Information Flow Inference for ML

François Pottier*
Francois.Pottier@inria.fr

Vincent Simonet*
Vincent.Simonet@inria.fr

Abstract

This paper presents a type-based information flow analysis for a call-by-value λ-calculus equipped with references, exceptions and let-polymorphism, which we refer to as Core ML. The type system is constraint-based and has decidable type inference. Its non-interference proof is reasonably lightweight, thanks to the use of a number of orthogonal techniques. First, a syntactic segregation between *values* and *expressions* allows a lighter formulation of the type system. Second, non-interference is reduced to *subject reduction* for a non-standard language extension. Lastly, a *semi-syntactic* approach to type soundness allows dealing with constraint-based polymorphism separately.

1 Introduction

Information flow analysis consists in statically determining how a program's outputs are related to its inputs, i.e. how the former *depend*, directly or indirectly, on the latter. This allows establishing *secrecy* and *integrity* properties of a program, i.e. proving that some aspects of its behavior convey no information about those of its inputs deemed "secret", or remain independent of those deemed "unreliable". These properties are instances of *non-interference* [7]: they state the absence of certain dependencies.

Because information flow analysis is complex and error-prone, it must be automated. During the past few years, several researchers have advocated its formulation as a *type system*. Then, existing type inference techniques provide automation, while type signatures provide concise, formal security specifications.

Our interest is in designing – and proving correct – a type-based information flow analysis for (the kernel of) a realistic sequential programming language. (In the presence of concurrency, the termination of a process is observable by other processes, creating new ways to leak information and requiring more restrictive type systems. Hence, it appears reasonable to first experiment with information flow control in a sequential setting.) To date, most formal results obtained in this area concern extremely reduced programming languages. Several papers address pure λ-calculi [8, 1, 16]. Volpano *et al.* [22, 21] study a core imperative programming language, where all variables store integers. Standing in sharp contrast, Myers [10, 11] considers the full Java language, including objects, exceptions, parameterized classes, etc. However, he does not give a formal proof of correctness; indeed, our formal approach uncovered a couple of flaws in his type system (see section 7.3).

In an attempt to bridge the gap between these approaches, we consider a call-by-value λ-calculus equipped with references, exceptions and let-polymorphism, which we refer to as Core ML. (Presentation set aside, it is identical to Wright and Felleisen's Core ML [24], except our exception names have global scope and are not first-class values.) Such a calculus can be viewed as the core of the functional programming language Caml-Light [9]. We endow it with a polymorphic, constraint-based type system, called MLIF, which has decidable type inference and guarantees non-interference.

A (monomorphic) treatment of references in a higher-order language can be found in [25]. Exceptions have been studied by Myers [10, 11] for Java. However, Myers' treatment relies on Java's explicit, monomorphic throws clauses, whereas our type system uses a more flexible, polymorphic effect analysis, giving rise to issues discussed in section 10. The combination of references, exceptions and constrained let-polymorphism, as well as our use of a standard subject reduction technique to establish non-interference, are novel. Our treatment of un-annotated tuple types and of polymorphic equality form ancillary contributions.

2 Overview

Type systems are typically used to establish *safety* properties, i.e. prove that a certain invariant holds throughout the execution of a program. Type safety is such a property. However, non-interference [7] requires *two* independent program runs, given different inputs, to yield the same output. As a result, its proof is often more delicate.

Abadi *et al.* [2] devised a *labelled* operational semantics of the λ-calculus, where the labels attached to a term indicate how much information it carries. Executing a program under such a semantics amounts to performing a *dynamic* dependency analysis along with the actual computation. Pottier and Conchon [16] later showed how *static*, type-based dependency analyses could be systematically derived, and proven safe, from such a labelled semantics.

Unfortunately, in a programming language with side ef-

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

To be presented at the 29th ACM Symposium on Principles Of Programming Languages, Portland, Oregon, January 2002.

fects, it is possible to leak information through the *absence* of a certain effect. Indeed, consider the program fragment "$\mathsf{if}\ x = 1\ \mathsf{then}\ y := 1$". If, after executing this statement, y isn't 1, then x cannot be 1 either. Thus, in that case, execution transfers information about x to y, even though no assignment takes place, since the statement $y{:=}1$ is skipped. It appears difficult for a labelled semantics to account for the effect of code that is *not* executed; so, the approach must be reconsidered.

Direct non-interference proofs, although straightforward for simple programming languages [22], become increasingly complex in richer languages, requiring cumbersome invariants to be manipulated [25]. To avoid this pitfall, we break our proof down into several independent steps. First, we define a special-purpose extension of the language, which allows explicit reasoning about the commonalities and differences between *two* arbitrary program configurations, and prove it adequate in a certain sense. Then, we define a type system for this extended language, and prove that it enjoys a subject reduction property. Lastly, we show that non-interference for the base language is a consequence of these results. In other words, we reduce the initial problem to subject reduction – a safety property – for our special-purpose language. The invariant preserved by reduction is thus expressed in the type system itself, making it easier to reason about.

In keeping with the ML tradition, our type system has let-polymorphism and type inference. In addition to structure, our types describe effects and security levels; polymorphism allows writing code that is generic with respect to all three. Type inference is an indispensable help, because our types are verbose and information flow is often un-intuitive. Because we employ subtyping (as well as other forms of constraints), our type inference system is constraint-based. Yet, if generalization, instantiation, and constraint manipulation were part of the type system from the outset, our subject reduction proof would be significantly obfuscated. To work around this problem, we adopt a *semi-syntactic* approach [15], which again consists in breaking down the construction into two steps. First, we present a system equipped with an extensional form of polymorphism, whose formal treatment is remarkably un-intrusive. Then, we build a constraint-based system in the style of HM(X) [12], which we prove correct with respect to the former.

We will now proceed as follows. We first present the syntax of Core ML (section 3). Then, we introduce our technical extension of it, which we refer to as "Core ML^2", give an operational semantics to both languages at once, and show how they relate to each other (section 4). Section 5 introduces MLIF$_0$, a type system for Core ML^2, and establishes subject reduction. Combining these results, we obtain a non-interference property for Core ML (section 6). In section 7, we digress and discuss a few language extensions. Culminating our development, section 8 presents MLIF, a constraint-based type system which we prove correct with respect to MLIF$_0$, allowing type inference. Sections 9 and 10 give some examples and conclude.

3 Core ML

Let k range over integers; let x, m, ε range over disjoint denumerable sets of *program variables*, *memory locations*, and *exception names*, respectively. Then, *values*, *outcomes*, *expressions* and *evaluation contexts* are defined as follows:

$$
\begin{array}{rcl}
v & ::= & x \mid \mathsf{fix}\, f.\lambda x.e \mid k \mid () \mid m \mid \varepsilon\, v \\
o & ::= & v \mid \mathsf{raise}\ (\varepsilon\, v) \\
e & ::= & o \\
 & \mid & v\, v \\
 & \mid & \mathsf{ref}\ v \mid v := v \mid !\, v \\
 & \mid & \mathsf{raise}\ v \\
 & \mid & \mathsf{let}\ x = v\ \mathsf{in}\ e \\
 & \mid & E[e] \\
E & ::= & \mathsf{bind}\ x = [\,]\ \mathsf{in}\ e \\
 & \mid & [\,]\ \mathsf{handle}\ \varepsilon\, x \succ e \\
 & \mid & [\,]\ \mathsf{handle}\ x \succ e
\end{array}
$$

Our values include variables, λ-abstractions, integers, a unit constant, memory locations, and exceptions. An abstraction $\mathsf{fix}\, f.\lambda x.e$ may recursively refer to itself through the program variable f. (This is done merely to avoid dealing with recursion separately.) Every exception name ε can be used as a data constructor to build exception values of the form $\varepsilon\, v$. Outcomes, known as *answers* in [24], represent inactive computations; they are either values or unhandled exceptions of the form $\mathsf{raise}\ (\varepsilon\, v)$. An expression is an outcome, a so-called *basic expression*, a let construct, or another expression enclosed within an evaluation context.

Basic expressions include function applications as well as instances of four primitive operations, which allow allocating, updating, dereferencing memory cells, and raising exceptions. They are built out of values, rather than out of arbitrary sub-expressions. This syntactic restriction, which is reminiscent of Flanagan *et al.*'s *A-normal forms* [6], offers a number of advantages. First, it enables a lighter formulation of our type-and-effect system. Indeed, because values have no computational effect, a basic expression's sub-expressions do not contribute to its effect. Furthermore, it allows our system to remain independent of the evaluation strategy, i.e. of the choice of left-to-right vs. right-to-left evaluation order. User programs, expressed in a more liberal syntax, must be translated down into our restricted syntax before they can be analyzed; different evaluation strategies will simply correspond to different translation schemes (see section 5.7).

The let construct $\mathsf{let}\ x = v\ \mathsf{in}\ e$ has the same meaning as the basic expression $(\mathsf{fix}\, f.\lambda x.e)\, v$ (where f is not free in e). However, as usual in ML [24], the let keyword directs the type checker to give x polymorphic type. Following Wright [23], we require the binding to contain a value v, rather than an arbitrary sub-expression, so as to avoid unsoundness in the presence of imperative features. As a result, let constructs do not appear among evaluation contexts.

Evaluation contexts provide glue to combine expressions and specify their evaluation order. The expression $\mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2$ evaluates e_1, binds its value (if any) to x, then evaluates e_2. The bind keyword does not request type generalization; it merely expresses sequentiality. Our decision of making let and bind separate constructs emphasizes this distinction. The handle constructs are dual to bind: they specify what happens after the expression under scrutiny raises an exception, rather than after it returns a value.

The meaning of the memory locations which occur in a Core ML expression is given by a *store* μ, i.e. a partial map from memory locations to values. We write $\mu[m \mapsto v]$ and $\mu \oplus [m \mapsto v]$ for the store which maps m to v and otherwise agrees with μ; the latter is defined only if $m \notin \mathrm{dom}(\mu)$.

$$
\begin{array}{lrcll}
\textbf{Basic reductions} & (\mathsf{fix}\, f.\lambda x.e)\, v /_i \mu & \rightarrow & e[x \Leftarrow v][f \Leftarrow \mathsf{fix}\, f.\lambda x.e] /_i \mu & (\beta) \\
& \mathsf{ref}\ v /_i \mu & \rightarrow & m /_i \mu \oplus [m \mapsto \mathrm{new}_i\, v] & (\mathrm{ref}) \\
& m := v /_i \mu & \rightarrow & () /_i \mu[m \mapsto \mathrm{update}_i\, \mu(m)\, v] & (\mathrm{assign}) \\
& !\, m /_i \mu & \rightarrow & \mathrm{read}_i\, \mu(m) /_i \mu & (\mathrm{deref}) \\
& \mathsf{let}\ x = v\ \mathsf{in}\ e /_i \mu & \rightarrow & e[x \Leftarrow v] /_i \mu & (\mathrm{let}) \\
\\
\textbf{Sequencing} & \mathsf{bind}\ x = v\ \mathsf{in}\ e /_i \mu & \rightarrow & e[x \Leftarrow v] /_i \mu & (\mathrm{bind}) \\
& \mathsf{raise}\ (\varepsilon\, v)\ \mathsf{handle}\ \varepsilon\, x \succ e /_i \mu & \rightarrow & e[x \Leftarrow v] /_i \mu & (\mathrm{handle}) \\
& \mathsf{raise}\ (\varepsilon\, v)\ \mathsf{handle}\ x \succ e /_i \mu & \rightarrow & e[x \Leftarrow \varepsilon\, v] /_i \mu & (\mathrm{handle\text{-}all}) \\
& E[o] /_i \mu & \rightarrow & o /_i \mu & (\mathrm{throw\text{-}context}) \\
& & & \text{if } \neg(E \text{ handles } \lfloor o \rfloor_1 \vee E \text{ handles } \lfloor o \rfloor_2) & \\
\\
\textbf{Lifting} & E[\langle o_1 \mid o_2 \rangle] / \mu & \rightarrow & \langle \lfloor E \rfloor_1[o_1] \mid \lfloor E \rfloor_2[o_2] \rangle / \mu & (\mathrm{lift\text{-}context}) \\
& & & \text{if none of the sequencing rules applies} & \\
& \langle v_1 \mid v_2 \rangle\, v / \mu & \rightarrow & \langle v_1\, \lfloor v \rfloor_1 \mid v_2\, \lfloor v \rfloor_2 \rangle / \mu & (\mathrm{lift\text{-}app}) \\
& \langle v_1 \mid v_2 \rangle := v / \mu & \rightarrow & \langle v_1 := \lfloor v \rfloor_1 \mid v_2 := \lfloor v \rfloor_2 \rangle / \mu & (\mathrm{lift\text{-}assign}) \\
& !\, \langle v_1 \mid v_2 \rangle / \mu & \rightarrow & \langle !\, v_1 \mid !\, v_2 \rangle / \mu & (\mathrm{lift\text{-}deref}) \\
& \mathsf{raise}\ \langle \varepsilon_1\, v_1 \mid \varepsilon_2\, v_2 \rangle / \mu & \rightarrow & \langle \mathsf{raise}\ (\varepsilon_1\, v_1) \mid \mathsf{raise}\ (\varepsilon_2\, v_2) \rangle / \mu & (\mathrm{lift\text{-}raise})
\end{array}
$$

$$
\textbf{Reduction under a context} \qquad \frac{e /_i \mu \rightarrow e' /_i \mu'}{E[e] /_i \mu \rightarrow E[e'] /_i \mu'} \qquad (\mathrm{context})
$$

$$
\frac{e_i /_i \mu \rightarrow e'_i /_i \mu' \qquad e_j = e'_j \qquad \{i, j\} = \{1, 2\}}{\langle e_1 \mid e_2 \rangle / \mu \rightarrow \langle e'_1 \mid e'_2 \rangle / \mu'} \qquad (\mathrm{bracket})
$$

$$
\begin{array}{lrclrclrcl}
& \mathrm{new}_\bullet\, v & = & v & \mathrm{update}_\bullet\, v\, v' & = & v' & \mathrm{read}_\bullet\, v & = & v \\
\textbf{Auxiliary functions} & \mathrm{new}_1\, v & = & \langle v \mid \mathsf{void} \rangle & \mathrm{update}_1\, v\, v' & = & \langle v' \mid \lfloor v \rfloor_2 \rangle & \mathrm{read}_1\, v & = & \lfloor v \rfloor_1 \\
& \mathrm{new}_2\, v & = & \langle \mathsf{void} \mid v \rangle & \mathrm{update}_2\, v\, v' & = & \langle \lfloor v \rfloor_1 \mid v' \rangle & \mathrm{read}_2\, v & = & \lfloor v \rfloor_2
\end{array}
$$

Figure 1: Operational semantics of Core ML^2

4 Core ML^2

4.1 Presentation

Non-interference requires reasoning about two programs and proving that they share some sub-terms throughout execution. To make such reasoning easier, we choose to represent them as a *single* term of an extended language, called Core ML^2, rather than as a pair of Core ML terms. The extension is as follows:

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

The Core ML^2 term $\langle e_1 \mid e_2 \rangle$ is intended to encode the pair of Core ML terms (e_1, e_2). It is important to note that it can appear at an arbitrary depth within a term. For instance, assuming v is a Core ML value, the terms $\langle v_1 \mid v_2 \rangle\, v$ and $\langle v_1\, v \mid v_2\, v \rangle$ both encode the pair $(v_1\, v, v_2\, v)$. The former, however, is more informative, because it explicitly records the fact that the application node and its argument v are shared, while the latter doesn't. We do not allow nesting $\langle \cdot \mid \cdot \rangle$ constructs.

We need to keep track of sharing not only between expressions, but also between stores. However, distinct stores may have distinct domains. To account for this fact, we introduce a special constant void. By creating bindings of the form $m \mapsto \langle v \mid \mathsf{void} \rangle$ and $m \mapsto \langle \mathsf{void} \mid v \rangle$ in the store, we represent situations where a memory location m is bound within only one of the two Core ML expressions encoded by a Core ML^2 term.

A *configuration* $e /_i \mu$ is a triple of an expression e, a store μ, and an index $i \in \{\bullet, 1, 2\}$, whose purpose is explained in section 4.2. We write e / μ for $e /_\bullet \mu$.

A configuration $e /_i \mu$ is *well-formed* if the following conditions hold:

- e does not contain void; furthermore, if $i \in \{1, 2\}$, then e is a Core ML expression;
- for each $m \in \mathrm{dom}(\mu)$, $\mu(m)$ is of the form v, $\langle v \mid \mathsf{void} \rangle$ or $\langle \mathsf{void} \mid v \rangle$, where v does not contain void.

Furthermore, we consider a memory location m to be bound within e and μ according to the following rules:

- if $\mu(m)$ is of the form v, then m is in scope everywhere within e and μ;
- if $\mu(m)$ is of the form $\langle v \mid \mathsf{void} \rangle$ (resp. $\langle \mathsf{void} \mid v \rangle$), then:
  - m is in scope within the left (resp. right) branch of every $\langle \cdot \mid \cdot \rangle$ construct in μ;
  - if $i = \bullet$, then m is in scope within the left (resp. right) branch of every $\langle \cdot \mid \cdot \rangle$ construct in e; if $i = 1$ (resp. $i = 2$), then m is in scope within e.

A configuration $e /_i \mu$ is *closed* if all occurrences of memory locations in it are in scope. We restrict our attention to well-formed, closed configurations. (These technical notions are preserved by reduction and guarantee that void is used exclusively in store bindings, as described above.) Furthermore, we identify configurations up to consistent renamings of memory locations.

The correspondence between Core ML and Core ML^2 is made explicit by means of two *projection* functions $\lfloor\cdot\rfloor_i$, where i ranges over $\{1,2\}$. They satisfy $\lfloor\langle e_1 \mid e_2\rangle\rfloor_i = e_i$ and are homomorphisms on other expression forms. They are extended to stores as follows: $\lfloor\mu\rfloor_i$ maps m to $\lfloor\mu(m)\rfloor_i$ if and only if the latter is defined and isn't void. Lastly, the projection of a configuration is defined by $\lfloor e \,/\, \mu\rfloor_i = \lfloor e\rfloor_i \,/\, \lfloor\mu\rfloor_i$.

4.2 Semantics

The small-step operational semantics of Core ML^2 is given in figure 1. The first two groups of reduction rules are those of Core ML, with a few technical twists explained below. The rules in the third group are specific to Core ML^2; they allow discarding sharing information if reduction cannot otherwise take place. The rules in the fourth group allow reduction under a context.

The rules are designed so that the image of any reduction step through a projection function is again a valid reduction step. Reduction may take place outside brackets, causing both projections to perform the same reduction step; inside brackets, letting one projection compute independently, while the other remains stationary; or lift up the bracket boundary, discarding some sharing information, while leaving both projections unchanged.

The capture-free substitution of v for x in e, written $e[x \Leftarrow v]$, is defined in the usual way, except at $\langle\cdot\mid\cdot\rangle$ nodes, where we must use an appropriate projection of v in each branch: $\langle e_1 \mid e_2\rangle[x \Leftarrow v]$ is $\langle e_1[x \Leftarrow \lfloor v\rfloor_1] \mid e_2[x \Leftarrow \lfloor v\rfloor_2]\rangle$.

We would like the rules in the first two groups to be applicable under any context. However, (ref), (assign) and (deref) need a small amount of contextual information. Indeed, the store must be accessed in a context-dependent manner: operations which take place inside a $\langle\cdot\mid\cdot\rangle$ construct must use or affect only one projection of the store. The index i carried by configurations is used for this purpose. Its value is $\bullet$ when dealing with top-level reduction steps; it is made 1 (resp. 2) by rule (bracket) when reducing within the left (resp. right) branch of a $\langle\cdot\mid\cdot\rangle$ construct. It is used in the auxiliary functions new_i, update_i and read_i to access the store in an appropriate way.

The rules in the second group describe how values and exceptions are bound (i.e. handled) or propagated. We say that E handles o if and only if $E[o]$ is reducible through (bind), (handle) or (handle-all).

The rules in the third group have no computational content: they leave both projections unchanged. Their purpose is to prevent $\langle\cdot\mid\cdot\rangle$ constructs from blocking reduction, which is done by lifting them up, thus causing some sub-terms to be duplicated, but allowing reduction to proceed independently within each branch. For instance, the left-hand expression in (lift-app) is not a β-redex. In its reduct, the application node and the sub-term v are duplicated, allowing two β-redexes to appear. A somewhat analogous rule can be found in Abadi *et al.*'s labelled semantics of the λ-calculus [2]. To understand the significance of the "lift" rules, one must bear in mind that the contents of every $\langle\cdot\mid\cdot\rangle$ construct will be viewed as "secret". By causing new sub-terms to become secret during reduction, these rules actually provide an explicit description of information flow. Our design attempts to discard as little sharing information as possible; indeed, replacing all of these rules with $e \to \langle\lfloor e\rfloor_1 \mid \lfloor e\rfloor_2\rangle$, while computationally correct, would cause the type system to view every expression as "secret".

Our "lift" rules are not optimal, because there are situations where they discard sharing information which could conceivably be preserved, and because they never re-create sharing information; however, they are precise enough for our purposes, which is to prove a particular type system sound.

We remark that, because of rule (bracket), reductions under a bracket may be interleaved in an arbitrary order, causing non-determinism to arise. However, confluence is preserved, as stated below.

Lemma 4.1 (Confluence) *If $e \,/\, \mu \to e'_1 \,/\, \mu'_1$ and $e \,/\, \mu \to e'_2 \,/\, \mu'_2$, then there exists a configuration $e' \,/\, \mu'$ such that $e'_1 \,/\, \mu'_1 \to^= e' \,/\, \mu'$ and $e'_2 \,/\, \mu'_2 \to^= e' \,/\, \mu'$.*

The semantics of Core ML can be obtained as a fragment of that of Core ML^2.

4.3 Relating Core ML^2 to Core ML

We now show that Core ML^2 is an appropriate tool to reason simultaneously about the execution of two Core ML programs. This is expressed by two properties. First, as explained above, the image of a valid reduction through projection remains a valid reduction. Conversely, if both projections of a term can be reduced to an outcome, then so can the term itself.

Lemma 4.2 *Let $i \in \{1,2\}$. If $e /_i \mu \to e' /_i \mu'$, then $e / \lfloor\mu\rfloor_i \to e' \,/\, \lfloor\mu'\rfloor_i$.*

Proof. By inspection of (ref), (assign) and (deref). □

Lemma 4.3 (Soundness) *Let $i \in \{1,2\}$. If $e / \mu \to e' / \mu'$, then $\lfloor e \,/\, \mu\rfloor_i \to^= \lfloor e' \,/\, \mu'\rfloor_i$.*

Proof. By inspection of the reduction rules and appeal to lemma 4.2. □

A configuration $e /_i \mu$ is *stuck* if it is irreducible and e isn't an outcome. It is *successful* if e is an outcome. The following lemma will be used in the proof of the completeness property.

Lemma 4.4 (Stuck Configurations) *If $e \,/\, \mu$ is stuck, then $\lfloor e \,/\, \mu\rfloor_i$ is stuck for some $i \in \{1,2\}$.*

Proof. By induction on the structure of e.

∘ *Case* $e = v_1\, v_2$. Because neither (β) nor (lift-app) is applicable, v_1 cannot be of the form $\langle v_{11} \mid v_{12}\rangle$ or $\mathsf{fix}\, f.\lambda x.e'$. As a result, for any $i \in \{1,2\}$, $\lfloor v_1\rfloor_i$ cannot be of the form $\mathsf{fix}\, f.\lambda x.e'$. It follows that $\lfloor e \,/\, \mu\rfloor_i$ is stuck.

∘ *Case* $e = (v_1 := v_2)$, $e = \,!v$. Similar to the previous case.

∘ *Case* $e = \mathsf{ref}\ v$, $e = (\mathsf{let}\ x = v\ \mathsf{in}\ e')$. $e \,/\, \mu$ is not stuck.

∘ *Case* $e = \mathsf{raise}\ v$. Because e isn't an outcome, v isn't of the form $\varepsilon\, v'$. Because (lift-raise) isn't applicable, v isn't of the form $\langle \varepsilon_1\, v_1 \mid \varepsilon_2\, v_2\rangle$. As a result, for some $i \in \{1,2\}$, $\lfloor v\rfloor_i$ cannot be of the form $\varepsilon\, v'$. It follows that $\lfloor e \,/\, \mu\rfloor_i$ is stuck.

∘ *Case* $e = E[e_1]$. By inspection of (bind), (handle), (handle-all) and (throw-context), one determines that if e_1 is an outcome, then $E[e_1]$ is reducible. So, e_1 is not an outcome, which implies that $e_1 \,/\, \mu$ is stuck. By induction hypothesis, $\lfloor e_1 \,/\, \mu\rfloor_i$ is stuck, for some $i \in \{1,2\}$. By inspection of the reduction rules, so is $F[\lfloor e_1\rfloor_i] \,/\, \lfloor\mu\rfloor_i$, for any evaluation context F; in particular, so is $\lfloor E[e_1] \rfloor_i \mathrel{/} \lfloor \mu \rfloor_i$, which is $\lfloor e \mathrel{/} \mu \rfloor_i$.

$\circ$ *Case* $e = \langle e_1 \mid e_2 \rangle$. Assume e / μ is stuck. By (bracket), both $e_1 /_1 \mu$ and $e_2 /_2 \mu$ are irreducible. Because e isn't an outcome, there exists $i \in \{1, 2\}$ such that e_i isn't an outcome. As a result, $e_i /_i \mu$ is stuck. It follows that $e_i / \lfloor \mu \rfloor_i$ is stuck as well. $\square$

Lemma 4.5 (Completeness) *Assume* $\lfloor e \mathrel{/} \mu \rfloor_i \rightarrow^\star o_i \mathrel{/} \mu'_i$ *for all* $i \in \{1, 2\}$. *Then, there exists a configuration* $o \mathrel{/} \mu'$ *such that* $e \mathrel{/} \mu \rightarrow^\star o \mathrel{/} \mu'$ *and, for all* $i \in \{1, 2\}$, $\lfloor o \mathrel{/} \mu' \rfloor_i = o_i \mathrel{/} \mu'_i$.

Proof. Let us first establish that e / μ does not admit an infinite reduction sequence. To this end, let us first notice that no infinite reduction sequence can consist exclusively of instances of the "lift" reduction rules. (Indeed, each of these rules moves some $\langle \cdot \mid \cdot \rangle$ constructor strictly closer to the term's root.) Furthermore, these are the only rules which leave both projections of a configuration unchanged. In light of this remark, if e / μ admits an infinite reduction sequence, then lemma 4.3 yields an infinite reduction sequence out of $\lfloor e \mathrel{/} \mu \rfloor_i$, for some $i \in \{1, 2\}$. However, this is impossible, because both $\lfloor e \mathrel{/} \mu \rfloor_1$ and $\lfloor e \mathrel{/} \mu \rfloor_2$ can be reduced to normal forms, and the semantics of the Core ML fragment is deterministic.

As a result, $e \mathrel{/} \mu$ can be reduced to an irreducible configuration. Let us now reason by induction on the number of steps in this reduction sequence.

First, assume the sequence is empty. Then, $e \mathrel{/} \mu$ is irreducible. If e / μ is stuck, then so is one of its projections, by lemma 4.4. However, this is impossible, since both $\lfloor e \mathrel{/} \mu \rfloor_1$ and $\lfloor e / \mu \rfloor_2$ can be reduced to successful configurations. So, $e \mathrel{/} \mu$ must be successful. Then, so must its projections; the result follows.

Next, assume the sequence begins with $e \mathrel{/} \mu \rightarrow e_0 \mathrel{/} \mu_0$. By lemma 4.3, $\lfloor e \mathrel{/} \mu \rfloor_i \rightarrow \lfloor e_0 \mathrel{/} \mu_0 \rfloor_i$ holds for all $i \in \{1, 2\}$. By lemma 4.1, this implies $\lfloor e_0 \mathrel{/} \mu_0 \rfloor_i \rightarrow^\star o_i \mathrel{/} \mu'_i$. Thus, we may apply the induction hypothesis to e_0 / μ_0, showing that it reduces to some $o \mathrel{/} \mu'$ such that $\lfloor o \mathrel{/} \mu' \rfloor_i = o_i \mathrel{/} \mu'_i$ holds for all $i \in \{1, 2\}$. The result follows. $\square$

Our completeness result requires both projections to converge; it is not applicable if one of them diverges. Indeed, define e as $\mathsf{bind}\ x = \langle \Omega \mid 0 \rangle\ \mathsf{in}\ 0$, where Ω is a non-terminating expression. Its right projection is $\mathsf{bind}\ x = 0\ \mathsf{in}\ 0$, which reduces to 0; yet, e cannot be reduced to any term whose right projection is 0, because e only reduces to itself. Such a formulation of completeness will naturally lead us to establish a *weak* non-interference result, whereby two programs can be guaranteed to yield the same result only if they both terminate. We do not aim at a *strong* non-interference result, because it would make little sense to plug information leaks related to termination without attacking timing leaks in general. Furthermore, such a result would require a much more restrictive type system.

In essence, the completeness lemma guarantees that we have provided enough "lift" rules to allow reducing all meaningful Core ML^2 expressions. In the next section, each of these rules will add one case to our subject reduction proof, forcing us to ensure that our type system accounts for all possible kinds of information flow.

5 Typing Core ML^2

We now give a type system, called MLIF$_0$, for Core ML^2. It is a *ground* type system: it has no type variables and deals with polymorphism in a simple, abstract way. As a result, it does not describe an algorithm; we will address this issue in section 8.

Throughout the paper, every occurrence of $*$ stands for a distinct anonymous meta-variable of appropriate kind.

5.1 Types

Let $(\mathcal{L}, \leq)$ be a lattice whose elements, denoted by ℓ and pc, represent *security levels*. (Following Denning [4], we typically use the meta-variable pc, rather than ℓ, when considering information obtained by observing the value of the "program counter".) *Types*, *rows* and *alternatives* are defined as follows:

$$
\begin{array}{rcl}
t & ::= & \mathsf{unit} \\
 & \mid & \mathsf{int}^\ell \\
 & \mid & (t \xrightarrow{pc\ [r]} t)^\ell \\
 & \mid & t\ \mathsf{ref}^\ell \\
 & \mid & r\ \mathsf{exn}^\ell \\
r & ::= & \{\varepsilon \mapsto a\}_{\varepsilon \in \mathcal{E}} \\
a & ::= & \mathsf{Abs} \\
 & \mid & \mathsf{Pre}\ pc
\end{array}
$$

A *row* r is an infinite, quasi-constant family of alternatives indexed by $\mathcal{E}$. (A family is *quasi-constant* if all but a finite number of its entries are equal.) We write $(\varepsilon : a; r)$ for the row whose element at index ε is a and whose other elements are given by the sub-row r, which is indexed by $\mathcal{E} \setminus \{\varepsilon\}$. We write $a \in r$ to indicate that a is a member of r's codomain.

Our types are those of ML's type system, decorated with extra annotations of two kinds.

First, we employ rows to keep track of exceptions, as in existing type-and-effect systems, such as Pessaux and Leroy's [13]. If an exception value has type $r\ \mathsf{exn}^*$, then the row r contains information about the exception's name. Specifically, for every $\varepsilon \in \mathcal{E}$, if $r(\varepsilon)$ is Abs, then the exception's name cannot be ε; if, on the other hand, it is $\mathsf{Pre}\ *$, then the exception may be named ε. Furthermore, function types carry an effect $[r]$. It is also a row, and gives a conservative description of all exceptions possibly raised by executing the function.

Second, we use security levels to keep track of how much information can be obtained by looking up integer values, executing functions, dereferencing memory locations, and handling exceptions. The remainder of this section describes their meaning.

Because there is only one value of type unit, the value of a unit expression yields no information whatsoever. As a result, it would be superfluous for the unit type constructor to carry a security level. Immutable tuple and record types can be dealt with similarly; see section 7.1. Thus, we break the convention set forth in a number of previous papers [8, 16] that all types be of the form $*^\ell$. We expect this feature to help reduce verbosity in practice.

The type int^ℓ describes integer expressions whose value may reflect information of security level ℓ.

Function types carry two security annotations. The external annotation ℓ represents information about the function's identity. When the function is applied, part of this information may be reflected in its result or in other aspects

$$\mathsf{int}^{\oplus} \qquad (\ominus \xrightarrow{\ominus\ [\oplus]} \oplus)^{\oplus} \qquad \odot\ \mathsf{ref}^{\oplus} \qquad \oplus\ \mathsf{exn}^{\oplus}$$

$$\{\varepsilon \mapsto \oplus\}_{\varepsilon \in \mathcal{E}} \qquad \mathsf{Pre}\ \oplus \qquad \mathsf{Abs} \leq \mathsf{Pre}\ *$$

Figure 2: Subtyping

of the function's behavior (i.e. in its effect); as a result, their security level will be made ℓ or greater. The annotation pc, found above the $\rightarrow$ symbol, tells how much information the function obtains merely by gaining control – indeed, observing that a particular function is called may allow telling which branches were previously taken. pc can be thought of as an extra parameter to the function, and indeed it is contravariant (see section 5.2). To avoid leaking this information, the function will be allowed to write into memory cells, or to raise exceptions, only at level pc or greater. This explains why the annotation pc is sometimes described as a lower bound on the level of the function's effects [8].

Reference types carry one annotation ℓ, which represents information about the reference's identity, i.e. about its address. Information about the reference's contents is found within the parameter t.

Exceptions are described by rows, within which every non-Abs entry, of the form $\varepsilon \mapsto \mathsf{Pre}\ pc$, carries an annotation pc, telling how much information will be obtained by observing (i.e. handling) the exception, if it is named ε. We follow Myers [10, 11] and associate a distinct security level with every exception name, so as to obtain better precision. Our rows are closely related to Myers' sets of path labels X, which map every exception name to either a special constant $\underline{\emptyset}$ or a security level; compare these with our alternatives Abs and $\mathsf{Pre}\ pc$. (See section 10 for further comparison with [10, 11].)

In addition to a row, exception types also carry an external annotation ℓ. It is, in fact, redundant with the row r. That is, manipulating an exception as a first-class value causes its external level ℓ to increase, leaving the row r unchanged; when the exception is later raised, every non-Abs entry in r is raised to level ℓ or greater. It would be possible to suppress the external annotation, at the cost of some extra implementation complexity. Another reasonable approach would be to restrict the language so that exceptions are no longer first-class values; this would allow us to do away with exn entirely.

The reader may notice that rows do not record the type of exception arguments, i.e. the constructor Pre has no type parameter. Indeed, as in ML, we make exceptions monomorphic by assuming given a fixed mapping *typexn* from exception names to types. This decision is useful in two ways. First, it should make function types (which include a row) much more compact. Second, it makes our subtyping relation atomic (see section 5.2), which we believe opens the way to simpler and (in practice) more efficient constraint solving techniques.

5.2 Subtyping

We equip types, rows and alternatives with a subtyping relation $\leq$, which extends the partial order $(\mathcal{L}, \leq)$. It is defined by the axioms in figure 2. The axiom $\mathsf{int}^{\oplus}$ is a compact version of the assertion $\mathsf{int}^{\ell_1} \leq \mathsf{int}^{\ell_2} \iff \ell_1 \leq \ell_2$. In other words, it states that int's parameter is covariant. The other axioms are to be understood similarly; $\oplus$, $\ominus$ and $\odot$ represent covariant, contravariant and invariant parameters, respectively. The fifth axiom extends subtyping to rows, point-wise and covariantly.

The last axiom is the only one which relates two constructors of different arities, apparently making the subtyping relation non-atomic. However, it is only superficially so. Indeed, it is possible to give a presentation of the system where the set of alternatives is merely the disjoint union $\{\mathsf{Abs}\} \cup \mathcal{L}$, causing the explicit injection Pre to disappear, because security levels become a subset of alternatives. In this presentation, subtyping is atomic [18]: alternatives form a set of atoms.

The use of subtyping in information flow control is ubiquitous [3, 4, 21, 8] and appears essential, because it allows building a *directed* view of the program's information flow graph, yielding better precision than a unification-based analysis.

5.3 Additional notation

A *polytype* s is a nonempty, upward-closed set of types. A *polytype environment* Γ is a partial mapping from program variables to polytypes. $\Gamma[x \mapsto s]$ denotes the environment which maps x to s and agrees with Γ otherwise. A *memory environment* M is a partial mapping from memory locations to types.

We define $\ell \lhd t$ (read: ℓ *guards* t) as follows:

$$\frac{\ell \leq \ell'}{\begin{array}{c}\ell \lhd \mathsf{unit} \qquad \ell \lhd \mathsf{int}^{\ell'} \qquad \ell \lhd (* \xrightarrow{*\ [*]} *)^{\ell'} \\ \ell \lhd *\ \mathsf{ref}^{\ell'} \qquad \ell \lhd *\ \mathsf{exn}^{\ell'}\end{array}}$$

The assertion $\ell \lhd t$ requires t to have security level ℓ or greater, and is used to record a potential information flow. Note that, for any given ℓ and t, there exists a supertype t' of t such that $\ell \lhd t'$ holds. Thus, the presence of $\ell \lhd t$ as a premise typically never prevents the application of a typing rule: indeed, preceding that rule with a subtyping step will satisfy the premise. One exception is E-ASSIGN, where t cannot be promoted to a supertype because it appears as an invariant argument to the ref type constructor. The predicate $\lhd$ has transitive behavior:

Lemma 5.1 *If $\ell' \leq \ell$ and $\ell \lhd t$ and $t \leq t'$ then $\ell' \lhd t'$.*

Proof. It is easy to see that $\ell \lhd t$ is equivalent to $\ell \leq \mathrm{level}(t)$ for some appropriately defined function level. The result follows. □

To every row r, we associate two security levels, defined by $\sqcup r = \sqcup\{pc \mid \mathsf{Pre}\ pc \in r\}$ and $\sqcap r = \sqcap\{pc \mid \mathsf{Pre}\ pc \in r\}$. Note that Abs entries in r do not contribute to these levels.

5.4 Typing judgements

We distinguish two forms of typing judgements: one deals with values only, the other with arbitrary expressions. Because values are normal forms, they have no side effects, so the former look quite simple:

$$\Gamma, M \vdash v : t$$

(We also write $\Gamma, M \vdash v : s$ when $\Gamma, M \vdash v : t$ holds for all $t \in s$.) On the other hand, expressions do produce side effects, so the latter are more elaborate:

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

The pc parameter again tells how much information the expression may acquire by gaining control; it is a lower bound on the level of the expression's effects. Previous works [21, 8] employ a similar parameter. The row r approximates the set of exceptions which the expression may raise.

Two extra judgement forms are employed to type stores: $M \vdash \mu$ and configurations: $\Gamma \vdash e /_i \mu : t \ [r]$.

In typing judgements, we omit Γ and M when they are empty; we sometimes omit pc and r when they are unspecified (i.e. when they could be written $*$).

Even though the security lattice $(\mathcal{L}, \leq)$ is arbitrary, it is desirable to establish a simple dichotomy between "low" and "high" security levels. Such a distinction simplifies our proofs; full generality will be recovered in section 6. In the present section, we assume H is a fixed, upward-closed subset of $\mathcal{L}$. We will view levels inside (resp. outside) H as "high" (resp. "low").

Non-interference demands that two expressions which differ only in high-level sub-terms have identical low-level behavior. To achieve this, our type system requires expressions of the form $\langle e_1 \mid e_2 \rangle$ – which we use to encode the differences between two Core ML expressions – to have high-security result and side effects. (See V-BRACKET and E-BRACKET in figure 3.) This will be our only use of H in this section.

5.5 Typing rules

We now comment on the typing rules, given in figure 3. V-UNIT and V-INT assign base types to constants. V-VOID allows typing values of the form $\langle v \mid \mathsf{void} \rangle$ or $\langle \mathsf{void} \mid v \rangle$ by pretending void has the same type as v. V-LOC and V-VAR assign types to memory locations and to variables by looking up the appropriate environment. Note that $\Gamma(x)$ is a polytype, of which V-VAR selects an arbitrary instance. As usual in type-and-effect systems, V-ABS records, on top of the $\rightarrow$ type constructor, information about a function's side effects. V-EXN associates to the exception value $\varepsilon\, v$ a row which maps the name ε to $\mathsf{Pre}\ *$ and leaves other entries unconstrained, allowing them to be Abs. V-BRACKET requires the components of a $\langle \cdot \mid \cdot \rangle$ construct to have a common type, which must have "high" security level, i.e. be guarded by some (arbitrary) element of H. V-SUB is standard.

E-VALUE allows viewing a value as an expression, and reflects the fact that values have no side effect.

E-APP governs function application. Because the effect of a function application is exactly the function's latent effect, the security level pc, which should represent a lower bound on the level of the former, must also be a lower bound on the latter's. Because a function's side effects may reveal information about its identity, their level must equal or exceed the function's own security level, namely ℓ. As a result of these remarks, the function's body must run at level $pc \sqcup \ell$. Because the function's result, too, may reveal information about its identity, we require its type to be guarded by ℓ.

E-REF and E-ASSIGN require $pc \lhd t$ to ensure that pc is indeed a lower bound on the security level of the memory cell that is written. E-ASSIGN and E-DEREF require $\ell \lhd t$ to reflect the fact that writing or reading a cell may indirectly reveal information about its identity.

E-RAISE requires $pc \leq \sqcap r$, ensuring that pc is a lower bound on the level of every non-Abs entry in the row r. Thus, any code fragment able to observe this expression's side effect must run at level pc or greater (see E-BIND, E-HANDLE and E-HANDLEALL). The security level ℓ, which reflects additional, exception-name-independent information, is dealt with similarly.

Because let only binds values, E-LET is nearly as simple as in ML. Note that v can be given a polytype s, allowing x to be used at different types within e.

In a binding construct $\mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2$, the expression e_2 observes, if it receives control, that no exception was raised by e_1. To account for this information channel, E-BIND typechecks e_2 at a security level augmented with $\sqcup r_1$, the combined level of all exceptions which e_1 can potentially raise. This is a conservative approximation, which works well in the common case where e_1 is statically known never to raise exceptions; see section 10 for details. $r_1 \sqcup r_2$ denotes the least common supertype of r_1 and r_2.

Like E-BIND, E-HANDLE typechecks e_2 at an increased security level, reflecting the fact that, by gaining control, e_2 observes that e_1 raised an exception named ε. The increment is exactly pc', the security level associated with ε in e_1's effect, so the analysis is, in this case, quite accurate. Because the result of the handle construct may also allow determining whether the handler was executed, we require $pc' \lhd t$. E-HANDLEALL is analogous; however, because the construct allows observing any exception, regardless of its name, we again use $\sqcup r_1$ as a conservative approximation of how much information is gained. Myers [10, 11] performs the same approximation.

As explained earlier, E-BRACKET requires both components of a $\langle \cdot \mid \cdot \rangle$ expression to have a common type, and demands that its side effects and its result be of "high" security level, i.e. guarded by an arbitrary $pc' \in H$. The auxiliary predicate $e\Uparrow$ holds if and only if e is of the form $E_1[\ldots E_n[\mathsf{raise}\ (\varepsilon\, v)]\ldots]$ where $n \geq 0$ and none of the E_i handles $\mathsf{raise}\ (\varepsilon\, v)$. The use of this predicate in E-BRACKET's last premise is technical; it is required for subject reduction to hold.

5.6 Subject reduction

Let us first state a few auxiliary lemmas, whose proofs are straightforward.

Lemma 5.2 (Subsumption) *$pc' \leq pc$ and $pc, \Gamma, M \vdash e : t\ [r]$ imply $pc', \Gamma, M \vdash e : t\ [r]$.*

Proof. By induction on the derivation of $pc, \Gamma, M \vdash e : t\ [r]$. By monotonicity of $\sqcup$, contravariance of $\rightarrow$ with respect to its pc parameter, rule V-SUB, lemma 5.1, and the induction hypothesis, it is easy to check that every premise remains valid when pc decreases. The result follows. □

Lemma 5.3 (Projection) *Let $i \in \{1, 2\}$. If $\Gamma, M \vdash v : t$ then $\Gamma, M \vdash \lfloor v \rfloor_i : t$. If $pc, \Gamma, M \vdash e : t\ [r]$ then $pc, \Gamma, M \vdash \lfloor e \rfloor_i : t\ [r]$.*

Proof. By induction on the input derivation. The only case of interest is that of E-BRACKET, where the expression at hand is $\langle e_1 \mid e_2 \rangle$. Then, one of the first two premises is $pc \sqcup pc', \Gamma, M \vdash e_i : t\ [r]$. Lemma 5.2 yields $pc, \Gamma, M \vdash e_i : t\ [r]$, as required. □

Lemma 5.4 (Guard) *If $\Gamma, M \vdash \langle v_1 \mid v_2 \rangle : t$ then there exists $pc' \in H$ such that $pc' \lhd t$.*

Proof. Thanks to lemma 5.1, we may assume, w.l.o.g., that the derivation of $\Gamma, M \vdash \langle v_1 \mid v_2 \rangle : t$ does not end with

Values

$$\textsc{v-Unit}\quad \Gamma, M \vdash () : \mathsf{unit}$$

$$\textsc{v-Int}\quad \Gamma, M \vdash k : \mathsf{int}^*$$

$$\textsc{v-Void}\quad \Gamma, M \vdash \mathsf{void} : *$$

$$\textsc{v-Loc}\quad \Gamma, M \vdash m : M(m)\ \mathsf{ref}^*$$

$$\textsc{v-Var}\quad \frac{t \in \Gamma(x)}{\Gamma, M \vdash x : t}$$

$$\textsc{v-Abs}\quad \frac{pc, \Gamma[x \mapsto t'][f \mapsto (t' \xrightarrow{pc\ [r]} t)^\ell], M \vdash e : t\ \ [r]}{\Gamma, M \vdash \mathsf{fix}\ f.\lambda x.e : (t' \xrightarrow{pc\ [r]} t)^\ell}$$

$$\textsc{v-Exn}\quad \frac{\Gamma, M \vdash v : \mathit{typexn}(\varepsilon)}{\Gamma, M \vdash \varepsilon\, v : (\varepsilon : \mathsf{Pre}\ *; *)\ \mathsf{exn}^*}$$

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

$$\textsc{v-Sub}\quad \frac{\Gamma, M \vdash v : t' \qquad t' \leq t}{\Gamma, M \vdash v : t}$$

Expressions

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

$$\textsc{e-App}\quad \frac{\Gamma, M \vdash v_1 : (t' \xrightarrow{pc \sqcup \ell\ [r]} t)^\ell \qquad \Gamma, M \vdash v_2 : t' \qquad \ell \lhd t}{pc, \Gamma, M \vdash v_1\, v_2 : t\ \ [r]}$$

$$\textsc{e-Ref}\quad \frac{\Gamma, M \vdash v : t \qquad pc \lhd t}{pc, \Gamma, M \vdash \mathsf{ref}\ v : t\ \mathsf{ref}^*\ \ [*]}$$

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

$$\textsc{e-Deref}\quad \frac{\Gamma, M \vdash v : t'\ \mathsf{ref}^\ell \qquad t' \leq t \qquad \ell \lhd t}{pc, \Gamma, M \vdash\ !v : t\ \ [*]}$$

$$\textsc{e-Raise}\quad \frac{\Gamma, M \vdash v : r\ \mathsf{exn}^\ell \qquad pc \sqcup \ell \leq \sqcap r}{pc, \Gamma, M \vdash \mathsf{raise}\ v : *\ \ [r]}$$

$$\textsc{e-Let}\quad \frac{\Gamma, M \vdash v : s \qquad pc, \Gamma[x \mapsto s], M \vdash e : t\ \ [r]}{pc, \Gamma, M \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e : t\ \ [r]}$$

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

$$\textsc{e-Handle}\quad \frac{pc, \Gamma, M \vdash e_1 : t\ \ [\varepsilon : \mathsf{Pre}\ pc'; r] \qquad pc \sqcup pc', \Gamma[x \mapsto \mathit{typexn}(\varepsilon)], M \vdash e_2 : t\ \ [\varepsilon : a; r] \qquad pc' \lhd t}{pc, \Gamma, M \vdash e_1\ \mathsf{handle}\ \varepsilon\, x \succ e_2 : t\ \ [\varepsilon : a; r]}$$

$$\textsc{e-HandleAll}\quad \frac{pc, \Gamma, M \vdash e_1 : t\ \ [r_1] \qquad pc \sqcup (\sqcup r_1), \Gamma[x \mapsto r_1\ \mathsf{exn}^*], M \vdash e_2 : t\ \ [r_2] \qquad (\sqcup r_1) \lhd t}{pc, \Gamma, M \vdash e_1\ \mathsf{handle}\ x \succ e_2 : t\ \ [r_2]}$$

$$\textsc{e-Bracket}\quad \frac{pc \sqcup pc', \Gamma, M \vdash e_1 : t\ \ [r] \qquad pc \sqcup pc', \Gamma, M \vdash e_2 : t\ \ [r] \qquad pc' \in H \qquad (pc' \lhd t) \vee (e_1 \Uparrow) \vee (e_2 \Uparrow)}{pc, \Gamma, M \vdash \langle e_1 \mid e_2 \rangle : t\ \ [r]}$$

$$\textsc{e-Sub}\quad \frac{pc, \Gamma, M \vdash e : t'\ \ [r'] \qquad t' \leq t \qquad r' \leq r}{pc, \Gamma, M \vdash e : t\ \ [r]}$$

Configurations

$$\textsc{Store}\quad \frac{\mathrm{dom}(M) = \mathrm{dom}(\mu) \qquad \forall m \in \mathrm{dom}(\mu) \quad M \vdash \mu(m) : M(m)}{M \vdash \mu}$$

$$\textsc{Conf}\quad \frac{pc, \Gamma, M \vdash e : t\ \ [r] \qquad M \vdash \mu}{\Gamma \vdash e \,/\, \mu : t\ \ [r]}$$

Figure 3: The type system $\textsc{mlif}_0$

an instance of V-SUB. Thus, it must end with an instance of V-BRACKET, among whose premises we find $pc' \lhd t$ and $pc' \in H$. □

Lemma 5.5 (Store access) *Let i be in $\{\bullet, 1, 2\}$. Assume $\Gamma, M \vdash v : t$ and $\Gamma, M \vdash v' : t$. Then, $\Gamma, M \vdash \mathrm{read}_i\, v : t$ holds. Moreover, if $i \in \{1, 2\}$, assume there exists some $pc' \in H$ such that $pc' \lhd t$. Then, $\Gamma, M \vdash \mathrm{new}_i\, v : t$ and $\Gamma, M \vdash \mathrm{update}_i\, v\, v' : t$ hold.*

Proof. By definition of the functions new, update and read (figure 1), by lemma 5.3, by V-VOID and V-BRACKET. □

Lemma 5.6 (Substitution) *Assume $M \vdash v : s$. Then, $\Gamma[x \mapsto s], M \vdash v' : t$ implies $\Gamma, M \vdash v'[x \Leftarrow v] : t$. Also, $pc, \Gamma[x \mapsto s], M \vdash e : t\ [r]$ implies $pc, \Gamma, M \vdash e[x \Leftarrow v] : t\ [r]$.*

Proof. By induction on the input derivation.

∘ *Case* V-VAR. If v' is x, then the premise is $t \in s$. Thus, the hypothesis $M \vdash v : s$ implies $M \vdash v : t$, and, a fortiori, $\Gamma, M \vdash v : t$. Considering $v'[x \Leftarrow v] = v$, this was the goal. If, on the other hand, v' isn't x, then the result stems from $\Gamma[x \mapsto s](v') = \Gamma(v')$ and $v'[x \Leftarrow v] = v'$.

∘ *Case* V-ABS. Then, the premise must be of the form $pc', \Gamma[x \mapsto s][y \mapsto t'][f \mapsto t_f], M \vdash e' : t''\ [r']$. Because typing judgements are stable under α-conversion, we will assume, w.l.o.g., that x, f and y are distinct. Then, $\Gamma[x \mapsto s][y \mapsto t'][f \mapsto t_f]$ coincides with $\Gamma[y \mapsto t'][f \mapsto t_f][x \mapsto s]$. We conclude by applying the induction hypothesis, followed by an instance of V-ABS.

∘ *Case* V-BRACKET. The first premise is of the form $\Gamma[x \mapsto s], M \vdash v'_1 : t$. By lemma 5.3, the hypothesis $M \vdash v : s$ implies $M \vdash \lfloor v \rfloor_1 : s$. Thus, by induction hypothesis, $\Gamma, M \vdash v'_1[x \Leftarrow \lfloor v \rfloor_1] : t$ holds. The second premise is dealt with similarly. By V-BRACKET, we obtain $\Gamma, M \vdash \langle v'_1[x \Leftarrow \lfloor v \rfloor_1] \mid v'_2[x \Leftarrow \lfloor v \rfloor_2] \rangle : t$, which, considering our definition of substitution (section 4.2), was our goal.

Other cases are either immediate or analogous to those above. □

Lemma 5.7 (Value) *$pc, \Gamma, M \vdash v : t\ [r]$ implies $\Gamma, M \vdash v : t$.*

Proof. By induction on the proof of $pc, \Gamma, M \vdash v : t\ [r]$.

∘ *Case* E-VALUE. Immediate.

∘ *Case* E-SUB. The result follows from the induction hypothesis and V-SUB.

∘ *Case* E-BRACKET. The predicate $\cdot\Uparrow$ is never true of a value, so $pc' \lhd t$ must hold. The result follows from the induction hypothesis and V-BRACKET. □

We can now state our main lemma:

Lemma 5.8 (Subject reduction) *Let $e\ /_i\ \mu \rightarrow e'\ /_i\ \mu'$. Assume $pc, M \vdash e : t\ [r]$ and $M \vdash \mu$. If $i \in \{1, 2\}$, assume $pc \in H$. Then, there exists a memory environment M', which extends M, such that $pc, M' \vdash e' : t\ [r]$ and $M' \vdash \mu'$.*

Proof. By induction on the derivation of $e\ /_i\ \mu \rightarrow e'\ /_i\ \mu'$. We assume, w.l.o.g., that the derivation of $pc, M \vdash e : t\ [r]$ does not end with an instance of E-SUB. As a result, it must end with an instance of the single syntax-directed rule that matches e's structure.

∘ *Case* (β). e is $(\mathsf{fix}\, f.\lambda x.e_0)\, v$. Let $\theta = (t' \xrightarrow{pc \sqcup \ell\ [r]} t)^\ell$. By E-APP, we have $M \vdash \mathsf{fix}\, f.\lambda x.e_0 : \theta$ and $M \vdash v : t'$. The former's derivation must end with an instance of V-ABS, followed by a number of instances of V-SUB. Because $\rightarrow$ is contravariant (resp. covariant) in its first and second (resp. third and fourth) parameters, applying lemma 5.2 and E-SUB to V-ABS's premise yields $pc, (x \mapsto t''; f \mapsto \theta'), M \vdash e_0 : t\ [r]$, for some t'' and θ' such that $t' \leq t''$ and $\theta \leq \theta'$. By V-SUB, $M \vdash v : t''$ and $M \vdash \mathsf{fix}\, f.\lambda x.e_0 : \theta'$ hold. Then, lemma 5.6 yields $pc, M \vdash e_0[x \Leftarrow v][f \Leftarrow \mathsf{fix}\, f.\lambda x.e_0] : t\ [r]$.

∘ *Case* (ref). e is $\mathsf{ref}\ v$, e' is m and μ' is $\mu \oplus [m \mapsto \mathrm{new}_i\, v]$. By E-REF, we have $M \vdash v : t'$ and $pc \lhd t'$ and $t = t'\ \mathsf{ref}^*$. By lemma 5.5, these imply $M \vdash \mathrm{new}_i\, v : t'$. Define $M' = M[m \mapsto t']$. By STORE, $M \vdash \mu$ yields $\mathrm{dom}(M) = \mathrm{dom}(\mu)$. Because $\mu \oplus [m \mapsto v]$ is defined, m isn't a member of $\mathrm{dom}(\mu)$. So, M' extends M. Because $M'(m) = t'$, V-LOC and E-VALUE yield $pc, M' \vdash e' : t\ [r]$. Lastly, $M \vdash \mu$ and $M \vdash \mathrm{new}_i\, v : t'$ entail $M' \vdash \mu'$.

∘ *Case* (assign). e is $m := v$ and e' is $()$. By E-ASSIGN, we must have $M \vdash m : t'\ \mathsf{ref}^*$ and $M \vdash v : t'$ and $pc \lhd t'$. Furthermore, t must be unit, which implies $pc, M \vdash e' : t\ [r]$. By V-LOC, V-SUB and by invariance of the ref type constructor, $M \vdash m : t'\ \mathsf{ref}^*$ implies $M(m) = t'$. Thus, $M \vdash \mu$ entails $M \vdash \mu(m) : t'$. By lemma 5.5, we have $M \vdash \mathrm{update}_i\, \mu(m)\, v : t'$, which yields $M \vdash \mu'$.

∘ *Case* (deref). e is $!\,m$. By E-DEREF, we have $M \vdash m : t'\ \mathsf{ref}^*$, where $t' \leq t$. As above, this entails $M \vdash \mu(m) : t'$. By lemma 5.5, $M \vdash \mathrm{read}_i\, \mu(m) : t'$ follows. Conclude with V-SUB and E-VALUE.

∘ *Case* (let). By E-LET and lemma 5.6.

∘ *Case* (bind). e is $\mathsf{bind}\ x = v\ \mathsf{in}\ e_2$ and e' is $e_2[x \Leftarrow v]$. By E-BIND, we have $pc, M \vdash v : t'\ [r_1]$ and $pc \sqcup (\sqcup r_1), (x \mapsto t'), M \vdash e_2 : t\ [r_2]$, where $r_2 \leq r$. By lemma 5.7, the former implies $M \vdash v : t'$. By lemma 5.2, the latter implies $pc, (x \mapsto t'), M \vdash e_2 : t\ [r_2]$. By lemma 5.6 and E-SUB, we obtain $pc, M \vdash e_2[x \Leftarrow v] : t\ [r]$.

∘ *Case* (handle). e is $\mathsf{raise}\ (\varepsilon\, v)\ \mathsf{handle}\ \varepsilon\, x \succ e_2$ and e' is $e_2[x \Leftarrow v]$. By E-HANDLE, we have $pc, M \vdash \mathsf{raise}\ (\varepsilon\, v) : t\ [\varepsilon : \mathsf{Pre}\ pc'; r']$ and $pc \sqcup pc', (x \mapsto \mathit{typexn}(\varepsilon)), M \vdash e_2 : t\ [\varepsilon : a; r']$, where r equals $(\varepsilon : a; r')$. By E-SUB, E-RAISE, V-SUB and V-EXN, the former yields $M \vdash v : \mathit{typexn}(\varepsilon)$. By lemmas 5.6 and 5.2, this yields $pc, M \vdash e_2[x \Leftarrow v] : t\ [r]$.

∘ *Case* (handle-all). e is $\mathsf{raise}\ (\varepsilon\, v)\ \mathsf{handle}\ x \succ e_2$ and e' is $e_2[x \Leftarrow \varepsilon\, v]$. By E-HANDLEALL, $pc, M \vdash \mathsf{raise}\ (\varepsilon\, v) : t\ [r_1]$ and $pc \sqcup (\sqcup r_1), (x \mapsto r_1\ \mathsf{exn}^\ell), M \vdash e_2 : t\ [r]$ hold. By E-SUB, E-RAISE, V-SUB and V-EXN, and by covariance of the exn type constructor, the former yields $M \vdash \varepsilon\, v : r_1\ \mathsf{exn}^\ell$. Lemmas 5.6 and 5.2 yield $pc, M \vdash e_2[x \Leftarrow \varepsilon\, v] : t\ [r]$.

∘ *Case* (throw-context). e is $E[o]$ and e' is o. Several sub-cases arise.

Sub-case $E = \mathsf{bind}\ x = [\ ]\ \mathsf{in}\ e_2$. By E-BIND, we must have $pc, M \vdash o : t'\ [r_1]$, where $r_1 \leq r$. Because o must be of the form $\mathsf{raise}\ (\varepsilon\, v)$ or $\langle \mathsf{raise}\ (\varepsilon_1\, v_1) \mid \mathsf{raise}\ (\varepsilon_2\, v_2) \rangle$, this judgement must be a consequence of E-RAISE, E-BRACKET and E-SUB. A derivation of identical shape can be built to establish $pc, M \vdash o : t\ [r_1]$. (In the case of E-BRACKET, the fourth premise is satisfied, though its first disjunct may be false, because the other two hold.) The result follows by E-SUB.

Sub-case $E = [\ ]\ \mathsf{handle}\ \varepsilon\, x \succ e_2$. By E-HANDLE, we have $pc, M \vdash o : t\ [\varepsilon : \mathsf{Pre}\ *; r']$. o must be of the form v or $\mathsf{raise}\ (\varepsilon'\, v)$ or $\langle v_1 \mid \mathsf{raise}\ (\varepsilon_2\, v_2) \rangle$ or $\langle \mathsf{raise}\ (\varepsilon_1\, v_1) \mid v_2 \rangle$ or $\langle \mathsf{raise}\ (\varepsilon_1\, v_1) \mid \mathsf{raise}\ (\varepsilon_2\, v_2) \rangle$, where ε', ε_1 and ε_2 are distinct from ε. As a result, a derivation of identical shape can be built to establish $pc, M \vdash o : t\ [\varepsilon : a; r']$, that is, $pc, M \vdash o : t\ [r]$.

Sub-case $E = [\ ]$ handle $x \succ e_2$. By E-HANDLEALL, $pc, M \vdash o : t\ [r_1]$ holds. Because o must be a value, a derivation of identical shape yields $pc, M \vdash o : t\ [r]$.

∘ *Case* (lift-app). e is $\langle v_1 \mid v_2 \rangle\, v$. Let $\theta = (t' \xrightarrow{pc \sqcup \ell\ [r]} t)^\ell$. E-APP's premises are $M \vdash \langle v_1 \mid v_2 \rangle : \theta$ and $M \vdash v : t'$ and $\ell \lhd t$. Lemma 5.3 yields $M \vdash v_i : \theta$ and $M \vdash \lfloor v \rfloor_i : t'$, for $i \in \{1, 2\}$. Then, E-APP yields $pc \sqcup \ell, M \vdash v_i\, \lfloor v \rfloor_i : t\ [r]$. Furthermore, applying lemma 5.4 to the first premise above and recalling that H is upward-closed yields $\ell \in H$. Because $\ell \lhd t$, E-BRACKET is applicable and yields $pc, M \vdash e' : t\ [r]$.

∘ *Case* (lift-assign). e is $\langle v_1 \mid v_2 \rangle := v$. E-ASSIGN's premises are $M \vdash \langle v_1 \mid v_2 \rangle : t'\ \mathsf{ref}^\ell$ and $M \vdash v : t'$ and $pc \sqcup \ell \lhd t'$. As above, applying lemma 5.3 and building new instances of E-ASSIGN, we obtain $pc \sqcup \ell, M \vdash v_i := \lfloor v \rfloor_i : t\ [r]$, for $i \in \{1, 2\}$. Similarly, lemma 5.4 allows establishing $\ell \in H$. The result follows by E-BRACKET.

∘ *Case* (lift-deref). e is $!\,\langle v_1 \mid v_2 \rangle$. E-DEREF's premises are $M \vdash \langle v_1 \mid v_2 \rangle : t'\ \mathsf{ref}^\ell$ and $t' \leq t$ and $\ell \lhd t$. As above, applying lemma 5.3 and building new instances of E-DEREF, we obtain $pc \sqcup \ell, M \vdash !\, v_i : t\ [r]$, for $i \in \{1, 2\}$. Similarly, lemma 5.4 yields $\ell \in H$. Lastly, by E-BRACKET, we obtain $pc, M \vdash \langle !\, v_1 \mid !\, v_2 \rangle : t\ [r]$.

∘ *Case* (lift-raise). e is raise $\langle \varepsilon_1\, v_1 \mid \varepsilon_2\, v_2 \rangle$. E-RAISE's premises are $M \vdash \langle \varepsilon_1\, v_1 \mid \varepsilon_2\, v_2 \rangle : r\ \mathsf{exn}^\ell$ and $pc \sqcup \ell \leq \sqcap r$. As above, lemma 5.3 and E-RAISE yield $pc \sqcup \ell, M \vdash$ raise $(\varepsilon_i\, v_i) : t\ [r]$, for $i \in \{1, 2\}$, while lemma 5.4 yields $\ell \in H$. The result follows by E-BRACKET, whose fourth premise is satisfied, though $\ell \lhd t$ may be false, because the other disjuncts hold.

∘ *Case* (lift-context). e is $E[\langle o_1 \mid o_2 \rangle]$. If E is a bind context, then, because e cannot be reduced by (bind), $\langle o_1 \mid o_2 \rangle$ cannot be a value. If, on the other hand, E is a handle context, then, because (throw-context) isn't applicable, E must handle o_1 or o_2. In either case, we conclude that o_i is of the form raise $(\varepsilon\, v)$, for some $i \in \{1, 2\}$. Now, e's typing derivation must end with an instance of E-BIND, E-HANDLE or E-HANDLEALL, whose first premise is of the form $pc, M \vdash \langle o_1 \mid o_2 \rangle : t'\ [r_1]$. Because $\langle o_1 \mid o_2 \rangle$ isn't a value, this must be a consequence of E-SUB and E-BRACKET, which yields $pc \sqcup \ell, M \vdash o_i : t'\ [r_1]$, for some $\ell \in H$ and for $i \in \{1, 2\}$. By E-RAISE, V-SUB and V-EXN, this implies Pre $\ell \leq r_1(\varepsilon)$ and $\ell \leq \sqcup r_1$; thus, the security assumption in E-BIND, E-HANDLE or E-HANDLEALL's second premise is greater than or equal to ℓ. As a result, by applying lemma 5.3 to that premise, then building new instances of E-BIND, E-HANDLE or E-HANDLEALL, we obtain $pc \sqcup \ell, M \vdash \lfloor E \rfloor_i[o_i] : t\ [r]$, for $i \in \{1, 2\}$. There remains to apply E-BRACKET. If E is a bind context, then $\lfloor E \rfloor_i[o_i]\Uparrow$ holds for some $i \in \{1, 2\}$; if, on the other hand, E is a handle context, then $\ell \lhd t$ holds, according to E-HANDLE or E-HANDLEALL's third premise. In either case, E-BRACKET's fourth premise holds.

∘ *Case* (bracket). e is $\langle e_1 \mid e_2 \rangle$ and e' is $\langle e_1' \mid e_2' \rangle$. We have $e_i /_i\, \mu \rightarrow e_i' /_i\, \mu'$ and $e_j = e_j'$, where $\{i, j\} = \{1, 2\}$. Because $\langle e_1 \mid e_2 \rangle$ isn't a value, its typing derivation must end with an instance of E-BRACKET, whose first two premises are $pc \sqcup pc', M \vdash e_i : t\ [r]$ and $pc \sqcup pc', M \vdash e_j : t\ [r]$. Because $pc' \in H$, the induction hypothesis is applicable, yielding a memory environment M', which extends M, such that $pc \sqcup pc', M' \vdash e_i' : t\ [r]$ and $M' \vdash \mu'$. Because M' extends M, $pc \sqcup pc', M' \vdash e_j : t\ [r]$ holds as well. The result follows by E-BRACKET.

∘ *Case* (context). e is $E[e_0]$ and e' is $E[e_0']$, where $e_0 /_i\, \mu \rightarrow e_0' /_i\, \mu'$. Applying the induction hypothesis to E-BIND, E-HANDLE or E-HANDLEALL's first premise yields a version of it with M and e_0 replaced with M' and e_0', where M' extends M and $M' \vdash \mu'$ holds. Because M extends M', the second premise remains valid when the former is replaced with the latter. Build a new instance of E-BIND, E-HANDLE or E-HANDLEALL to conclude. □

The previous lemma entails the following, more abstract statement:

Theorem 5.1 (Subject reduction) *If* $\vdash e / \mu : t\ [r]$ *and* $e / \mu \rightarrow e' / \mu'$ *then* $\vdash e' / \mu' : t\ [r]$.

Proof. By CONF and lemma 5.8. □

We do not establish *progress* (i.e. "no well-typed configuration is stuck"), even though it does hold, because it is unrelated to our concerns.

5.7 On evaluation order

As explained in section 3, our restricted syntax is fully explicit about evaluation order. In practice, it is possible to use more permissive syntax, provided some evaluation strategy is fixed. For instance, if left-to-right evaluation order is chosen, then $e_1\, e_2$ (the application of an expression to another expression) is syntactic sugar for bind $x_1 = e_1$ in bind $x_2 = e_2$ in $x_1\, x_2$. This gives rise to the following derived typing rule:

$$\frac{\begin{array}{c} pc, \Gamma, M \vdash e_1 : (t' \xrightarrow{pc \sqcup \ell \sqcup (\sqcup r_1) \sqcup (\sqcup r_2)\ [r]} t)^\ell\ [r_1] \\ pc \sqcup (\sqcup r_1), \Gamma, M \vdash e_2 : t'\ [r_2] \qquad \ell \lhd t \end{array}}{pc, \Gamma, M \vdash e_1\, e_2 : t\ [r \sqcup r_1 \sqcup r_2]}$$

Conversely, under right-to-left evaluation order, $e_1\, e_2$ is encoded as bind $x_2 = e_2$ in bind $x_1 = e_1$ in $x_1\, x_2$, yielding a different derived rule:

$$\frac{\begin{array}{c} pc \sqcup (\sqcup r_2), \Gamma, M \vdash e_1 : (t' \xrightarrow{pc \sqcup \ell \sqcup (\sqcup r_1) \sqcup (\sqcup r_2)\ [r]} t)^\ell\ [r_1] \\ pc, \Gamma, M \vdash e_2 : t'\ [r_2] \qquad \ell \lhd t \end{array}}{pc, \Gamma, M \vdash e_1\, e_2 : t\ [r \sqcup r_1 \sqcup r_2]}$$

In either case, the second expression to be evaluated is typechecked at an increased security level, reflecting the fact that, by receiving control, it is able to observe that the expression which was executed first terminated normally.

Caml-Light [9] does not specify its evaluation order. It is possible to give a conservative typing rule which is safe with respect to both left-to-right and right-to-left evaluation orders. Such a rule typechecks e_i under $pc \sqcup (\sqcup r_j)$, for $\{i, j\} = \{1, 2\}$. Because exceptions are annotated with the value of pc at the point where they are raised, and because pc can only increase within sub-expressions, this typically entails $\sqcup r_j \leq \sqcap r_i$. Furthermore, for every row r with at least one non-Abs entry, $\sqcap r \leq \sqcup r$ holds. As a result, if e_i is liable to raise some exception, then all exceptions in r_j must have the same security level. Thus, under-specifying the evaluation order causes an important loss of precision in our analysis. Caml-Light's current implementation uses a right-to-left evaluation strategy; for our purposes, this should be made part of its specification.

6 Non-interference

From here on, the set H is no longer fixed. We introduce it explicitly when needed, writing $\vdash_H$ instead of $\vdash$ in Core ML^2 typing judgements. (This is not necessary for those judgements which involve plain Core ML expressions, because H is used only in V-BRACKET and E-BRACKET.) We write $e \rightarrow^\star o$ if there exists a store μ such that $e / \varnothing \rightarrow^\star o / \mu$, where $\varnothing$ is the empty store.

Our type system keeps track of $\langle \cdot \mid \cdot \rangle$ constructs by assigning them "high" security levels (i.e. levels in H). By subject reduction, any expression which may evaluate to such a construct must also carry a "high" annotation. Conversely, no expression with a "low" annotation can evaluate to such a construct, as stated, in the particular case of integers, by the following lemma:

Lemma 6.1 *Let H be an upward-closed subset of $\mathcal{L}$. Let $\ell \notin H$. If $\vdash_H e : \mathsf{int}^\ell$ and $e \rightarrow^\star v$ then $\lfloor v \rfloor_1 = \lfloor v \rfloor_2$.*

Proof. By theorem 5.1 and CONF, there exists a memory environment M such that $M \vdash_H v : \mathsf{int}^\ell \ [*]$ holds. A value of type int^* must be of the form k or $\langle k_1 \mid k_2 \rangle$. If the latter, then, by V-BRACKET or E-BRACKET, there exists $pc' \in H$ such that $pc' \leq \ell$, which implies $\ell \in H$, a contradiction. Thus, we must have $v = k = \lfloor v \rfloor_1 = \lfloor v \rfloor_2$. □

We can now use the correspondence between Core ML and Core ML^2 developed in section 4.3 to reformulate this result in a Core ML setting:

Theorem 6.1 (Non-interference) *Choose $\ell, h \in \mathcal{L}$ such that $h \not\leq \ell$. Let $h \lhd t$. Assume $(x \mapsto t) \vdash e : \mathsf{int}^\ell$, where e is a Core ML expression. If $\vdash v_i : t$ and $e[x \Leftarrow v_i] \rightarrow^\star v'_i$, for $i \in \{1, 2\}$, then $v'_1 = v'_2$.*

Proof. Let $H = \uparrow\{h\}$. Define $v = \langle v_1 \mid v_2 \rangle$. By V-BRACKET, $\vdash_H v : t$ holds. Lemma 5.6 yields $\vdash_H e[x \Leftarrow v] : \mathsf{int}^\ell$. Now, $\lfloor e[x \Leftarrow v] \rfloor_i$ is $e[x \Leftarrow v_i]$, which, by hypothesis, reduces to v'_i. According to lemma 4.5, there exists an outcome o such that $e[x \Leftarrow v] \rightarrow^\star o$ and, for $i \in \{1, 2\}$, $\lfloor o \rfloor_i = v'_i$. Because of the latter, o must be a value. Lastly, $h \not\leq \ell$ yields $\ell \notin H$. The result follows by lemma 6.1. □

In words, h and ℓ are security levels such that information flow from h to ℓ is disallowed by the security lattice. Assuming the hole x has a "high"-level type t, the expression e can be given the "low"-level type int^ℓ. Then, no matter which value (of type t) is placed in the hole, e will compute the same value (that is, if it does produce a value at all).

7 Extensions

In this section, we describe a number of language extensions. Some are standard programming facilities which we have left out so far, namely products, sums, and primitive operations. Others are new language constructs which capture common idioms, so as to make them more amenable to analysis. We omit all proofs in this section; they can be found in [17].

7.1 Products and sums

Extending our system with products and sums is straightforward. We extend values and expressions with standard constructs:

$$
\begin{array}{lcll}
v & ::= & \ldots \mid (v,\ v) \mid \mathsf{inj}_j\ v & j \in \{1,2\} \\
e & ::= & \ldots \mid \mathsf{proj}_j\ v \mid v\ \mathsf{case}\ v\ v & j \in \{1,2\}
\end{array}
$$

The semantics of Core ML^2 is extended with the reduction rules given in figure 4. Rules (proj) and (case) are standard. (lift-proj) and (lift-case) handle the situation where the desired structure is found under a $\langle \cdot \mid \cdot \rangle$ construct; the brackets are then lifted up, as usual, causing some sub-terms to be duplicated. The grammar of types is extended as follows:

$$t \ ::= \ \ldots \mid t \times t \mid (t + t)^\ell$$

Our treatment of sums is similar to that of [8]. $\ell \lhd (* + *)^{\ell'}$ is, by definition, equivalent to $\ell \leq \ell'$. Products carry no security annotation because, in the absence of a physical equality operator, all of the information carried by a tuple is in fact carried by its components. To reflect this, we define $\ell \lhd t_1 \times t_2$ as $\ell \lhd t_1 \wedge \ell \lhd t_2$. The typing rules for products and sums are given in figure 5. In V-INJ, $(t_1 +^j t_2)^\ell$ stands for $(t_j + t_i)^\ell$, where $\{i, j\} = \{1, 2\}$.

Our treatment of products is slightly innovative, and has implications on constraint solving. Indeed, if every type carried a security annotation, as in previous works [8, 1, 16], then $\ell \lhd *^m$ would be syntactic sugar for $\ell \leq m$. Because it is not the case here, constraints involving $\lhd$ must receive special treatment by the constraint solver (see section 8.4).

7.2 Primitive operations

Practical programming languages usually provide many primitive operations, such as integer arithmetic operators. Some languages, such as Caml-Light [9], provide generic (i.e. polymorphic) comparison, hashing or marshalling functions. In the following, we present a way of assigning types to such primitive operations, without knowledge of their semantics, i.e. by considering them as "black boxes" which potentially use all of the information content of their arguments.

Semantics Assuming given a set $\mathcal{F}$ of primitive operations f, we extend the syntax of expressions as follows:

$$e \ ::= \ \ldots \mid f\, v$$

(We only consider unary operations; multiple arguments must be passed in a tuple.) The semantics of every primitive operation f is a partial function $[\![f]\!]$ which maps closed Core ML configurations $v\ /\ \mu$ to closed Core ML outcomes. Let $|v\ /\ \mu|$ denote the configuration obtained from $v\ /\ \mu$ by removing all bindings in μ which are not accessible through v. The semantics of Core ML^2 is extended as follows:

$$
\begin{array}{rcll}
f\, v /_i\, \mu & \rightarrow & [\![f]\!](|v\ /\ \mathrm{read}_i\ \mu|)\ /_i\ \mu & \text{(prim)} \\
f\, v\ /\ \mu & \rightarrow & \langle f\ \lfloor v \rfloor_1 \mid f\ \lfloor v \rfloor_2 \rangle\ /\ \mu & \text{(lift-prim)} \\
& & \text{if (prim) isn't applicable} &
\end{array}
$$

Rule (prim) gives the basic semantics of f. It uses the auxiliary function read_i to access the store; compare to (deref). Its use of $|\cdot|$ models the fact that the primitive operation can access the store only through v. The operation cannot affect the store; it may, however, raise an exception, since $[\![f]\!]$ ranges over outcomes, rather than values. Rule (lift-prim) must be applied whenever the configuration $|v\ /\ \mu|$ contains at least one $\langle \cdot \mid \cdot \rangle$ constructor; indeed, $[\![f]\!]$ is defined on Core ML configurations only. In that case, we lift all brackets to the toplevel. This is quite crude, but good enough given our intended typing.

Basic reductions	$\mathsf{proj}_j\ (v_1,\ v_2)\ /_i\ \mu$	$\rightarrow$	$v_j\ /_i\ \mu$	(proj)
	$(\mathsf{inj}_j\ v)\ \mathsf{case}\ v_1\ v_2\ /_i\ \mu$	$\rightarrow$	$v_j\ v\ /_i\ \mu$	(case)
Lifting	$\mathsf{proj}_j\ \langle v_1 \mid v_2\rangle\ /\ \mu$	$\rightarrow$	$\langle \mathsf{proj}_j\ v_1 \mid \mathsf{proj}_j\ v_2\rangle\ /\ \mu$	(lift-proj)
	$\langle v_1 \mid v_2\rangle\ \mathsf{case}\ v_1'\ v_2'\ /\ \mu$	$\rightarrow$	$\langle v_1\ \mathsf{case}\ \lfloor v_1'\rfloor_1\ \lfloor v_2'\rfloor_1 \mid v_2\ \mathsf{case}\ \lfloor v_1'\rfloor_2\ \lfloor v_2'\rfloor_2\rangle\ /\ \mu$	(lift-case)

Figure 4: Semantics of products and sums

$$\textsc{v-Pair}\quad \frac{\Gamma, M \vdash v_1 : t_1 \qquad \Gamma, M \vdash v_2 : t_2}{\Gamma, M \vdash (v_1,\ v_2) : t_1 \times t_2} \qquad \textsc{v-Inj}\quad \frac{\Gamma, M \vdash v : t}{\Gamma, M \vdash \mathsf{inj}_j\ v : (t +^j *)^*} \qquad \textsc{e-Proj}\quad \frac{\Gamma, M \vdash v : t_1 \times t_2}{*, \Gamma, M \vdash \mathsf{proj}_j\ v : t_j\ \ [*]}$$

$$\textsc{e-Match}\quad \frac{\Gamma, M \vdash v : (t_1 + t_2)^\ell \qquad \forall j \in \{1,2\} \quad \Gamma, M \vdash v_j : (t_j \xrightarrow{pc \sqcup \ell_j \sqcup \ell\ [r]} t)^{\ell_j} \qquad \ell \sqcup \ell_1 \sqcup \ell_2 \lhd t}{pc, \Gamma, M \vdash v\ \mathsf{case}\ v_1\ v_2 : t\ \ [r]}$$

Figure 5: Typing products and sums

$$\mathsf{unit} \blacktriangleleft \ell \qquad \frac{\ell' \leq \ell}{\mathsf{int}^{\ell'} \blacktriangleleft \ell} \qquad \frac{t_1 \blacktriangleleft \ell \qquad t_2 \blacktriangleleft \ell}{t_1 \times t_2 \blacktriangleleft \ell}$$

$$\frac{\ell' \leq \ell \qquad t_1 \blacktriangleleft \ell \qquad t_2 \blacktriangleleft \ell}{(t_1 + t_2)^{\ell'} \blacktriangleleft \ell} \qquad \frac{t \blacktriangleleft \ell \qquad \ell' \leq \ell}{t\ \mathsf{ref}^{\ell'} \blacktriangleleft \ell}$$

Figure 6: Collecting security annotations

Typing In the following, $\bar{r}$ denotes a row ranging over $\{\mathsf{Abs}, \mathsf{Pre}\}$. We write $\bar{r} \cdot pc$ for the row defined as follows: $\bar{r} \cdot pc(\varepsilon)$ equals $\mathsf{Pre}\ pc$ if $\bar{r}(\varepsilon)$ is Pre; it equals Abs otherwise.

The typing of primitive operations, like their semantics, is defined in two steps. First, we assume given, for every $f \in \mathcal{F}$, a set $\mathit{typeof}(f)$ such that, for every $(t', t, \bar{r}) \in \mathit{typeof}(f)$, $M \vdash v : t'$ and $M \vdash \mu$ imply $pc, M \vdash [\![f]\!](|v\ /\ \mu|) : t\ \ [\bar{r} \cdot pc]$. This amounts to assuming subject reduction for (prim); so far, no security concerns need be taken into account.

Then, to enforce security, we define a two-place predicate $\blacktriangleleft$, whose arguments are a type and a security level (figure 6). In short, $t \blacktriangleleft \ell$ requires *all* of the security annotations which appear in t and its sub-terms to be less than (or equal to) ℓ. It also requires t to have no function or exception types in its sub-terms. (Functions are not valid arguments to the polymorphic comparison operators; exceptions must be ruled out because exn is, in practice, an extensible type, i.e. the mapping *typexn* is never fully known.) The predicate $\blacktriangleleft$ enjoys the following property:

Lemma 7.1 *Assume* $\vdash_H v\ /\ \mu : t\ \ [*]$ *and* $t \blacktriangleleft \ell$. *If* $|v\ /\ \mu|$ *isn't a Core ML configuration, then* $\ell \in H$.

We give the following typing rule for applications of primitive operations:

$$\textsc{e-Primitive}\quad \frac{(t', t, \bar{r}) \in \mathit{typeof}(f) \qquad \Gamma, M \vdash v : t' \qquad t' \blacktriangleleft \ell \qquad \ell \lhd t}{pc, \Gamma, M \vdash f\ v : t\ \ [\bar{r} \cdot (pc \sqcup \ell)]}$$

This is quite crude, since we require the security level of the result type t to dominate all those which appear in the argument type t'. However, as long as nothing is known about $[\![f]\!]$, no better approximation can be given; the outcome may actually depend on any part of f's argument.

Non-interference We now check that the new reduction rules satisfy subject reduction under the extended type system.

∘ *Case* (prim). By Conf and e-Primitive, we have $(t', t, \bar{r}) \in \mathit{typeof}(f)$ and $M \vdash v : t'$ and $M \vdash \mu$. According to our assumption concerning $\mathit{typeof}(\cdot)$, this implies $pc, M \vdash [\![f]\!](|v\ /\ \mu|) : t\ \ [\bar{r} \cdot pc]$. The result follows by e-Sub.

∘ *Case* (lift-prim). Conf and e-Primitive's premises allow applying lemma 7.1, yielding $\ell \in H$. Applying lemma 5.3 and building a new instance of e-Primitive, we get $pc \sqcup \ell, M \vdash f\ \lfloor v\rfloor_i : t\ \ [\bar{r} \cdot (pc \sqcup \ell)]$ for $i \in \{1,2\}$. Recalling $\ell \lhd t$, we conclude with e-Bracket.

Applications Let us now illustrate the use of this general mechanism.

The treatment of binary integer arithmetic operations is quite simple, because they are monomorphic: they map pairs of integers to integers. This rule effectively makes the result's security level the union of the arguments' levels:

$$\frac{\Gamma, M \vdash v_1 : \mathsf{int}^\ell \qquad \Gamma, M \vdash v_2 : \mathsf{int}^\ell}{*, \Gamma, M \vdash v_1 \star v_2 : \mathsf{int}^\ell\ \ [*]} \quad \star \in \{+, -, \times, \ldots\}$$

The treatment of the generic (i.e. polymorphic) comparison operators is more interesting.

$$\frac{\Gamma, M \vdash v_1 : t \qquad \Gamma, M \vdash v_2 : t \qquad t \blacktriangleleft \ell}{*, \Gamma, M \vdash v_1 \star v_2 : \mathsf{bool}^\ell\ \ [*]} \quad \star \in \{=, \leq, \geq, \ldots\}$$

(The type bool^ℓ can be defined as $(\mathsf{unit} + \mathsf{unit})^\ell$ or added as a primitive type.) Because these operators traverse data structures recursively, the result of a comparison may reveal information about any sub-term. The premise $t \blacktriangleleft \ell$ reflects this by requiring ℓ to dominate all security annotations which appear in t.

Generic hashing and marshalling operations can be dealt with similarly:

$$\frac{\Gamma, M \vdash v : t \qquad t \blacktriangleleft \ell}{*, \Gamma, M \vdash \mathsf{hash}\, v : \mathsf{int}^\ell \; [*]} \qquad \frac{\Gamma, M \vdash v : t \qquad t \blacktriangleleft \ell}{*, \Gamma, M \vdash \mathsf{marshal}\, v : \mathsf{int}^\ell \; [*]}$$

By contrast, in Myers' Java-based framework [10, 11], hashing is done by having every class override the standard hashCode method, which is declared in class Object with signature int{this} hashCode (). A re-implementation of hashCode by a sub-class of Object must also satisfy this signature. As a result, it may only rely on fields labelled this. The parametric class Vector[L], for instance, must compute hash codes in a way that does not depend upon the vector's length or contents, because their label is L. Of course, this severely limits hashCode's usefulness.

7.3 Common idioms

Because our type system is quite conservative, some common programming idioms deserve special treatment, even though they are already expressible in the language.

For instance, consider the expression form e_1 finally e_2, akin to Lisp's unwind-protect and Java's try-finally constructs. Such an expression could be viewed as syntactic sugar for bind $x = (e_1$ handle $y \succ e_2$; raise $y)$ in e_2; x. However, by duplicating e_2, this encoding prevents the type-checker from discovering that e_2 is executed always, i.e. regardless of e_1's behavior. As a result, e_2 is typechecked under an increased security assumption pc. Zdancewic and Myers [25] show how ordered linear continuations provide a general solution to this problem. In our case, it is simpler to make e_1 finally e_2 a primitive construct, whose typing rule is given in figure 7.

Following Myers [10, 11], we typecheck e_1 and e_2 at a common pc. However, we add the premise $\sqcup r_2 \leq \sqcap r_1$, which reflects that, by observing an exception thrown by e_1, one may deduce that e_2 terminated normally. Its absence in Myers' work is a flaw. Myers' typing rule in fact exhibits a second flaw: its overall effect should be $X_1 \oplus X_2$, rather than $X_1[\underline{\mathsf{n}} := \underline{\emptyset}] \oplus X_2$, because normal termination of the whole statement implies normal termination of e_1. This fact is taken into account in our typing rule, even though we do not explicitly associate a security level to normal termination; see section 10. Both flaws in Myers' framework were uncovered by our formal approach [Andrew C. Myers, personal communication, June 2001].

Another common idiom which seems to require special treatment is the one which consists in anonymously handling an exception, then raising it again, to be handled further up the call chain. This is typically written e_1 handle $x \succ (e_2$; raise $x)$. In our type system, the handler e_2; raise x is typechecked at a security level increased by $\sqcup r_1$, where the row r_1 describes the exception x. Then, the second premise of E-RAISE requires $\sqcup r_1 \leq \sqcap r_1$, i.e. the security levels associated with all exception names in r_1 must be conflated, leading to a loss of precision. If, on the other hand, we introduce a new expression form e_1 handle $x \succ e_2$ reraise with the same meaning, then we can safely give it a more precise type; see figure 7.

Non-interference The syntax of evaluation contexts and the semantics of Core ML^2 are extended as described in figure 8. (Making new evaluation contexts and new sequencing rules available effectively extends (throw-context),

E-FINALLY

$$\frac{pc, \Gamma, M \vdash e_1 : t \; [r_1] \qquad pc, \Gamma, M \vdash e_2 : * \; [r_2] \qquad \sqcup r_2 \leq \sqcap r_1}{pc, \Gamma, M \vdash e_1 \; \mathsf{finally} \; e_2 : t \; [r_1 \sqcup r_2]}$$

E-RERAISE

$$\frac{pc, \Gamma, M \vdash e_1 : t \; [r_1] \qquad pc \sqcup (\sqcup r_1), \Gamma[x \mapsto r_1 \; \mathsf{exn}^*], M \vdash e_2 : * \; [r_2] \qquad \sqcup r_2 \leq \sqcap r_1}{pc, \Gamma, M \vdash e_1 \; \mathsf{handle} \; x \succ e_2 \; \mathsf{reraise} : t \; [r_1 \sqcup r_2]}$$

Figure 7: Typing finally and reraise

(lift-context) and (context) as well.) Sequential composition $e_1; e_2$ is defined as syntactic sugar for bind $x = e_1$ in e_2, where x doesn't appear free in e_2.

We begin by establishing the following simple lemma:

Lemma 7.2 *$pc, M \vdash o : t \; [r]$ and $pc' \leq \sqcap r$ imply $pc \sqcup pc', M \vdash o : t \; [r]$.*

Proof. If o is a value, the result is a consequence of lemma 5.7 and E-VALUE. If o is of the form raise $(\varepsilon\, v)$, then (discarding, w.l.o.g., any instances of E-SUB) the type derivation ends with an instance of E-RAISE, whose premises remain valid if pc is replaced with $pc \sqcup pc'$, thanks to the hypothesis $pc' \leq \sqcap r$. If o is $\langle o_1 \mid o_2 \rangle$, the result follows by E-BRACKET and the induction hypothesis. □

We now check that the new reduction rules satisfy subject reduction under the extended type system.

∘ *Case* (finally). e is o finally e_2 and e' is $(e_2; \; o)$. By E-FINALLY, we have $pc, M \vdash o : t \; [r_1]$ and $pc, M \vdash e_2 : * \; [r_2]$ where $\sqcup r_2 \leq \sqcap r_1$. By lemma 7.2, the former yields $pc \sqcup (\sqcup r_2), M \vdash o : t \; [r_1]$. By E-BIND, we obtain $pc, M \vdash e' : t \; [r_1 \sqcup r_2]$.

∘ *Case* (reraise). e is raise $(\varepsilon\, v)$ handle $x \succ e_2$ reraise and e' is $(e_2[x \Leftarrow \varepsilon\, v]$; raise $(\varepsilon\, v))$. By E-RERAISE and lemma 5.2, we have $pc, M \vdash$ raise $(\varepsilon\, v) : t \; [r_1]$ and $pc, (x \mapsto r_1 \; \mathsf{exn}^\ell), M \vdash e_2 : * \; [r_2]$ where $\sqcup r_2 \leq \sqcap r_1$. By lemma 7.2, the former yields $pc \sqcup (\sqcup r_2), M \vdash$ raise $(\varepsilon\, v) : t \; [r_1]$. By E-SUB, E-RAISE, V-SUB and V-EXN, it also yields $M \vdash \varepsilon\, v : r_1 \; \mathsf{exn}^\ell$. By lemma 5.6, the latter then yields $pc, M \vdash e_2[x \Leftarrow \varepsilon\, v] : * \; [r_2]$. Then, by E-BIND, $pc, M \vdash e' : t \; [r_1 \sqcup r_2]$ holds.

∘ *Case* (throw-context), sub-case $E = [\;]$ handle $x \succ e_2$ reraise. e is $E[o]$ and e' is o. By E-RERAISE, $pc, M \vdash o : t \; [r_1]$ holds. By E-SUB, so does $pc, M \vdash o : t \; [r_1 \sqcup r_2]$.

∘ *Case* (lift-context), (context). The descriptions in the proof of lemma 5.8 still apply.

8 A constraint-based type system

We now give a more algorithmic presentation of our type system, called MLIF. It differs from MLIF_0 mainly by introducing type variables, constraints, and using them to form universally quantified, constrained type schemes, in the style of HM(X) [12]. Like HM(X), it has principal types and decidable type inference. Because the construction is not the central topic of this paper, we will describe it only succinctly; the reader is referred to [12, 15] for more details.

$$E \quad ::= \quad \dots \mid [\,]\ \mathsf{finally}\ e \mid [\,]\ \mathsf{handle}\ x \succ e\ \mathsf{reraise}$$

$$\begin{array}{rcll} o\ \mathsf{finally}\ e\ /_i\ \mu & \to & e;\ o\ /_i\ \mu & \text{(finally)} \\ \mathsf{raise}\ (\varepsilon\, v)\ \mathsf{handle}\ x \succ e\ \mathsf{reraise}\ /_i\ \mu & \to & e[x \Leftarrow \varepsilon\, v];\ \mathsf{raise}\ (\varepsilon\, v)\ /_i\ \mu & \text{(reraise)} \end{array}$$

Figure 8: Syntax and semantics of finally and reraise

8.1 Types and constraints

In MLIF, the grammar of types, rows, alternatives and levels is extended with *type variables.* (We let α range over type variables of all four kinds; no ambiguity will arise.) Furthermore, Rémy's [19] row syntax is introduced, turning rows into finite lists of bindings from exception names to alternatives, terminated with a row variable.

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \mathsf{unit} \mid \mathsf{int}^\lambda \mid (\tau \xrightarrow{\pi\ [\rho]} \tau)^\lambda \mid \tau\ \mathsf{ref}^\lambda \mid \rho\ \mathsf{exn}^\lambda \\ \rho & ::= & \alpha \mid (\varepsilon : \eta; \rho) \\ \eta & ::= & \alpha \mid \mathsf{Abs} \mid \mathsf{Pre}\ \pi \\ \lambda, \pi & ::= & \alpha \mid \ell \end{array}$$

The variable-free types (resp. rows, alternatives, levels) of MLIF are isomorphic to the types (resp. rows, alternatives, levels) of MLIF_0; we identify them and refer to them as *ground.* Then, *constraints* are defined as follows:

$$\begin{array}{rcl} C & ::= & \mathbf{true} \mid C \wedge C \mid \exists \alpha. C \\ & \mid & \tau \leq \tau \mid \rho \leq \rho \mid \eta \leq \eta \mid \lambda \leq \lambda \\ & \mid & \lambda \lhd \tau \mid \sqcup \rho \leq \lambda \mid \lambda \leq \sqcap \rho \mid \tau \blacktriangleleft \lambda \end{array}$$

The constraint forms on the first line are standard [12]. Those on the second line are subtyping constraints; those on the third line are custom constraint forms, which correspond to the notions developed in sections 5 and 7.2. We omit the sorting rules necessary to ensure that terms and constraints involving rows are well-formed; see [19].

Let a *ground assignment* ϕ map every type variable α to a ground type, row, alternative, or level, according to its kind. The meaning of terms and constraints under an assignment ϕ is defined in the obvious way. We write $C \Vdash C'$ (read: C entails C') if and only if every assignment ϕ which satisfies C satisfies C' as well.

Let a *type scheme* be a triple of a set of quantifiers $\bar{\alpha}$, a constraint C and a type τ; we write $\sigma = \forall \bar{\alpha}[C].\tau$. The type variables in $\bar{\alpha}$ are bound in σ; type schemes are considered equal modulo α-conversion. By abuse of notation, a type τ may be viewed as a type scheme $\forall \varnothing[\mathbf{true}].\tau$. An *environment* Γ is a partial mapping from program variables to type schemes.

8.2 Typing rules

The typing rules for MLIF are given in figure 9. They look very similar to those of MLIF_0; let us briefly discuss the differences. We restrict our attention to *source* expressions, i.e. Core ML expressions which do not contain memory locations; this is enough for our purposes. Thus, typing judgements no longer contain a memory environment M. Every judgement begins with a constraint C which represents an assumption about its free type variables; for the judgement to be valid, C must be satisfiable. (We omit C when it is **true**.) Constrained type schemes are introduced by E-LET, which performs generalization, and eliminated by V-VAR, which performs instantiation. For the sake of conciseness, some rules use the binary operator $\sqcup$ on levels and on rows, as well as the unary operator $\sqcup$ on rows, as if they were part of our term syntax; we let the reader check that these notations can be de-sugared into extra meta-variables and constraints.

8.3 Non-interference

We prove the following statement by induction on type derivations, along the lines of [15].

Lemma 8.1 (Soundness) *Assume $C, \pi, \Gamma \vdash e : \tau\ [\rho]$. Let ϕ be an arbitrary ground assignment which satisfies C. Then, $\phi(\pi), \phi(\Gamma), \varnothing \vdash e : \phi(\tau)\ \ [\phi(\rho)]$ holds in MLIF_0.*

(We do not define $\phi(\Gamma)$ here; see [15].) In particular, every ground typing judgement in MLIF is also a valid judgement in MLIF_0. This allows us to lift our non-interference result to MLIF. That is, the statement of theorem 6.1 remains valid if $(x \mapsto t) \vdash e : \mathsf{int}^\ell$ and $\vdash v_i : t$ are read as MLIF typing judgements.

The typing rules given in figure 9 do not necessarily allow deriving *ground* typing judgements about every expression. However, it is easy to enrich the system with rules similar to HM(X)'s $\exists$-INTRO and WEAKEN [20, 15], which allow specializing a non-ground judgement to any of its ground instances.

8.4 Type inference

It is easy to check that there exists a type inference algorithm which computes principal types for MLIF. Sulzmann *et al.* [20] show how to derive a set of type inference rules from a set of typing rules similar to ours. The main point that remains to be settled is whether constraint solving is decidable.

As explained in section 5.2, our subtyping relation is atomic; constraint solving for atomic subtyping is decidable and well understood [18]. The introduction of rows is essentially orthogonal to other constraint solving issues [5, 14]. Lastly, our custom constraint forms can be solved in a "lazy" manner. That is, a constraint of the form $\lambda \lhd \alpha$, $\alpha \blacktriangleleft \lambda$, $\sqcup \alpha \leq \lambda$ or $\lambda \leq \sqcap \alpha$ remains suspended as long as nothing is known about α, and is decomposed into a number of subconstraints only when α is unified with a non-variable term τ or row ρ. Further details, including proofs and algorithms, will be given in a later paper.

9 Examples

We intend to integrate MLIF into a realistic programming language, such as Caml-Light [9]. In this section, we give a taste of that by describing the principal type schemes inferred for some library functions by our prototype implementation. We use Caml-Light syntax, which can be easily de-sugared into Core ML.

We omit type annotations on top of $\to$ when they are unconstrained, anonymous type variables. Because none of

Values

$$\textsc{v-Unit}\quad C,\Gamma \vdash () : \mathsf{unit}$$

$$\textsc{v-Int}\quad C,\Gamma \vdash k : \mathsf{int}^*$$

$$\textsc{v-Var}\quad \frac{\Gamma(x) = \forall\bar{\alpha}[D].\tau \qquad C \Vdash \exists\bar{\alpha}.D}{C \wedge D, \Gamma \vdash x : \tau}$$

$$\textsc{v-Abs}\quad \frac{C,\pi,\Gamma[x \mapsto \tau'][f \mapsto (\tau' \xrightarrow{\pi\ [\rho]} \tau)^\lambda] \vdash e : \tau\ \ [\rho]}{C,\Gamma \vdash \mathsf{fix}\, f.\lambda x.e : (\tau' \xrightarrow{\pi\ [\rho]} \tau)^\lambda}$$

$$\textsc{v-Exn}\quad \frac{C,\Gamma \vdash v : \mathit{typexn}(\varepsilon)}{C,\Gamma \vdash \varepsilon\, v : (\varepsilon : \mathsf{Pre}\, *; *)\ \mathsf{exn}^*}$$

$$\textsc{v-Sub}\quad \frac{C,\Gamma \vdash v : \tau' \qquad C \Vdash \tau' \leq \tau}{C,\Gamma \vdash v : \tau}$$

Expressions

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

$$\textsc{e-App}\quad \frac{C,\Gamma \vdash v_1 : (\tau' \xrightarrow{\pi\sqcup\lambda\ [\rho]} \tau)^\lambda \qquad C,\Gamma \vdash v_2 : \tau' \qquad C \Vdash \lambda \lhd \tau}{C,\pi,\Gamma \vdash v_1\, v_2 : \tau\ \ [\rho]}$$

$$\textsc{e-Ref}\quad \frac{C,\Gamma \vdash v : \tau \qquad C \Vdash \pi \lhd \tau}{C,\pi,\Gamma \vdash \mathsf{ref}\ v : \tau\ \mathsf{ref}^*\ \ [*]}$$

$$\textsc{e-Assign}\quad \frac{C,\Gamma \vdash v_1 : \tau\ \mathsf{ref}^\lambda \qquad C,\Gamma \vdash v_2 : \tau \qquad C \Vdash \pi \sqcup \lambda \lhd \tau}{C,\pi,\Gamma \vdash v_1 := v_2 : \mathsf{unit}\ \ [*]}$$

$$\textsc{e-Deref}\quad \frac{C,\Gamma \vdash v : \tau'\ \mathsf{ref}^\lambda \qquad C \Vdash \tau' \leq \tau \qquad C \Vdash \lambda \lhd \tau}{C,\pi,\Gamma \vdash\ !v : \tau\ \ [*]}$$

$$\textsc{e-Raise}\quad \frac{C,\Gamma \vdash v : \rho\ \mathsf{exn}^\lambda \qquad C \Vdash \pi \sqcup \lambda \leq \sqcap\rho}{C,\pi,\Gamma \vdash \mathsf{raise}\ v : *\ \ [\rho]}$$

$$\textsc{e-Let}\quad \frac{C \wedge D,\Gamma \vdash v : \tau' \qquad C,\pi,\Gamma[x \mapsto \forall\bar{\alpha}[D].\tau'] \vdash e : \tau\ \ [\rho] \qquad \bar{\alpha} \cap \mathrm{fv}(C,\Gamma) = \varnothing}{C \wedge \exists\bar{\alpha}.D,\pi,\Gamma \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e : \tau\ \ [\rho]}$$

$$\textsc{e-Bind}\quad \frac{C,\pi,\Gamma \vdash e_1 : \tau'\ \ [\rho_1] \qquad C,\pi \sqcup (\sqcup\rho_1),\Gamma[x \mapsto \tau'] \vdash e_2 : \tau\ \ [\rho_2]}{C,\pi,\Gamma \vdash \mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2 : \tau\ \ [\rho_1 \sqcup \rho_2]}$$

$$\textsc{e-Handle}\quad \frac{C,\pi,\Gamma \vdash e_1 : \tau\ \ [\varepsilon : \mathsf{Pre}\,\pi';\rho] \qquad C,\pi \sqcup \pi',\Gamma[x \mapsto \mathit{typexn}(\varepsilon)] \vdash e_2 : \tau\ \ [\varepsilon : \eta;\rho] \qquad C \Vdash \pi' \lhd \tau}{C,\pi,\Gamma \vdash e_1\ \mathsf{handle}\ \varepsilon\, x \succ e_2 : \tau\ \ [\varepsilon : \eta;\rho]}$$

$$\textsc{e-HandleAll}\quad \frac{C,\pi,\Gamma \vdash e_1 : \tau\ \ [\rho_1] \qquad C,\pi \sqcup (\sqcup\rho_1),\Gamma[x \mapsto \rho_1\ \mathsf{exn}^*] \vdash e_2 : \tau\ \ [\rho_2] \qquad C \Vdash (\sqcup\rho_1) \lhd \tau}{C,\pi,\Gamma \vdash e_1\ \mathsf{handle}\ x \succ e_2 : \tau\ \ [\rho_2]}$$

$$\textsc{e-Sub}\quad \frac{C,\pi,\Gamma \vdash e : \tau'\ \ [\rho'] \qquad C \Vdash \tau' \leq \tau \qquad C \Vdash \rho' \leq \rho}{C,\pi,\Gamma \vdash e : \tau\ \ [\rho]}$$

Figure 9: The type system MLIF

the type schemes below has free type variables, we omit the universally quantified variables after $\forall$.

We have not explained how to include datatype declarations in the language. Since we already have product and sum types, this should be straightforward. Let us assume the type constructor list is declared as follows:

```
type ('a, 'b) list = <'b>
  | []
  | (::) of 'a * ('a, 'b) list
```

In $\alpha\,\mathsf{list}^\beta$, the parameter α is the type of the list's elements, as usual, while β is a security level. The annotation `<'b>` on the right-hand side is meant to indicate that β is the security annotation carried by the sum type. Our first example function computes the length of a list:

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

A valid type scheme for `length` is $\forall[\alpha \leq \beta].\, *\ \mathsf{list}^\alpha \longrightarrow \mathsf{int}^\beta$. As expected, the result's security annotation β does not depend on the type of the list's elements. The constraint $\alpha \leq \beta$ describes the information flow induced by the function: the length of a list contains some information about its structure. This type scheme is in fact equivalent to $\forall[].\, *\ \mathsf{list}^\alpha \longrightarrow \mathsf{int}^\alpha$, a simplification which our implementation performs automatically.

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

`iter` applies `f` successively to every element of a list. Its inferred type scheme is

$$\forall[\sqcup\gamma \leq \beta].(\alpha \xrightarrow{\beta\ [\gamma]} *)^\beta \longrightarrow \alpha\,\mathsf{list}^\beta \xrightarrow{\beta\ [\gamma]} \mathsf{unit}$$

Here, γ represents `f`'s effect. Because `iter` does not throw any exceptions of its own, γ is also `iter`'s effect. β is `f`'s *pc* parameter. It must dominate `iter`'s own *pc* parameter (because `f` is invoked by `iter`), the list's security level (because gaining control tells `f` that the list is nonempty) and $\sqcup\gamma$ (because gaining control tells `f` that its previous invocation terminated normally).

```
let incr r =
  r := !r + 1
```

`incr` has $\forall[].\mathsf{int}^\alpha\ \mathsf{ref}^\alpha \xrightarrow{\alpha\ [*]} \mathsf{unit}$ as principal type scheme. Indeed, by E-ASSIGN, the security level of the reference's contents must dominate both `incr`'s *pc* parameter and the reference's own security level. We now re-implement `length` in imperative style:

```
let length' l =
  let count = ref 0 in
  iter (fun () -> incr count) l;
  !count
```

We obtain $\forall[].*\ \mathsf{list}^\alpha \xrightarrow{\alpha\ [*]} \mathsf{int}^\alpha$. This appears more restrictive than `length`'s type scheme: the result's security level must now be greater than or equal to the function's *pc* parameter. However, the difference is only superficial; it can be checked that both types in fact have the same expressive power. Formalizing this claim, and understanding its consequences, are left for future work. We continue with a few library functions which deal with association lists.

```
let rec mem_assoc x = function
  | []          -> false
  | (y, _) :: l ->
      if x = y then true else mem_assoc x l
```

Because `mem_assoc`'s result reveals information about both the structure of the list and the keys stored in it, we obtain:

$$\forall[\alpha \blacktriangleleft \beta].\alpha \to (\alpha \times *)\ \mathsf{list}^\beta \to \mathsf{bool}^\beta$$

The constraint $\alpha \blacktriangleleft \beta$, which arises due to the use of polymorphic equality, specifies that β must be an upper bound for all security annotations which occur in the type of the keys.

```
let rec assoc x = function
  | []          -> raise Not_found
  | (y, d) :: l -> if x = y then d else assoc x l
```

`assoc` returns the piece of data associated with a given key. If no such key exists, `Not_found` is raised, as reflected in `assoc`'s effect:

$$\forall[\alpha \blacktriangleleft \beta, \beta \lhd \gamma, \beta \leq \delta].\alpha \to (\alpha \times \gamma)\ \mathsf{list}^\beta \xrightarrow{\delta\ [\mathsf{Not\_found}:\ \delta;\ *]} \gamma$$

Here, as in `mem_assoc`, β represents the information associated with the list's structure and keys. Because this information is reflected both in `assoc`'s normal and exceptional results, the type system requires $\beta \lhd \gamma$ and $\beta \leq \delta$.

Lastly, we re-implement `mem_assoc` in terms of `assoc`, using an exception handler:

```
let mem_assoc' x l =
  try
    let _ = assoc x l in
    true
  with Not_found ->
    false
```

As in the case of `length` vs. `length'`, the new type scheme requires the result's security level to be greater than or equal to the function's *pc* parameter:

$$\forall[\alpha \blacktriangleleft \beta].\alpha \to (\alpha \times *)\ \mathsf{list}^\beta \xrightarrow{\beta\ [*]} \mathsf{bool}^\beta$$

This betrays the fact that the function's implementation uses effects, but does not otherwise restrict its applicability.

10 Discussion

The reader may notice that normal and exceptional results are not dealt with in a symmetric way by our type system. Indeed, in a typing judgement $pc, \Gamma, M \vdash e : t\ [r]$, the row r associates a security level with every exception name, so as to record how much information is gained by observing that particular exception. However, no information level is explicitly associated with normal termination. Instead, the typing rule for sequential composition, namely E-BIND, uses $\sqcup r$ as an approximation of it.

Myers' [10, 11] sets of path labels X, on the other hand, record the security level associated with normal termination under a special label $\underline{n}$, which is then used in the sequential composition rule. It is, however, typically an upper bound for the value reached by *pc* inside every sub-expression of the expression at hand, so this design alone would make the type system very restrictive. To prevent that, Myers adds a non-syntax-directed rule, the *single-path* rule, stating that $X[\underline{n}]$ can be reset to $\underline{\emptyset}$ if the expression at hand can be shown to always terminate normally.

Our system doesn't need the single-path rule: indeed, when all entries in r_1 are Abs, then $\sqcup r_1$ is the least element of $\mathcal{L}$, and E-BIND typechecks e_1 and e_2 at a common *pc*, as desired. Myers' system is more precise than ours in a few cases, which involve expressions that *never* terminate normally; experience will tell how common they are. The single-path rule requires counting the number of non-Abs entries in a row; in the presence of row variables, this requires new (and quite heavy) constraint forms, which is why we avoid it. This difficulty does not arise in Myers' framework because it relies on Java's explicit, monomorphic throws clauses.

There exists a simple monadic encoding of exceptions into sums. Thus, it is possible, in principle, to derive a type system for exceptions out of a type system that can handle sums. This approach sounds interesting, because it is systematic and promises to yield a symmetric treatment of normal vs. exceptional results. However, some experiments show that, in order to obtain acceptable precision in the end, the treatment of sums that is chosen as a starting point must be very accurate (much more so than the one given in this paper). We leave it as a topic of future research.

Our main direction for future work is to create a full implementation of the system on top of Caml-Light and to assess its usability through a number of case studies. We also plan to study a variant of Core ML where exceptions are second-class citizens, i.e. where $\mathsf{raise}\ x$ is disallowed. In exchange for this slight loss of expressive power, we hope to be able to use a simpler type and constraint language.

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon 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, January 1999. URL: http://www.soe.ucsc.edu/~abadi/Papers/flowpopl.ps.

[2] Martín Abadi, Butler Lampson, and Jean-Jacques 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://www.soe.ucsc.edu/~abadi/Papers/make-preprint.ps.

[3] D. E. Bell and Leonard J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report MTR-2997, The MITRE Corp., Bedford, Massachusetts, July 1975. URL: http://www.mitre.org/resources/centers/infosec/infosec.html.

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

[5] Manuel Fähndrich. BANE*: A Library for Scalable Constraint-Based Program Analysis*. PhD thesis, University of California at Berkeley, 1999. URL: http://research.microsoft.com/~maf/diss.ps.

[6] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, June 1993. URL: http://www.cs.rice.edu/CS/PLT/Publications/pldi93-fsdf.ps.gz.

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

[8] Nevin Heintze and Jon 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, January 1998. URL: http://cm.bell-labs.com/cm/cs/who/nch/slam.ps.

[9] Xavier Leroy, Damien Doligez, et al. The Caml Light system, release 0.74. URL: http://caml.inria.fr/, 1997.

[10] Andrew C. Myers. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages*, pages 228–241, San Antonio, Texas, January 1999. ACM Press. URL: http://www.cs.cornell.edu/andru/papers/popl99/myers-popl99.ps.gz.

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

[12] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999. URL: http://www.cs.mu.oz.au/~sulzmann/publications/tapos.ps.

[13] François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, 2000. URL: http://pauillac.inria.fr/~xleroy/publi/exceptions-toplas.ps.gz.

[14] François Pottier. `Wallace`: an efficient implementation of type inference with subtyping, February 2000. URL: http://pauillac.inria.fr/~fpottier/wallace/.

[15] François Pottier. A semi-syntactic soundness proof for HM(X). Research Report 4150, INRIA, March 2001. URL: ftp://ftp.inria.fr/INRIA/publication/RR/RR-4150.ps.gz.

[16] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 46–57, September 2000. URL: http://pauillac.inria.fr/~fpottier/publis/fpottier-conchon-icfp00.ps.gz.

[17] François Pottier and Vincent Simonet. Information flow inference for ML. Full version. URL: http://pauillac.inria.fr/~fpottier/publis/fpottier-simonet-popl02-long.ps.gz, July 2001.

[18] Jakob Rehof. Minimal typings in atomic subtyping. In *Conference Record of the 24th ACM Symposium on Principles of Programming Languages*, pages 278–291, Paris, France, January 1997. URL: http://research.microsoft.com/~rehof/popl97.ps.

[19] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993. URL: ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/taoop1.ps.gz.

[20] Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Research Report ACRC–99–009, University of South Australia, School of Computer and Information Science, July 1999. URL: http://www.ps.uni-sb.de/~mmueller/papers/hm-constraints.ps.gz.

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

[22] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996. URL: http://www.cs.nps.navy.mil/people/faculty/volpano/papers/jcs96.ps.Z.

[23] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995. URL: http://www.cs.rice.edu/CS/PLT/Publications/lasc95-w.ps.gz.

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

[25] Steve Zdancewic and Andrew C. Myers. Secure information flow and CPS. In David Sands, editor, *Proceedings of the 2001 European Symposium on Programming (ESOP'01)*, Lecture Notes in Computer Science. Springer Verlag, April 2001. URL: http://www.cs.cornell.edu/zdance/lincont.ps.