry, the converse also holds: if f converges, then e converges to the same value. Furthermore, by theorems 6.1 and 6.2, neither e nor f can go wrong. The result follows. □

Here, to converge means to be able to reach a value along some reduction path. To diverge means not to converge and not to go wrong. To go wrong means to get stuck along some reduction path. By normalization[2], these are the same notions, regardless of whether $\to$ or $\to_{\textsc{cbn}}$ is being used.

Corollary 6.4 guarantees not only that e and f produce the same value, but also that they behave similarly with respect to termination. This is a strong non-interference statement. With a call-by-value semantics, one would obtain a slightly weaker result, whereby e and f would be guaranteed to yield the same value *only* if they terminate. Indeed, if $e \to_{\textsc{cbv}} v$ and $f \to_{\textsc{cbv}} w$, where v and w are values, then, by normalization, $e \to_{\textsc{cbn}} v$ and $e \to_{\textsc{cbn}} w$, whence, by corollary 6.4, $v = w$. To obtain a strong non-interference result in a call-by-value setting, one may modify the labelled calculus accordingly [2, section 3.7], and repeat our construction. This yields, however, a significantly more restrictive type system.

These non-interference results are stated in the source calculus, which has non-standard semantics. However, labels are not first-class entities, i.e. they cannot affect the course of computations, as shown by [2, prop. 3]. Thus, if all labels are removed before execution, i.e. if we evaluate "stripped" terms within a standard λ-calculus, then the non-interference results still hold.

Lastly, it is important to prove that "enough" programs are accepted by the new system, which may otherwise turn out to be devoid of practical interest. Unfortunately, doing so at an abstract level requires more axioms, which are difficult to state in an elegant way. For this reason, we will only address this issue in the concrete setting of section 8.

[2]We haven't proved a normalization theorem for the labelled λ-calculus, but this can be done using existing techniques.

$$
\begin{array}{rcl}
[\![k]\!] & = & \langle k, \epsilon \rangle \\
[\![x]\!] & = & \langle \mathrm{fst}\, x, \mathrm{snd}\, x \rangle \\
[\![\lambda x.e]\!] & = & \langle \lambda x.[\![e]\!], \epsilon \rangle \\
[\![e_1\; e_2]\!] & = & \mathrm{letp}\ \langle x, t \rangle = [\![e_1]\!] \text{ in} \\
& & \mathrm{letp}\ \langle y, u \rangle = x\; [\![e_2]\!] \text{ in} \\
& & \langle y, t @ u \rangle \\
[\![\mathrm{let}\ x = e_1 \text{ in } e_2]\!] & = & \mathrm{let}\ x = [\![e_1]\!] \text{ in } [\![e_2]\!] \\
[\![l : e]\!] & = & \mathrm{letp}\ \langle x, t \rangle = [\![e]\!] \text{ in} \\
& & \langle x, l @ t \rangle
\end{array}
$$

Figure 2: Linear translation

7 A more efficient translation

The translation scheme presented in section 5 behaves nicely with respect to reduction: it enjoys a very simple simulation lemma. However, because it duplicates sub-expressions, it has exponential complexity. In this section, we present a slightly different translation scheme, which only has linear overhead, because it uses local variable definitions to share sub-expressions where needed. We could have chosen to use this one in the first place, but losing lemma 5.1 would have made our proofs somewhat heavier.

The new encoding's definition is shown in figure 2. The auxiliary local variables x, t, y, u must be chosen so as to avoid variable capture. We use letp $\langle x, y \rangle = e_1$ in e_2 to denote $((\lambda p.((\lambda xy.e_2)\ (\mathrm{fst}\ p)\ (\mathrm{snd}\ p)))\ e_1)$ where p does not appear free in e_2. It would also be possible to implement "letp" in terms of "let". Our choice emphasizes the fact that no polymorphism is required: the point is to avoid duplicating e_1, not to generalize its type.

To establish a relationship between $[\![\cdot]\!]$ and $(\!| \cdot |\!)$, we need a few extra axioms concerning the target type system. Let us use $e :_C t$ to denote $C[e] : t$. This gives us a crude way of denoting the "type" of a non-closed expression.

Axiom 7 *If $\langle \mathrm{fst}\, e, \mathrm{snd}\, e \rangle :_C t$, then $e :_C t$.*

Axiom 8 *If x and y appear free exactly once in f, and if $f[e_1/x, e_2/y] :_C t$, then* letp $\langle x, y \rangle = \langle e_1, e_2 \rangle$ in $f :_C t$.

Axiom 9 let $x = e$ in $\langle f_1, f_2 \rangle :_C t$ *holds if and only if* $\langle \mathrm{let}\ x = e \text{ in } f_1, \mathrm{let}\ x = e \text{ in } f_2 \rangle :_C t$ *does.*

Using these extra axioms, as well as axiom 2, it is easy to prove that $(\!| \cdot |\!)$ and $[\![\cdot]\!]$ give rise to the same derived type system:

Theorem 7.1 *$[\![e]\!] : t$ if and only if $(\!| e |\!) : t$.*

Thus, if type checking (resp. type inference) has complexity $O(n^k)$, where $k \geq 1$, in the original system, then it has the same complexity in the derived system. Indeed, $[\![\cdot]\!]$ may be computed in linear time, and the size of its output remains within a constant factor of that of its input.

8 A concrete case

In this section, we illustrate our approach with a concrete example. We first pick an existing type system for the target calculus, and show that it satisfies the axioms given in the previous section. Our construction thus gives rise to a concrete information flow-aware type system for the source

$$\perp \leq \underline{t} \qquad \underline{t} \leq \top \qquad \texttt{int} \leq \texttt{int} \qquad \frac{l \preccurlyeq l'}{l \leq l'}$$

$$\frac{\underline{t}_0' \leq \underline{t}_0 \quad \underline{t}_1 \leq \underline{t}_1'}{\underline{t}_0 \rightarrow \underline{t}_1 \leq \underline{t}_0' \rightarrow \underline{t}_1'} \qquad \frac{\underline{t}_0 \leq \underline{t}_0' \quad \underline{t}_1 \leq \underline{t}_1'}{\underline{t}_0 \times \underline{t}_1 \leq \underline{t}_0' \times \underline{t}_1'}$$

Figure 3: Subtyping on ground types

calculus, a direct description of which is given in the form of a set of type inference rules. Lastly, to illustrate the system's power, we exercise it on a simple program fragment.

8.1 Typing the target calculus

Our target calculus is simply core ML with pairs, extended with label constants and the primitive operation @. In practice, almost any type system for core ML will do, provided it allows giving an appropriate type scheme to every $l \in \mathcal{L}$ and to @. Thus, we have a very wide range of systems to choose from, e.g. [10, 24, 26, 22, 3, 16]. We pick a subtyping-constraint-based type system, previously studied by the first author. By lack of space, we must describe it very succinctly. More detailed accounts appear in [20, 19].

For simplicity, we distinguish identifiers bound by λ, denoted $x, y, \ldots$ from those bound by "let", denoted $X, Y, \ldots$ We expect each λ-identifier to be bound at most once in a given program. Furthermore, in every expression of the form let $X = e_1$ in e_2, we require X to appear free within e_2. Overcoming these restrictions is of course possible, but requires more cumbersome typing rules.

The presentation of the system begins with a definition of *ground types*. They are the *regular* trees described by the following *co-inductive* definition:

$$\underline{t} ::= \perp \mid l \mid \texttt{int} \mid \underline{t} \times \underline{t} \mid \underline{t} \rightarrow \underline{t} \mid \top$$

Ground types are equipped with a *subtyping* order, given in figure 3. It is, again, defined co-inductively: a subtyping assertion holds if and only if it has a finite or infinite derivation. Let us assume, from here on, that $(\mathcal{L}, \preccurlyeq)$ is a lattice. Then, ground types also form a lattice, within which $\mathcal{L}$ is embedded.

We then (inductively) define *types* and *constraints*:

$$\begin{aligned} t &::= \alpha, \beta, \ldots \mid \perp \mid l \mid \texttt{int} \mid t \times t \mid t \rightarrow t \mid \top \\ c &::= t \leq t \end{aligned}$$

Here, $\alpha, \beta, \ldots$ range over a denumerable set of *type variables*. A *ground substitution* ϕ is a map from type variables to ground types. ϕ *satisfies* a constraint $t_1 \leq t_2$ if and only if $\phi(t_1) \leq \phi(t_2)$. ϕ satisfies a constraint set C if and only if it satisfies each of its elements.

A *context* A is a set of bindings of the form $x : t$. A *type scheme* σ is a triple of a constraint set, a context and a type, written $\forall C.\, A \Rightarrow t$. Intuitively speaking, *all* variables which appear in σ should be thought of as universally quantified, hence the $\forall$ notation. More formally, the *denotation* of a type scheme is defined by

$$[\![\forall C.\, A \Rightarrow t]\!] = \uparrow\{\phi(A \Rightarrow t)\,;\, \phi \text{ satisfies } C\}$$

$$\frac{\alpha \text{ fresh}}{\Gamma \vdash_{\mathrm{I}} x : \langle x : \alpha\rangle \Rightarrow \alpha}$$

$$\frac{\Gamma \vdash_{\mathrm{I}} e : \forall C.\, A \Rightarrow t}{\Gamma \vdash_{\mathrm{I}} \lambda x.e : \forall C.\, (A \setminus x) \Rightarrow A(x) \rightarrow t}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{I}} e_1 : \forall C_1.\, A_1 \Rightarrow t_1 \qquad \Gamma \vdash_{\mathrm{I}} e_2 : \forall C_2.\, A_2 \Rightarrow t_2 \\ \alpha \text{ fresh} \qquad C = C_1 \cup C_2 \cup \{t_1 \leq t_2 \rightarrow \alpha\}\end{array}}{\Gamma \vdash_{\mathrm{I}} e_1\, e_2 : \forall C.\, (A_1 \sqcap A_2) \Rightarrow \alpha}$$

$$\frac{\Gamma(X) = \sigma \qquad \rho \text{ fresh renaming of } \sigma}{\Gamma \vdash_{\mathrm{I}} X : \rho(\sigma)}$$

$$\frac{\Gamma \vdash_{\mathrm{I}} e_1 : \sigma_1 \qquad \Gamma + [X \mapsto \sigma_1] \vdash_{\mathrm{I}} e_2 : \sigma_2}{\Gamma \vdash_{\mathrm{I}} \text{let } X = e_1 \text{ in } e_2 : \sigma_2}$$

Figure 4: Type inference rules

where $\uparrow X$ represents the upper cone of a set X with respect to ground subtyping. (This requires a straightforward extension of $\leq$ to objects of the form $\underline{A} \Rightarrow \underline{t}$.) Given two type schemes σ_1 and σ_2, we say the former is *more general than* the latter, and we write $\sigma_1 \leq^{\forall} \sigma_2$, if and only if $[\![\sigma_1]\!] \supseteq [\![\sigma_2]\!]$.

Figure 4 gives the type inference rules of the system. Judgements are of the form $\Gamma \vdash_{\mathrm{I}} e : \sigma$, where Γ is an *environment* (i.e. a list of bindings of the form $X : \sigma$), e is a target expression, and σ is a type scheme. As far as notation is concerned, $\langle x : \alpha\rangle$ represents a context containing a single entry. $A \setminus x$ is the context obtained by removing x's binding (if any) from A. We shorten the notation $\forall C.\, A \Rightarrow t$ to $\forall C.\, t$, $A \Rightarrow t$, or simply t, if A, C, or both are empty.

For the sake of readability, we slightly abuse notation. We let $A(x)$ stand for the type associated with x in A, if A contains a binding for x, and for $\top$ otherwise. We use $A_1 \sqcap A_2$ to denote the point-wise intersection of A_1 and A_2. That is, whenever x has a binding in A_1 or A_2, its binding in $A_1 \sqcap A_2$ is $A_1(x) \sqcap A_2(x)$. Because the system does not have intersection types, this expression must in fact be understood as a fresh type variable, accompanied by an appropriate conjunction of subtyping constraints.

Every type scheme is implicitly required to have a non-empty denotation, i.e. a solvable set of constraints.

These rules only describe core ML. Type inference for the full target language, as defined in section 4, is obtained by adding the following (pseudo-)bindings to the initial typing environment Γ_0:

$$\begin{array}{rll} k: & \texttt{int} & (k \in \mathbb{N}) \\ \text{fst}: & \alpha \times \top \rightarrow \alpha & \\ \text{snd}: & \top \times \alpha \rightarrow \alpha & \\ l: & l & (l \in \mathcal{L}) \\ @: & \forall\{\alpha \leq \omega\}.\, \alpha \rightarrow \alpha \rightarrow \alpha & \\ \langle\cdot,\cdot\rangle: & \alpha \rightarrow \beta \rightarrow \alpha \times \beta & \end{array}$$

Here, ω stands for the greatest element of the lattice $\mathcal{L}$. The constraint $\alpha \leq \omega$ guarantees that @ is only applied to expressions which denote label constants.

$$\Gamma \vdash_{\mathrm{D}} k : \mathtt{int}^{\epsilon}$$

$$\frac{\alpha, \varphi \text{ fresh}}{\Gamma \vdash_{\mathrm{D}} x : \langle x : \alpha^{\varphi} \rangle \Rightarrow \alpha^{\varphi}}$$

$$\frac{\Gamma \vdash_{\mathrm{D}} e : \forall C.\, A \Rightarrow t}{\Gamma \vdash_{\mathrm{D}} \lambda x.e : \forall C.\, (A \setminus x) \Rightarrow (A(x) \rightarrow t)^{\epsilon}}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{D}} e_1 : \forall C_1.\, A_1 \Rightarrow t_1 \qquad \Gamma \vdash_{\mathrm{D}} e_2 : \forall C_2.\, A_2 \Rightarrow t_2 \\ \alpha, \varphi \text{ fresh} \\ C = C_1 \cup C_2 \cup \{t_1 \leq (t_2 \rightarrow \alpha^{\varphi})^{\varphi},\, \varphi \leq \omega\}\end{array}}{\Gamma \vdash_{\mathrm{D}} e_1\, e_2 : \forall C.\, (A_1 \sqcap A_2) \Rightarrow \alpha^{\varphi}}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{D}} e : \forall C.\, A \Rightarrow t \\ \alpha, \varphi \text{ fresh} \qquad C' = C \cup \{t \leq \alpha^{\varphi},\, l \leq \varphi \leq \omega\}\end{array}}{\Gamma \vdash_{\mathrm{D}} (l : e) : \forall C'.\, A \Rightarrow \alpha^{\varphi}}$$

Figure 5: Derived type inference rules

8.2 Back to the source calculus

We are now ready to apply the results of section 6 in the concrete setting considered here.

Theorem 8.1 *Define* types*, in the sense of section 6, to be* type schemes*, in the sense of this section. Define* $e : \sigma$ *to hold if and only if* $\exists \sigma' \leq^{\forall} \sigma \quad \Gamma_0 \vdash_{\mathrm{I}} e : \sigma'$*. Then, axioms 1–9 are satisfied.*

A proof of subject reduction, in the case of core ML, appears in [20]. Extending it to the language considered here, as well as checking the other axioms, is straightforward, although somewhat lengthy. We omit proofs.

Composing $[\![\cdot]\!]$ with the type inference algorithm of the target calculus yields a type inference algorithm for the source calculus, enjoying all of the properties stated in section 6. Let us now give a more direct description of this algorithm. By systematically composing the definition of the encoding with the rules of figure 4, and performing a few constraint simplification steps, as allowed by the use of $\leq^{\forall}$ in theorem 8.1, we obtain a set of derived type inference rules, shown in figure 5. The last two rules, which deal with "let"-bound variables and "let" definitions, are unchanged, so they are not shown. The product notation $t \times u$ has been replaced with t^u, so as to insist on the fact that we are dealing with types carrying security annotations.

These rules seem quite intuitive. They resemble the rules of figure 4, with the following differences. Values (integer constants and λ-abstractions) are annotated with ϵ upon creation. The security level of a function application expression is the join of the result level with the function level, thus recording the fact that the function contributes to the computation. The security level of a labelled expression $l : e$ is the join of e's level with l. Of course, it would have been easy to come up with these rules directly. However, our approach has several advantages over a manual approach. First, it is *systematic*, leaving no doubt that these rules are natural. Second, we obtain correctness proofs (almost) *for free*, which is non-trivial, considering the system has polymorphism, subtyping, recursive types, and type inference. Lastly, our approach is *general* and may be applied to many other type systems.

As a last refinement, it would be possible to partition types, *a posteriori*, into three sorts: plain types t, label types u, and secure types, of the form t^u. This would allow getting rid of the constraint $\varphi \leq \omega$ in the last two rules: φ would then range over labels, making it redundant.

Comparing typability in the derived and in the original system is now easy. Let strip denote the natural projection from source to target language; in particular, $\mathrm{strip}(l : e)$ is $\mathrm{strip}(e)$. Then,

Theorem 8.2 (Conservativity) *The source expression e is well-typed in the derived system if and only if the target expression* $\mathrm{strip}(e)$ *is well-typed in the original system.*

The proof relies on two simple remarks. First, any solution of the constraints generated by the rules of figure 5 also satisfies those inferred by the rules of figure 4. Conversely, any solution of the latter may be extended to a solution of the former, where every label variable φ is mapped to ω.

This result shows that one may switch to the new type system, and label any number of sub-expressions in a program, without affecting its typability. A program may become untypable only if a non-trivial security policy, expressed by inserting typing assertions, is adopted.

Example Let us use the rules of figure 5 to infer the type of our running example. The type scheme inferred for $\mathrm{H} : 27$ clearly is $\mathtt{int}^{\mathrm{H}}$. The one inferred for $\mathrm{L} : (\lambda xy.y)$ is $(\top \rightarrow (\alpha^{\varphi} \rightarrow \alpha^{\varphi})^{\epsilon})^{\mathrm{L}}$. Thus, the term $e = (\mathrm{L} : (\lambda xy.y))\ (\mathrm{H} : 27)$ receives the type scheme $(\alpha^{\varphi} \rightarrow \alpha^{\varphi})^{\mathrm{L}}$.

This type scheme states that evaluating e does not reveal any information of level H. Thus, the type inference algorithm *statically* finds that e does not leak the value 27, a fact which we had previously *dynamically* obtained by evaluating e (see section 3). Furthermore, this type scheme is polymorphic in α *and* in φ, showing that e's result – which is $\mathrm{L} : \lambda y.y$, the identity function labelled L – is able to accept any argument, regardless of its content *and* of its security level.

8.3 A realistic example

We conclude this section with a longer example. We assume the source language is extended with operations on Booleans, strings, pairs, variants and records. By lack of space, we do not define typing rules for these constructs. Provably correct rules can be obtained in (at least) two ways. One is to explicitly extend the target language, the translation scheme, and our proofs. The other is to derive correct typing rules for these constructs by considering their Church encodings into the basic language.

Figure 6 shows a small example program. It is a full program, which contains no type information, but does contain a few security annotations, in the form of labelled expressions.

The program first defines a classic predicate on lists, *exists*, which tells whether a given predicate is satisfied by at least one element of a given list. Recursion is achieved via an explicit fix-point combinator, *fix*. It is well-typed, because the system has recursive types. Thus, we are able to write recursive programs, even though our formal development did not explicitly deal with recursion.

To improve readability, we write t instead of t^{ϵ} (resp. t^{ω}) when t occurs positively (resp. negatively) in a type scheme.

```
let fix ff =
  (fun f x → ff (f f) x) (fun f x → ff (f f) x)

let exists = fix (
  fun exists predicate list →
    match list with
      Nil →
        false
    | Cons (element, rest) →
        if predicate element then
          true
        else
          exists predicate rest
)

let users =
  Cons({ login = "Pam"; pw = Sys : "7nuggets" },
  Cons({ login = "Sam"; pw = Sys : "" },
  Nil))

let query1 =
  exists (fun r →
    r.login = Priv : "Monica"
  ) users

let query2 =
  exists (fun r →
    r.pw = ""
  ) users
```

Figure 6: Example program

Then, the type scheme computed by the type inference algorithm for *exists* is

$$\forall C. (\alpha^{\varphi} \rightarrow \mathtt{bool}^{\psi})^{\psi} \rightarrow \zeta \rightarrow \mathtt{bool}^{\psi}$$

where C contains a single constraint:

$$\zeta \leq [\,\mathtt{Nil} \mid \mathtt{Cons}\ of\ (\alpha^{\varphi} \times \zeta)^{\psi}\,]^{\psi}$$

Intuitively, this recursive constraint requires ζ to represent a list, whose elements have type α^{φ}, and whose security level is ψ. *exists*'s first argument, a predicate, must accordingly accept an argument of type α^{φ}. If the predicate has level ψ, and if it returns a Boolean result of level ψ, then so will *exists*. Notice that φ are ψ are *a priori* unrelated: they will become related only if *exists* is applied to a predicate which leaks some information about its argument.

Three important points must be made here. First, this type scheme is precise, and highly polymorphic. Thus, multiple applications of *exists*, e.g. to lists with distinct security levels, or to predicates with different behavior, will not "pollute" each other. This is a requirement when writing libraries, since code duplication would otherwise be necessary. Second, the code of *exists* contains no security annotations, and its type was inferred without help from the user. This feature is also of utmost importance for *backward compatibility*: it allows a large body of code, written without any security requirements in mind, to be re-used in a program where security matters. Third, this type scheme is independent of the underlying security lattice. Even though it does not mention any constant label $l \in \mathcal{L}$, it does encode relevant dependency information. In other words, the choice of a particular security lattice is irrelevant when analyzing generic code; it is required only when wishing to enforce a particular security policy.

Let us come back to the program in figure 6. Its next step is to define a list, called *users*, whose elements are records containing name (*login*) and password (*pw*) strings. The information contained in *pw* fields, which is deemed somehow important, is labelled *Sys*. Notice that labelling a piece of data does not *restrict* access to it; it only forces any computations which make use of this data to receive a type which *reveals* this dependency. In other words, our type system does not *forbid* anything by default; it merely *watches* everything. Security restrictions, when required, may be added using additional type constraints, as we will see below.

The rest of the program consists of two queries about the *users* list, implemented using *exists*. The first query checks whether some user is called Monica. The programmer, perhaps wishing not to disclose the fact that he is looking for this particular person, has marked the string `"Monica"` with the label *Priv*. The second query looks for a user with an empty password string.

According to the type inference algorithm, the type of *query1* is $\mathtt{bool}^{Priv}$. Thus, the query's result reveals some information about the string `"Monica"`. Notice, however, that it does not carry the label *Sys*: it does not leak anything about the passwords contained in the list *users*. The type of *query2*, on the other hand, is $\mathtt{bool}^{Sys}$, which tells that it does contain information about the passwords.

If these information channels are deemed undesirable, they can be easily eliminated by adding typing assertions to the program. For instance, if *Public* is a label such that neither $Priv \preccurlyeq Public$ nor $Sys \preccurlyeq Public$ hold, then writing

```
let query1 : bool^Public = ...
let query2 : bool^Public = ...
```

causes both definitions to become ill-typed, thus revealing and forbidding the leaks. Thus, typing assertions may be used to express, and statically enforce, a security policy.

9 Access control

Information flow analysis offers a way of proving an untrusted program correct with respect to a security policy. However, it is a restrictive discipline, since it does not allow *declassification*. For instance, a function which compares a secret password string against a given input must return a secret result, even though it usually yields far less than one bit of information about the password. Thus, some useful programs cannot be proved correct; for this reason, *trust*, in the form of *access control*, must be re-introduced.

9.1 A calculus with access control

Let us briefly describe explicit access control. Assume given a fixed set of *principals* $\mathcal{P}$, equipped with an arbitrary binary relation $\succcurlyeq$. The assertion $p \succcurlyeq q$ intuitively means that p *acts for* q, i.e. q trusts p. As a result, q grants p the ability of directly accessing any value to which q has access. Assume the calculus' syntax includes the following productions:

$$\begin{array}{rcll} e & ::= & & \text{terms} \\ & \mid & \mathrm{lock}_p & \text{locking } (p \in \mathcal{P}) \\ & \mid & \mathrm{unlock}_p & \text{unlocking } (p \in \mathcal{P}) \\ & \mid & \ldots & \end{array}$$

Assume its semantics includes the following reduction rule:

$$\text{unlock}_p\ (\text{lock}_q\ e) \rightarrow e \qquad \text{if } p \succcurlyeq q$$

Then, a value locked with q's authority becomes unusable until it is unlocked by some principal p which acts for q. Any attempt to unlock a value by an unauthorized principal results in a failure. Of course, in practice, some compiler and operating system support is required to ensure that unlock_p is only used in code which actually acts on behalf of principal p. This usually requires the use of cryptographic authentication techniques.

It is also possible to design a calculus with implicit access control, i.e. where every value is implicitly locked upon creation, and unlocked upon access, as in e.g. [12]. We discuss both cases below.

9.2 Typing

Again, extending an existing type system with access control features can be done abstractly, i.e. independently of the system's definition, using a translation-based approach. Let us briefly sketch how.

Assume given a target calculus with pairs $\langle\!\langle \cdot, \cdot \rangle\!\rangle$, plus, for every principal $p \in \mathcal{P}$, a constant p and a primitive operation actsfor_p. Require $(\text{actsfor}_p\ v)$ to be well-typed only if v is a constant $q \in \mathcal{P}$ such that $p \succcurlyeq q$. To easily meet this requirement, the target system's implementor may wish to assume $(\mathcal{P}, \succcurlyeq)$ forms a lattice. Define a type system for the source calculus by lifting the target system through a simple encoding:

$$\begin{array}{rcl} [\![\text{lock}_p]\!] & = & \lambda x.\langle\!\langle x, p \rangle\!\rangle \\ [\![\text{unlock}_p]\!] & = & \lambda x. \\ & & \quad \text{letp } \langle\!\langle x, q \rangle\!\rangle = x \text{ in} \\ & & \quad \text{actsfor}_p\ q\,; \\ & & \quad x \end{array}$$

Then, the derived type system enjoys subject reduction and progress properties. In particular, access control is entirely *static*: if a program is well-typed, then all of its access control checks must succeed. As a result, all checks can be compiled away. In other words, the above encoding only serves *typing*, not *compilation*, purposes.

9.3 Combining information flow and access control

Information flow and access control may coexist. Extend the syntax and semantics of the source calculus presented in section 3 with (explicit or implicit) access control features, while preserving its stability property. Find a typed target calculus equipped with principal constants $p \in \mathcal{P}$, label constants $l \in \mathcal{L}$, and suitable operations thereon. Then, lift the target type system through an appropriate encoding.

If the source calculus has implicit access control, then a simple encoding, where every expression e is mapped to a triple $\langle e_c, e_p, e_l \rangle$, will do. The components of the triple respectively represent e's computational content, the principal whose authority has been used to lock e, and e's label. This yields a system where every type carries two annotations, a principal and a label.

If the source calculus has explicit access control operations, then a different encoding must be used. Map every expression e to a pair $\langle \cdot, e_l \rangle$, whose first component is $\langle\!\langle e_c, e_p \rangle\!\rangle$, if e is locked, and simply e_c otherwise. This is *exactly* the encoding presented in section 5, extended to deal with lock_p and unlock_p. As before, it yields a system where every type carries one (information flow) label. A value of computational type α, carrying a label φ, will receive type α^φ if it is unlocked, and $(\alpha\ \pi\ \texttt{locked})^\varphi$ if it is locked at level π. Here, `locked` is the type constructor associated with pairs of the form $\langle\!\langle \cdot, \cdot \rangle\!\rangle$ in the target system.

In common programs, access control features should be used only at a few key places. For this reason, making access control explicit, rather than implicit, may be preferable. Indeed, this approach yields types which are usually more concise, and where access control restrictions are syntactically more apparent.

Marrying information flow analysis with access control is not a new idea. Stoughton [25] and Heintze and Riecke [12] notice that access control and information flow control serve different purposes, and propose hybrid systems where both mechanisms coexist.

However, these works fail, in our opinion, to make a crucial point: the two mechanisms are entirely *unrelated*, and should coexist *without* interacting. Indeed, access control involves *principals*, *trust* and *authentication*, while information flow control requires neither. Furthermore, describing access control usually involves introducing some form of *security violation* in the language's semantics, while information flow control does not. Why? Access control implements a fixed security policy, defined by $(\mathcal{P}, \succcurlyeq)$; it is meaningless in the absence of such a policy. Information flow control, on the other hand, does make sense even when $(\mathcal{L}, \preccurlyeq)$ is left unspecified, as pointed out in section 8.3, because it is only a *dependency* analysis. It does *not*, fundamentally, have anything to do with security, which explains why it can be formalized without a notion of security violation.

Why such emphasis on this point? Both Stoughton [25] and Heintze and Riecke [12] define systems where access control and information flow *interact*, by identifying principals with labels, i.e. setting $\mathcal{P} = \mathcal{L}$, and requiring every value to carry an information flow label l which is *less restrictive* than its access control label p, i.e. $l \preccurlyeq p$. Furthermore, [12] defines an operational semantics where both kinds of labels interact: the expression $(\texttt{protect}_{ir}\ v)$ uses the information flow label ir to update not only v's information flow label, but also its access control label.

In these works, the alleged justification for requesting $l \preccurlyeq p$ is as follows. p tells who may use the value *directly*, while l tells who may use it *indirectly*, i.e. have (possibly partial) access to the information contained in it. Because any principal who is granted direct access is thereby granted indirect access at the same time, requiring $l \preccurlyeq p$ may seem natural. We deem it wrong, however, because these notions are really orthogonal: while p indeed tells *who* may use the value, l tells *which* information it contains. Mixing the two mechanisms yields a needlessly complex system. Separating them makes the system more modular, conceptually simpler, and potentially more expressive, since $\mathcal{P}$ and $\mathcal{L}$ may be distinct.

Myers and Liskov [15, 14] propose a "decentralized" label model which is a subtle mixture of access control and information flow control. The model also rests on a set of principals $(\mathcal{P}, \succcurlyeq)$. A *label* is a set of tagged *policies*, where the tag carried by every policy is a principal, called its *owner*. A *policy* is a set of principals, called *readers*. Labels form a pre-order, whose underlying order is a lattice; it is used, as in this paper, to perform information flow analysis. However, Myers and Liskov also allow a number of "safe" declassification operations: a principal p may choose to *relax* the label carried by a given value, by arbitrarily modifying any policy owned by a principal q which it acts for. Of course, p is not allowed to affect the policies owned by principals whose trust it has not received. So, labels do not only carry *dependency* information; they also contain *access control* information, since the use of declassification is restricted.

We think Myers and Liskov's model has significant practical interest. However, we believe that comparable expressive power[3] can be achieved in a theoretically simpler system. Indeed, imagine orthogonal access control and information flow control, as suggested above. Then, one may selectively allow declassification by providing, in the initial typing environment, a number of declassification operations, *locked* at appropriate levels of authority. The sets $\mathcal{P}$ and $\mathcal{L}$ may, in general, be chosen independently; only the types of the declassification operations provide a connection between the two. This presentation of the system is modular and abstract. By varying $\mathcal{P}$, $\mathcal{L}$, and the level of authority required by each declassification operation, one obtains a wide range of concrete systems, some of which are in fact very close to Myers and Liskov's, and have comparable expressiveness.

We prefer to present declassification as *selective*, rather than *safe*, since its use breaks the non-interference property – at least partially. Although it is only a matter of terminology, speaking of "safe" declassification is somewhat misleading: this sort of declassification is only safe for principals whose authority is *not* granted to the operation.

Let us illustrate our proposal with a very simple example, inspired from the ACCAT Guard [6]. Assume $\mathcal{L}$ is the lattice product of the 2-point lattice $\text{Secret} = \{\text{L} \preccurlyeq \text{H}\}$ with some unspecified lattice $\mathcal{M}$. Thus, in a calculus with (say) implicit access control, types will be of the form $\alpha^{\pi,\,(\lambda,\,\mu)}$, where π, λ and μ range over $\mathcal{P}$, Secret and $\mathcal{M}$, respectively. Assume the initial typing environment offers the binding

$$\mathit{declass} : \forall\alpha\pi\mu.(\alpha^{\pi,\,(\text{H},\,\mu)} \rightarrow \alpha^{\pi,\,(\text{L},\,\mu)})^{\text{SWO},\,\epsilon}$$

where $\text{SWO} \in \mathcal{P}$ is a fixed principal. Then, information may freely flow from level (L, m) to level (H, m), for any $m \in \mathcal{M}$, since the former is a sub-type of the latter. However, the only way of allowing flows in the reverse direction is to use *declass*, which requires approval by the principal SWO, since it must be unlocked when invoked. This allows modeling a "guard", i.e. a gateway between a classified and a non-classified system, where flows which appear to violate security must be approved by a trusted Security Watch Officer. Furthermore, notice that even the principal SWO may not perform arbitrary declassifications: it is unable to modify the second component of labels. Thus, a partial non-interference result holds: a result whose information label is $(\cdot\,, m)$ cannot depend on any input whose label is $(\cdot\,, n)$, where $n \notin \downarrow m$. For instance, if *Nuclear* and *Strategic* are incomparable elements of $\mathcal{M}$, then a computation whose result type is $(\text{L}, \mathit{Nuclear})$ cannot leak any information of type $(\text{L}, \mathit{Strategic})$, even though it may reveal some information of level $(\text{H}, \mathit{Nuclear})$. This is exactly what Myers and Liskov term "safe" declassification.

10 Discussion

We have shown how to systematically extend an arbitrary type system with dependency information, and how soundness and non-interference proofs for the new system may rely upon, rather than duplicate, the soundness proof of the original system. This allows enriching virtually any of the type systems known today with information flow analysis, while requiring only a minimal proof effort.

We recently became aware of Ross and Sagiv's reduction of a flow dependence problem to a may-alias problem [23]. Although the programming language (first-order, imperative vs. higher-order, functional) and the target system (pointer analysis vs. type inference) considered are rather different from ours, both papers rely on a similar encoding, where every value is translated to a pair of a value and a tag. We take this as evidence of the strength of this reductionistic approach.

Our work complements Abadi *et al.*'s [1]. They show that several program analyses, including secrecy and integrity analyses, program slicing, and binding-time analysis, are dependency analyses, which only differ by the choice of the information lattice $\mathcal{L}$. As a unifying dependency calculus, they propose a simply-typed λ-calculus, based on Moggi's computational λ-calculus. In turn, we show that it is possible to enrich any standard type system with dependency information. Combining these results yields expressive type systems for all of the analyses above.

By varying $\mathcal{L}$, a dependency analysis may be used to obtain secrecy or integrity guarantees about a program. It is interesting to notice that both may be obtained at the same time, *without* requiring two annotations per type: one is enough, provided $\mathcal{L}$ is the *product* of a secrecy lattice with an integrity lattice. The same trick can be applied to access control: by labeling data with "locks", rather than principals, and choosing the lattice of locks to be the lattice product of $(\mathcal{P}, \succcurlyeq)$ with its own dual, a single annotation suffices to manage and enforce restrictions on value access *and* creation. Thus, extending the SLam calculus to deal with integrity [12, section 4] was unnecessary: not only is it enough to maintain two security annotations, rather than four, but no new correctness proof needs be given.

In fact, in a type system enriched with dependency annotations, a *polymorphic* type scheme (such as that of *exists*, given in section 8.3) *fully* and *abstractly* describes the dependencies induced by a piece of code. (This idea appears in several previous works, e.g. [6, 5, 27].) Indeed, it documents not only the behavior of the code at different security levels, but also within different security lattices. This explains the remark of the previous paragraph: rather than add new annotations, use a new lattice.

We have argued that access control and information flow control should be implemented independently. The latter is *not* a refinement of the former; they are different mechanisms. One is based on *trust*, the other on *proof*. It is possible, however, to let them coexist within a single design. We have shown that this gives rise to interesting possibilities, including *selective* declassification.

To conclude, we believe we have found a very lightweight approach to non-interference proofs. It is based on an untyped operational semantics for a labelled calculus, together with a translation to an unlabelled calculus. Two basic results must be proved: a stability theorem, which states that the labelled semantics never "drops" labels, and a simulation lemma, which shows that the translation is meaningful. Because of its simplicity, this approach should be directly applicable to other computing paradigms, such as object or process calculi. We are currently investigating this issue.

[3] assuming $(\mathcal{P}, \succcurlyeq)$ is fixed, i.e. may not vary at runtime.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Conference Record of the 26th ACM Symposium on Principles of Programming Languages*, pages 147–160, San Antonio, Texas, Jan. 1999. URL: http://pa.bell-labs.com/~abadi/Papers/flowpopl.ps.

[2] M. Abadi, B. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 83–91, Philadelphia, Pennsylvania, May 1996. URL: http://pa.bell-labs.com/~abadi/Papers/make-preprint.ps.

[3] A. Aiken, M. Fähndrich, J. S. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. *Lecture Notes in Computer Science*, 1473:78, 1998. URL: http://www.cs.berkeley.edu/~aiken/papers/tic98.ps.

[4] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, Jan. 1980.

[5] J.-P. Banâtre, C. Bryce, and D. Le Métayer. Compile-time detection of information flow in sequential programs. In D. Gollmann, editor, *Proceedings of the 3rd European Symposium on Research in Computer Security*, volume 875 of *Lecture Notes in Computer Science*, pages 55–74. Springer Verlag, 1994. URL: ftp://ftp.irisa.fr/local/lande/dlm-esorics94.ps.Z.

[6] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[7] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[8] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, May 1974.

[9] C. Fournet, L. Maranget, C. Laneve, and D. Rémy. Implicit typing à la ML for the join-calculus. In *8th International Conference on Concurrency Theory (CONCUR'97)*, volume 1243 of *Lecture Notes in Computer Science*, pages 196–212, Warsaw, Poland, 1997. Springer. URL: ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/typing-join.ps.gz.

[10] Y.-C. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In J. Díaz and F. Orejas, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development : Vol. 2*, volume 352 of *LNCS*, pages 167–183, Berlin, Mar. 1989. Springer.

[11] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, Apr. 1982.

[12] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Conference Record of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, San Diego, California, Jan. 1998. URL: http://cm.bell-labs.com/cm/cs/who/nch/slam.ps.

[13] B. W. Lampson. A note on the confinement problem. *Communications of the Association for Computing Machinery*, 16(10):613–615, Oct. 1973. URL: http://research.microsoft.com/lampson/11-Confinement/WebPage.html.

[14] A. C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachusetts Institute of Technology, Jan. 1999. Technical Report MIT/LCS/TR-783. URL: http://www.cs.cornell.edu/andru/release/tr783.ps.gz.

[15] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 186–197, May 1998. URL: http://www.cs.cornell.edu/andru/papers/sp98/top.html.

[16] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1), 1999. URL: http://www.cs.mu.oz.au/~sulzmann/publications/tapos.ps.

[17] P. Ørbæk and J. Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557–591, Nov. 1997. URL: http://www.cs.purdue.edu/homes/palsberg/paper/jfp97.ps.gz.

[18] J. Palsberg and P. Ørbæk. Trust in the λ-calculus. *Lecture Notes in Computer Science*, 983:314–330, 1995. URL: ftp://ftp.daimi.au.dk/pub/empl/poe/lambda-trust.dvi.gz.

[19] F. Pottier. Simplifying subtyping constraints: a theory. Submitted for journal publication, Dec. 1998. URL: http://pauillac.inria.fr/~fpottier/publis/fpottier-journal-98.ps.gz.

[20] F. Pottier. Type inference in the presence of subtyping: from theory to practice. Technical Report 3483, INRIA, Sept. 1998. URL: ftp://ftp.inria.fr/INRIA/publication/RR/RR-3483.ps.gz.

[21] D. Rémy. Projective ML. In *1992 ACM Conference on Lisp and Functional Programming*, pages 66–75, New-York, 1992. ACM Press. URL: ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/lfp92.ps.gz.

[22] D. Rémy and J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 40–53, Paris, France, Jan. 1997. URL: ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/objective-ml!popl97.ps.gz.

[23] J. L. Ross and M. Sagiv. Building a bridge between pointer aliases and program dependences. *Nordic Journal of Computing*, 5(4):361–386, 1998. URL: http://www.math.tau.ac.il/~sagiv/njc98.ps.

[24] G. S. Smith. Polymorphic type inference with overloading and subtyping. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93*, volume 668 of *Lecture Notes in Computer Science*, pages 671–685. Springer-Verlag, Apr. 1993.

[25] A. Stoughton. Access flow: A protection model which integrates access control and information flow. In *Proceedings of the 1981 IEEE Symposium on Security and Privacy*, pages 9–18, 1981.

[26] V. Trifonov and S. Smith. Subtyping constrained types. In *Proceedings of the Third International Static Analysis Symposium*, volume 1145 of *LNCS*, pages 349–365. SV, Sept. 1996. URL: `http://www.cs.jhu.edu/~trifonov/papers/subcon.ps.gz`.

[27] D. Volpano and G. Smith. A type-based approach to program security. *Lecture Notes in Computer Science*, 1214:607–621, Apr. 1997. URL: `http://www.cs.nps.navy.mil/people/faculty/volpano/papers/tapsoft97.ps.Z`.

[28] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, Nov. 1994. URL: `http://www.cs.rice.edu/CS/PLT/Publications/ic94-wf.ps.gz`.